CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

BACHELORARBEIT

(Individuelles Projekt)

Mathematische Simulation aktueller Strukturannahmen über den Prozess visuell-räumlicher Aufmerksamkeit

Christian Hinrichs

Erstprüfender Prof. Dr. Helmut Hildebrandt
Zweitprüfender Prof. Dr. Claus Möbus

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Oldenburg, am 9. Juli 2007

Christian Hinrichs

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel werden die Motivation, die Ziele sowie der Aufbau der vorliegenden Arbeit beschrieben.

1.1 Motivation

Im Bereich der klinischen Psychologie werden Probleme bezüglich der Funktionsweise des zentralen Nervensystems (ZNS) und der kognititven Funktionen oftmals per Rückschlussverfahren analysiert. Dazu wird das ZNS als *Blackbox* betrachtet, welche wohldefinierte Eingaben und Ausgaben hat, deren interne Verarbeitung jedoch weitgehend unbekannt ist. Durch die Analyse von fehlerhaften Systemen (z.B. Patienten mit Hirnschädigungen) lassen sich dann unter Beachtung der Art und Lokalisation des Fehlers Rückschlüsse auf die Arbeitsweise des Systems ziehen. Ein Teilgebiet dieser Disziplin ist die Untersuchung der visuellen Wahrnehmung. Die zu klärenden Problemstellungen erstrecken sich von der Bildanalyse über die Aufmerksamkeit bis hin zu der Interpretation einer wahrgenommenen Szene. Das in dieser Arbeit behandelte Thema entstammt der Diskussion über die Art der kognitiven Repräsentation von einzelnen Objekten.
Die visuelle Wahrnehmung von Objekten basiert auf Raumkoordinaten, welche die Objekte in eine räumliche Relation zum Beobachter setzen. Zusätzlich ist jedoch auch eine objektzentrierte Repräsentation denkbar, welche den Objekten jeweils ein eigenes Koordinatensystem zuschreibt, in dem sie beschrieben werden können. Die Frage ist nun, ob für das menschliche Sehen rein beobachterzentrierte Koordinaten ausreichen, oder ob zusätzlich eine objektzentrierte Beschreibung notwendig ist. Ein Ansatz zur Klärung dieser Frage besteht darin, Menschen mit Hirnverletzungen zu untersuchen und Situationen zu analysieren, in denen beide Repräsentationen zum Einsatz kommen.

1.2 Ziele dieser Arbeit

Mit MORSEL hat Michael C. Mozer ein mathematisches Modell zur aufmerksamkeitsgetriebenen Objekterkennung erstellt. Die Abkürzung steht für *multiple object recognition and attentional selection* und stellt ein konnektionistisches System zur visuellen Auswahl und Kategorisierung von Objekten dar (vgl. [11]). Entgegen dem sich in der neurowissenschaftlichen Forschung abzeichnenden Trend zur Annahme, eine objektzentrierte Repräsentation sei für die visuelle Wahrnehmung erforderlich, simuliert Mozer mit seinem Modell eine rein auf egozentrierten Koordinaten basierende Objekterkennung (siehe [12]). In dieser Arbeit soll zunächst der für die Aufmerksamkeitssteuerung zuständige Teil aus MORSEL rekonstruiert und mit Mozers Simulationen nachvollzogen werden. Anschließend sollen dann Ergebnisse aktueller Hirnforschung simuliert werden, um nachzuprüfen, ob Mozers These auch für diese neuen Daten aufrecht erhalten werden kann. In einem dritten Schritt soll dann die Aufmerksamkeitsarchitektur des Systems für neuere Ergebnisse der Hirnforschung erweitert werden.

1.3 Aufbau

Die vorliegende Arbeit wird wie folgt gegliedert sein: Zunächst werde ich in Kapitel 2 die psychologischen und physiologischen Grundlagen für das Verständnis der Problemstellung erläutern. Kapitel 3 wird sich mit dem bereits erwähnten System MORSEL beschäftigen und die Struktur und Arbeitsweise dieses Modells beschreiben. Kapitel 4 wird dann die Anforderungen an die in dieser Arbeit zu erstellende Software formulieren, welche in Kapitel 5 im Entwurf für das Produkt umgesetzt werden. Die Implementierung des Entwurfes wird in Kapitel 6 beschrieben. Nachfolgend enthält Kapitel 7 die nach dem Entwurf im Laufe der Arbeit hinzugekommenen Erweiterungen. Kapitel 8 enthält dann schließlich die Gesamtübersicht der Anwendung inklusive der Hinweise zur Installation. Am Schluss wird Kapitel 9 die vorliegende Arbeit noch einmal als Ganzes betrachten und das erstellte Produkt hinsichtlich der Anforderungen bewerten.
Anhang A enthält die Dateien, über welche die Software in ihrem Entwicklungskontext konfiguriert wurde. Anhang B enthält die Dateien, welche von der erstellten Anwendung produziert werden. In Anhang C ist zum Schluss der Inhalt der beigefügten CDROM beschrieben.

Kapitel 2

Grundlagen

Dieses Kapitel enthält die für das Verständnis der Arbeit notwendigen Grundlagen der neurowissenschaftlichen Forschung.

2.1 Neglect

Neglect ist eine schwache Form der halbseitigen Nichtwahrnehmung des eigenen Körpers nach einseitiger Schädigung des Gehirns. Menschen mit dieser Krankheitsform haben eine Wahrnehmungsstörung, welche sich über diverse Sinne erstrecken kann, jedoch immer auf eine Seite des Körpers beschränkt ist. Dabei liegt nicht zwingend eine Anosognosie vor, d.h. die Patienten können sich durchaus dieser Tatsache bewusst sein, und sind dabei nicht im Wachbewusstsein oder der Orientiertheit eingeschränkt (vgl. [7, S. 215]). Die Seite der Vernachlässigung ist dabei grundsätzlich kontraläsional. Neurophysiologische bzw. psychologische Erklärungsansätze für den Effekt der Vernachlässigung beinhalten die folgenden Punkte (adaptiert von [8, Folie 2]):

- Aufmerksamkeitsorientierung: Die Aufmerksamkeit tendiert zur ipsiläsionalen Seite, ein Wechsel zu einem weiter kontraläsional gelegenen Reiz ist nicht möglich.
- Repräsentationsdefizit: Es fehlt die mentale Repräsentation kontralateraler Reize.
- Transformationsfehler: Die Transformation der sensorischen Informationen vom retinalen in andere Koordinatensysteme gelingt nicht oder nur teilweise.
- Koordinatenrotation: Die mentalen Koordinatensysteme sind um die vertikale Körperachse des Betrachters zur ipsiläsionalen Seite rotiert.

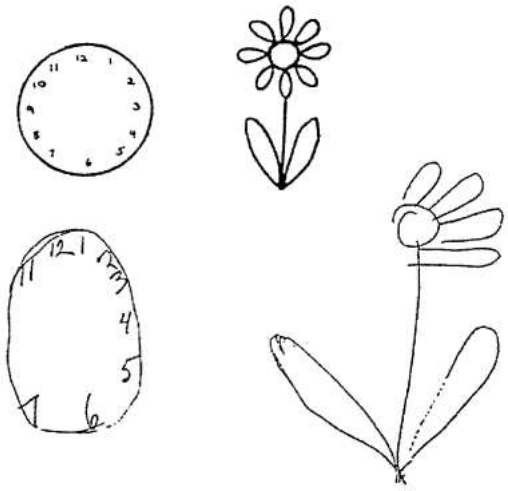


Abbildung 2.1: Beispiel eines linksseitigen Neglect, entnommen [2, S. 308]

Ich werde die Erläuterungen an dieser Stelle auf den visuellen und mentalen Neglect beschränken, da die anderen Formen für diese Arbeit nicht relevant sind.

2.1.1 Visueller Neglect

Diese Art der Einschränkung wird auch als hemianopische Aufmerksamkeitsschwäche bezeichnet (abgeleitet von Hemianopsie: halbseitiger Gesichtsfeldausfall) und beschreibt das Phänomen, dass die Patienten zwar uneingeschränkt sehen können, jedoch unter bestimmten Umständen eine Seite ihres Gesichtfeldes vernachlässigen. So erscheinen beide Gesichtsfelder zunächst intakt, wenn sie sukzessive geprüft werden. Bei gleichzeitiger Stimulation beider Seiten jedoch wird einer der beiden Stimuli nicht wahrgenommen.
Typische Versuche zur Detektion und Analyse dieses Defekts sind beispielsweise Suchaufgaben, in welchen die Patienten in einer Menge von Objekten spezifische Merkmale entdecken sollen, oder Aufgaben, bei denen die Probanden Zeichnungen von bekannten Ojekten anfertigen oder vorliegende Zeichnungen kopieren sollen. In Abbildung 2.1 ist eine solche Kopieraufgabe zu sehen. Im oberen Bereich der Abbildung sind die zu kopierenden Originalzeichnungen zu sehen, darunter die von den Patienten erstellten. In diesem Fall hat der Patient systematisch die linke Seite der Figuren vernachlässigt, was auf einen linksseitigen Neglect hinweist.

2.1.2 Mentaler Neglect

Diese Form des Neglect betrifft keine Sinnesorgane direkt, sondern spielt sich in der mentalen Repräsentation ab. So wurden Einzelfälle beobachtet, in welchen Patienten mit rechtsseitiger Schädigung des Gehirns Schwierigkeiten damit hatten, sich korrekt an alle Details erlebter Situationen zu erinnern. Sie vernachlässigten bei der Beschreibung ihrer Erinnerung einen Teil jener Details, welche sich aus ihrem derzeitigen Beobachterstandpunkt in der linken Hälfte des Außenraumes befanden. Bei der Vorstellung etwa, sie blickten auf den Dom von Mailand, waren sie nicht in der Lage, die linke Hälfte der eigentlich für sie vertrauten Umgebung des Platzes um den Dom korrekt zu beschreiben. Die rechte Seite bereitete indessen keine Probleme. Als sie dann gebeten wurden, sich den Platz von der anderen Seite (also von hinten betrachtet) vorzustellen, drehte sich das Problem um. Auf einmal konnten sie die Seite der Szene, welche vorher so fehlerhaft geschildert wurde, gut beschreiben, während die andere Seite, bei der es vorher keine Probleme gab, plötzlich nicht mehr zu erinnern war.
Ich möchte hier nicht weiter auf die Bedeutung dieser Funde eingehen, sondern zur vertiefenden Lektüre auf [7, S. 216] und [3] verweisen.

2.2 Lokalisation

Der genaue Ort der Neglect verursachenden Hirnschädigungen spielt eine große Rolle in Bezug auf die Deutung der Experimente und Untersuchungen. Nach umfangreichen Analysen wurde als hauptsächlicher Bereich der rechtsseitig parietale Teil des Gehirns festgelegt (ca. Brodmann-Areal 7, siehe [7, S. 217] und [18, S. 146]). Die Autoren aus [4] postulieren „ein neuronales System, das der lateral gerichteten Aufmerksamkeit zugrunde liegt und welches frontale und limbische kortikale Areale, die Stammganglien und als zentrale Struktur die parietale Area 7 einschließt“ (entnommen [7, S. 217]). Diese Angaben sind insofern interessant, als dass sie auf eine unsymmetrische Aufgabenverteilung der Verarbeitung sensorischer Information im Gehirn hinweisen, was bei der Modellierung eines nicht nur funktional, sondern auch strukturell äquivalenten Systems von Bedeutung ist. In [7, S. 212] heißt es dazu: „Aus dem starken Überwiegen rechtshirniger über linkshirnige Läsionen wird auf eine funktionelle Asymmetrie für die integrative (oder assoziative) Verarbeitung von somatosensorischen Afferenzen mit ebenfalls rechtsdominanten Funktionen der räumlichen Orientierung geschlossen.“

2.3 Auswirkungen auf die Objektrepräsentation

Phänomene wie „Neglect“ helfen bei der Diskussion um die Funktionsweise der mentalen Objektrepräsentation, da sie jeweils sehr spezifische Einschrän-

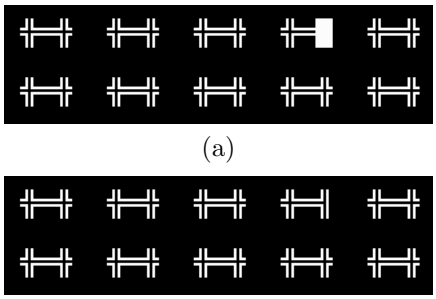
kungen und Auswirkungen auf die Wahrnehmung haben. Sie ermöglichen es, neue Theorien zu bilden oder vorhandene zu verwerfen. So deuten zum Beispiel Effekte wie die oben erwähnten Fehler beim Kopieren von Figuren darauf hin, dass eine objektzentrierte Repräsentation existitert, und dass diese darüberhinaus für die korrekte Wahrnehmung von Objekten notwendig ist. Dennoch liefern diese Hinweise noch keine Beweise für die Existenz solcher Mechanismen, was durch den Gegenversuch von Mozer deutlich wird, indem er in seinem Modell eine Objekterkennung völlig ohne objektspezifische Koordinatensysteme durchführt (vgl. [12]). Da jedoch auch Mozer durch seine Untersuchungen nur Hinweise liefert, bleibt die ursprüngliche Diskussion bestehen. Um die Frage weiter zu klären, werden spezifischere Experimente durchgeführt, welche den einen oder anderen Standpunkt entweder belegen oder bekräftigen sollen.
Im Folgenden werde ich ein Experiment von Hildebrandt et.al. vorstellen, welches für die vorliegende Aufgabenstellung grundlegend ist und direkt an die beschriebene Diskussion anschließt.

2.4 Experiment: Visuelle Suche bei Schlaganfallpatienten

In dem Artikel „*Visual search for item- and array-centered locations in patients with left middle cerebral artery stroke*“ [9] ist ein Experiment von Hildebrandt et.al. beschrieben, bei dem Patienten, mit Schlaganfall im Bereich der linken Hemisphäre des Gehirns, auf ihre Fähigkeiten bei der visuellen Suche geprüft wurden. Die Patienten hatten negative Befunde für Hemianopsie und Neglect, womit dieser Versuch auf eine mehr elementare Ebene der Objektrepräsentation, nämlich die neuronale Aufmerksamkeitsarchitektur abzielt.
Die Aufgabe bestand aus der visuellen Suche nach bestimmten Abweichungen in einem 5x2 Elemente umfassenden Feld (*Array*) von Objekten. In Abbildung 2.2 ist exemplarisch ein solches Feld dargestellt. Die zu erkennenden

Abbildung 2.2: Feld von 5x2 Elementen ohne Feature

Abweichungen umfassen zwei Typen, welche in Abbildung 2.3 aufgeführt werden. Der erste Typ ermöglicht durch seine ausgeprägte Abhebung vom Hintergrund ein *paralleles* Suchparadigma (a), in dem das Feature (bzw. Merkmal), welches die Abweichung verursacht, sofort und ohne längere Su-



(a) (b)

Abbildung 2.3: Feld von 5x2 Elementen mit Feature zur (a) parallelen und (b) seriellen Suche

che erkannt wird. Der zweite erfordert eine *serielle* Suche (b), da das zu findende Feature aus einer geschlossenen statt offenen Linie am Rand eines Elementes besteht und somit weniger offensichtlich ist. Dies kann der Leser beim Betrachten der Abbildung selbst feststellen, denn in den meisten Fällen kann die Abweichung nur durch ein serielles Untersuchen der einzelnen Elemente gefunden werden (vgl. [15, S. 132ff]). Wichtig zu erwähnen ist an dieser Stelle, an welchen Positionen die Features auftreten können. Und zwar wird prinzipiell zwischen der Array-spezifischen *Position* des Merkmals, und der Item-spezifischen *Seite* unterschieden, sodass für jeden der beiden Abweichungstypen 20 verschiedene Orte des Auftretens existieren. Die Testdurchläufe wurden nach Paradigma getrennt und bestanden aus jeweils 40 Abbildungen mit und 40 ohne Feature in zufälliger Reihenfolge. Die Features waren dabei gleichmäßig verteilt, indem an jeder Position und jeder Elementseite jeweils zweimal eine Abweichung auftrat (10 Positionen $\cdot$ 2 Seiten $\cdot$ 2 Vorkommen = 40 Abbildungen mit Feature). Die Aufgabe der Probanden bestand nun darin, für ein angezeigtes Bild so schnell wie es ihnen möglich war zu entscheiden, ob eine Abweichung vorhanden war oder nicht (Position, Seite und Typ spielten hier keine Rolle). Dazu sollten sie entweder die Leertaste für *ja* oder die '0' des Ziffernblocks für *nein* drücken. 500ms nach einer gegebenen Antwort wurde das nächste Bild angezeigt. Insgesamt wurden pro Paradigma jeweils mindestens acht Durchläufe durchgeführt, wobei immer zunächst der parallele, gefolgt vom seriellen Typ getestet wurde. Neben den Schlaganfallpatienten wurde zudem eine Kontrollgruppe den gleichen Tests unterzogen, damit die Ergebnisse verglichen und interpretiert werden konnten.

Gemessen wurden während des Experiments einerseits die Anzahl der Treffer, sowie die Reaktionszeit der Patienten zu jeder einzelnen Antwort. In den Abbildungen 2.4 und 2.5 sind die Daten dieser Messungen für die Patienten (durchgehende Kurve) sowie für die Kontrollgruppe (gestrichelte Kurve) dargestellt. Die Grafiken sind jeweils in vier vertikale Bereiche unterteilt. In-

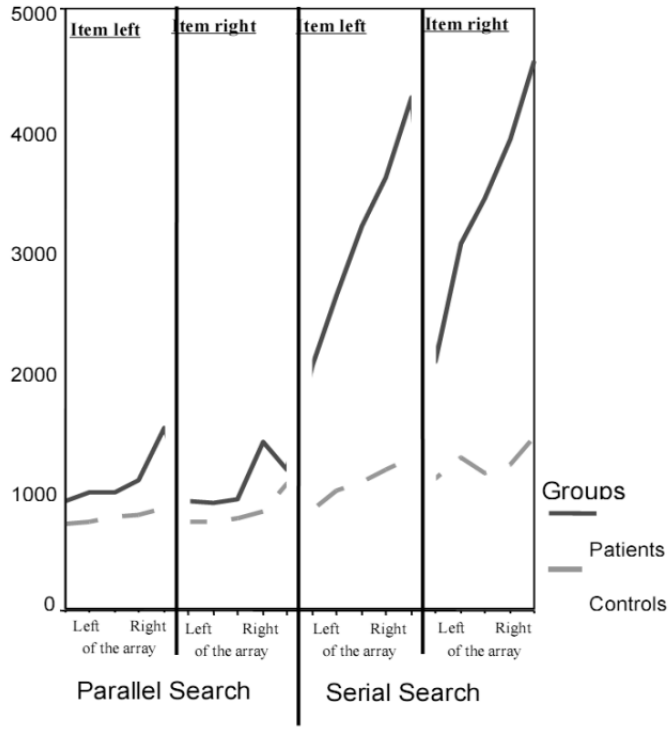


Abbildung 2.4: Reaktionszeiten, entnommen [9, S. 421]

nerhalb dieser Bereiche sind die Ergebnisse für alle Positionen im Array (von links nach rechts) dargestellt. Die Unterscheidung der Bereiche ist einerseits durch den Typ der Abweichung (Bereiche 1 und 2: parallele Suche, Bereiche 3 und 4: serielle Suche) sowie die Seite des Merkmals gegeben (Bereiche 1 und 3: linke Itemseite, Bereiche 2 und 4: rechte Itemseite), sodass die Daten für beide Typen und für jede Position und Seite separat ablesbar sind.
Als Ergebnis zeigte das Experiment die folgenden Zusammenhänge: Von links nach rechts in Arraykoordinaten steigt die Reaktionszeit der Probanden nahezu stetig an, unabhängig von der Seite des Suchmerkmals (Abb. 2.4). Eine Varianzanalyse der Daten ergab jedoch in Bezug auf die Auslassungen (also nichtgefundene Merkmale) eine Interaktion zwischen Position und Seite für die Fälle, in denen sich das Suchmerkmal auf der rechten Seite eines Elementes befand. Diese Interaktion ist umso ausgeprägter, je weiter rechts sich das Merkmal im Feld befindet. Für die Fälle mit Merkmal auf der linken Itemseite ist dieser Zusammenhang nicht nachweisbar. Das überraschende daran ist, dass ein im Feld weiter links positioniertes Merkmal mehr Auslassungen verursacht, wenn es auf der rechten Seite eines Items steht, als ein weiter rechts positioniertes Merkmal auf der linken Seite eines Items. Dieser Effekt ist in Abbildung 2.6 schematisch dargestellt.

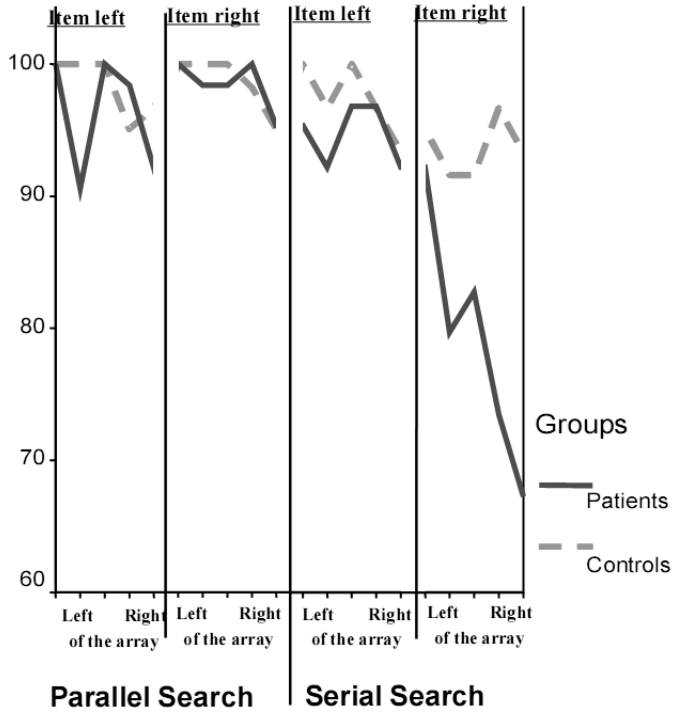


Abbildung 2.5: Prozentuale Treffer, entnommen [9, S. 421]

Dies steht der allgemeinen Einschränkung entgegen, dass ein Reiz desto mehr Auslassungen bedingt, je weiter rechts er positioniert ist. Die Autoren ziehen daraus den Schluss, dass die Seite eines Reizes einen systematischen Einfluss auf die Wahrnehmung der Patienten ausübt.

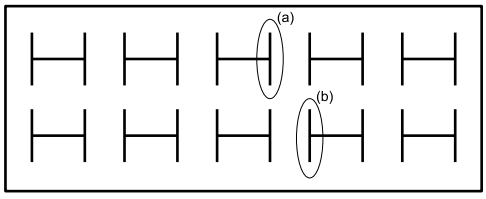


Abbildung 2.6: Veranschaulichung der Interaktion zwischen Position und Seite: Reiz (a) verursacht mehr Auslassungen als (b).

Kapitel 3

Modell MORSEL

3.1 Der Aufbau von MORSEL

Michael C. Mozer hat 1991 das Modell *multiple object recognition and attentional selection* (MORSEL) veröffentlicht, welches ein konnektionistisches System zur visuellen Auswahl und Kategorisierung von Objekten darstellt. Die Informationen aus diesem Kapitel entstammen den Quellen [11] und [12]. MORSEL besteht aus einer Reihe von Modulen, welche in einer Schichtarchitektur miteinander kommunizieren und den perzeptuellen Input einer simulierten Retina verarbeiten, um die von ihr „wahrgenommenen" Objekte zu erkennen. In der ursprünglichen Implementation war MORSEL zur Erkennung von Buchstaben und Wörtern gedacht, sodass an einigen Stellen bewusst auf überflüssige Komplexität verzichtet wurde. Die Retina ist beispielsweise als 36×6 Pixel messendes Feld konzipiert, um so insgesamt zwölf Buchstaben fassen zu können, wobei ein Buchstabe dabei 3×3 Pixel in Anspruch nimmt. Mozer hat dazu eine Schriftart entwickelt, welche darauf ausgelegt ist, einen Buchstaben möglichst eindeutig durch 3×3 Punkte beschreiben zu können.
In Abbildung 3.1 ist der strukturelle Aufbau des Systems dargestellt. Am unteren Rand der Grafik sind drei Erkennungseinheiten für bestimmte Objekteigenschaten zu sehen, welche zusammen das modulare Netzwerk BLIRNET bilden (der Name entstammt seinem Zweck: *it builds location invariant representations*). Die einzelnen Module, von denen hier Form (*Shape*), Farbe (*Color*) und Bewegung (*Motion Detection Module*) dargestellt sind, erhalten die perzeptuellen Daten als Input und extrahieren daraus die Eigenschaften, welche dem dargestellten Objekt am meisten zuzuschreiben sind. Die Extraktion erfolgt jeweils durch ein neuronales Netz, welches als Eingabe direkt die Bildpunkte der Retina erhält, und als Ausgabe ein Aktivierungsmuster von Neuronen liefert, das einer erkannten Eigenschaft entspricht (zur vertiefenden Lektüre von künstlichen neuronalen Netzen siehe Fachliteratur wie z.B. [25]). So würde ein rotes X beispielsweise im Modul für Form die Aus-

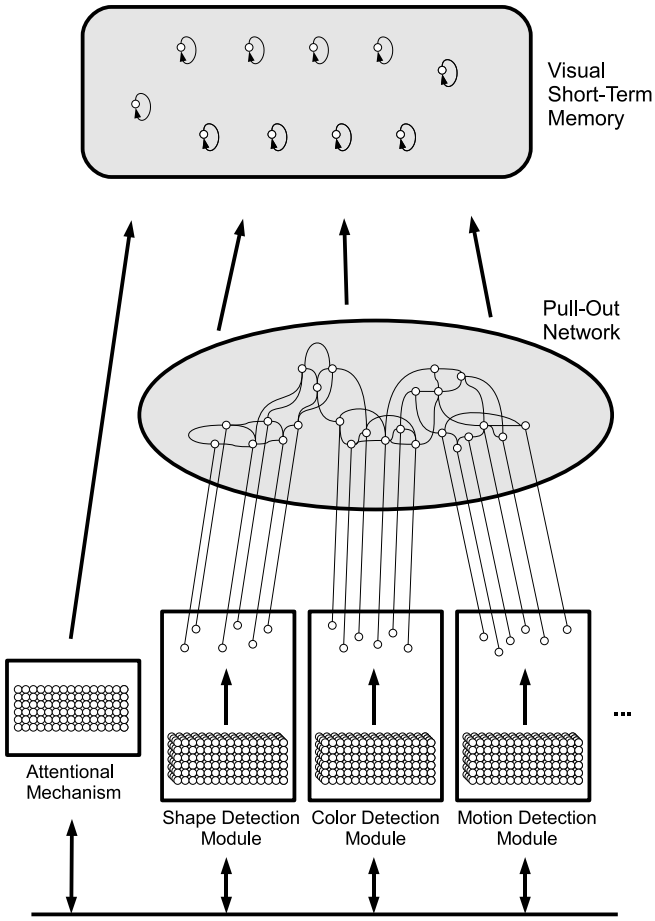


Abbildung 3.1: Der strukturelle Aufbau von MORSEL. Aus [11]

gabeneuronen in der Art aktivieren, dass sie die Eigenschaft der Form 'X' repräsentieren. Das Modul für Farbe würde eine Ausgabe erzeugen, welche der Eigenschaft 'Rot' entspricht usw. Die Kodierung der Ausgabe ist dabei implementationsspezifisch, sodass gewährleistet sein muss, dass nachfolgend geschaltete Schichten diese interpretieren können.
Daneben ist das sogenannte *Attentional Mechanism* (AM) zu sehen. Dieses Modul gehört nicht zu BLIRNET, ist aber für dessen korrekte Funktionsweise sehr wichtig, da die BLIRNET Erkennungseinheiten (oder auch Feature Detektoren) lokationsinvariant sind und jeweils immer das gesamte von der Retina repräsentierte Feld untersuchen. Dies kann zu Fehlern führen, wenn

mehrere Objekte dargestellt und erkannt werden sollen. Als Beispiel bietet sich hier das Modul für die Erkennung von Form an, dazu muss jedoch erklärt werden, wie dieses Modul arbeitet. Mozer verwendet dazu eine Untermenge von Textonen (siehe [10]), und zwar Liniensegmente in bestimmten Orientierungen: 0° ('|'), 45° ('/'), 90° ('—') und 135° ('\'). Diese Segmente können auch konkateniert werden. Zusätzlich definiert er Terminatoren, welche die Endpunkte dieser Linienzüge darstellen. So würde etwa ein 'V' als Eingabe die Features '\' und '/' in der Ausgabe aktivieren, zuzüglich einiger Terminatoren. Problematisch wird es jetzt, wenn mehrere Objekte gleichzeitig präsentiert werden. So würden etwa bei der parallelen Eingabe von 'V' und 'I' die Features '\', '/' und '|' aktiviert werden, was in der Interpretation der Ausgabe ein 'Y' ergeben könnte. Die Feature Detektoren sind also nicht in der Lage zu unterscheiden, ob erkannte Features zusammen gehören oder nicht. An dieser Stelle greift das AM ein: dieses Modul beeinflusst die Erkennung von Features, indem es bestimmte Bereiche der Eingabe unterdrückt und andere hervorhebt, und dadurch versucht, die dargestellten Objekte separat und sukzessive zu erkennen. Wie dies genau geschieht, wird im folgenden Abschnitt 3.2 erläutert.
Oberhalb der Feature Detektoren in der Grafik befindet sich das mit *Pull-Out Network* (PO) bezeichnete künstliche neuronale Netz, welches für die Interpretation der extrahierten Features zuständig ist. Wie in der Abbildung zu erkennen, enthält das PO als Eingabe eine direkte Kopie der Ausgabe der BLIRNET Module. Diese werden innerhalb des Netzes durch interne Verschaltung kombiniert und liefern als Ausgabe dasjenige Objekt (in diesem Fall denjenigen Buchstaben), welches den extrahierten Features am meisten entspricht.
In dem darüber liegenden *Visual Short-Term Memory* wird diese Information nun zusammen mit den Retina-bezogenen Ortsinformationen (welche das AM liefert) des aktuell bearbeiteten Objektes gespeichert bzw. für eine weitere Verarbeitung höher gelegenen Strukturen zur Verfügung gestellt.

3.2 Das Attentional Mechanism (AM)

Das AM simuliert in MORSEL in etwa das, was bei uns Menschen die Aufmerksamkeit darstellt. Es analysiert die gefundenen Features und selektiert diejenigen Informationen, welche vom Rest des Systems verarbeitet werden sollen. Nicht-selektierte Bereiche werden unterdrückt, sodass sich ein Aufmerksamkeitsfokus ergibt, in welchem das System agiert. Das AM erhält dabei nicht nur die Informationen der Feature Detektoren (als sogenannten exogenen Input) als Eingabe, sondern wird zusätzlich von den höher gelegenen Schichten beeinflusst, um eine sukzessive Verarbeitung der anliegenden Informationen zu ermöglichen. So würde das AM beispielsweise bei der simultanen Präsentation von zwei Buchstaben zunächst einen von bei-

den selektieren, und den zweiten unterdrücken. Das System arbeitet nun im Bereich des ausgewählten Objektes. Wurde die Information verarbeitet und die Objektdaten bis in die oberen Schichten propagiert, wird das AM angewiesen, eine andere Region zu selektieren, und die bisher bearbeitete zu unterdrücken. In Abbildung 3.2 ist der Einfluss des AM auf die Verbindung der Feature Detektoren zum *Pull-Out Network* dargestellt. In der Grafik sind das

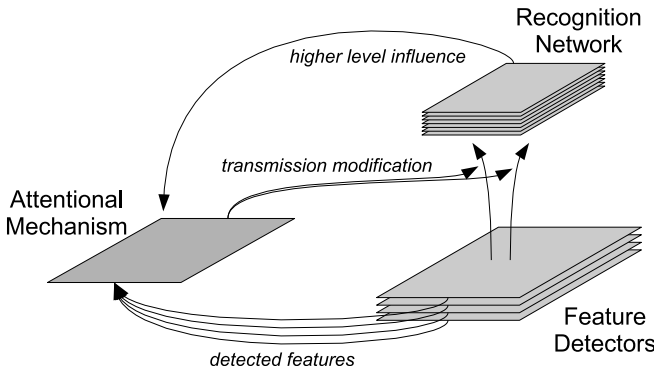


Abbildung 3.2: Der Einfluss des AM. Adaptiert von [12]

Pull-Out Network sowie das *Visual Short-Term Memory* zur Vereinfachung unter dem Begriff *Recognition Network* zusammengefasst. Das AM analysiert den exogenen Input und modifiziert auf dieser Basis die Verbindungen zwischen den Feature Detektoren und dem Erkennungsnetzerk, indem es bestimmte Bereiche unterdrückt. Im Einzelnen bedeutet dies, dass jedem Bildpunkt ein Zahlenwert zugeordnet wird, welcher die Wahrscheinlichkeit angibt, mit der die Information dieses Bildpunktes zur Weiterverarbeitung zugelassen wird. Diese Wahrscheinlichkeit nennt Mozer *transmission propability* (Übertragungswahrscheinlichkeit), oder auch *Aktivität* eines Punktes. So würden Punkte innerhalb eines selektierten Bereiches eine hohe, und die erstlichen Punkte eine niedrige Wahrscheinlichkeit erhalten. Dabei ist zu beachten, dass hier nicht zwingend eine scharfe Trennung zwischen selektierten und nicht selektierten Bereichen vorliegen muss. Ein Punkt kann somit auch zu einem gewissen Grad aktiviert sein, indem ihm eine mehr oder weniger große Übertragungswahrscheinlichkeit zugewiesen wird.
Diese Wahrscheinlichkeitswerte werden vom AM durch eine iterative Formel berechnet. Iterativ deshalb, weil diese Formel pro Bildpunkt angewendet wird und immer die im letzten Schritt berechneten Werte zur aktuellen Bestimmung miteinbezieht. Somit kann sie auch als Update-Gleichung be-

zeichnet werden. Sie setzt sich aus folgenden Termen zusammen (vgl. [11]):

$$a_{xy}(t+1) = f\left(a_{xy}(t) + exo_{xy} + \mu \sum_{\substack{i,j\in \\ NEIGH_{xy}}} [a_{ij}(t) - a_{xy}(t)] - \theta[\bar{a}(t) - a_{xy}(t)]\right)$$

Dabei ist f eine Funktion, welche die berechnete Aktivität auf das Intervall $[0, 1]$ beschränkt, indem sie außerhalb liegende Werte abschneidet:

$$f(z) = \begin{cases} 0 & \text{für } z < 0 \\ z & \text{für } 0 \leq z \leq 1 \\ 1 & \text{für } z > 1 \end{cases}$$

Das Argument dieser Funktion besteht in der Updategleichung aus vier Termen. Der erste, $a_{xy}(t)$, bezeichnet die zuvor für den Punkt (x, y) berechnete Aktivität und stellt den Basiswert dieser Gleichung dar. Er bewirkt, dass ein Punkt seine aktuelle Aktivität hält. Die folgenden Terme setzen jeweils eine von Mozer formulierte Regel um und modifizieren diese Aktivität:
exo_{xy} ist der exogene Input an der Stelle (x, y) (also ein Zahlenwert, welcher die Menge der an diesem Punkt erkannten Features repräsentiert) und realisiert die sogenannte *bias rule*, welche die Aktivität anhebt, falls für den aktuellen Punkt Features erkannt wurden.
Der darauffolgende Term

$$\mu \sum_{\substack{i,j\in \\ NEIGH_{xy}}} [a_{ij}(t) - a_{xy}(t)]$$

erfüllt die *cooperation rule*. Diese bewirkt eine Angleichung der Aktivität an den Durchschnitt der Werte der unmittelbar angrenzenden Punkte, indem für jeden Punkt aus $NEIGH_{xy}$ (die Nachbarschaft des aktuellen Punktes, bestehend aus den acht angrenzenden Punkten) jeweils die Differenz zwischen seiner Aktivität und der des aktuellen Punktes berechnet und aufsummiert wird, und anschließend diese Summe gewichtet als Summand in die Updategleichung einfließt. Der Faktor μ ist dabei eine positive variable Gewichtungsgröße.
Der letzte Term stellt die *competition rule* dar und veranlasst den Punkt, sich der durchschnittlichen Aktivität des gesamten AM anzunähern, indem die Differenz zwischen der Durchschnittsaktivität des AM und der des aktuellen Punktes gebildet wird und gewichtet als Subtrahend die Updategleichung beeinflusst. Der Gewichtungsfaktor θ ist ebenfalls eine positive variable Größe. Die Durchschnittsaktivität $\bar{a}(t)$ wird dabei folgendermaßen berechnet:

$$\bar{a}(t) = \frac{\gamma}{n_{ACT}} \sum_{x,y} a_{xy} \quad \text{mit } 0 < \gamma \leq 1$$

Diese Berechnung liefert nicht den Mittelwert aller Aktivitäten im AM, sondern es werden nur solche Punkte miteinbezogen, deren Aktivität größer null ist. Dies wird durch den Wert n_{ACT} erreicht, welcher die Anzahl der aktiven Einheiten des AM darstellt. Zudem ist ein weiterer Gewichtungsfaktor γ in der Gleichung enthalten, von Mozer als „*depreciation factor*“ bezeichnet. Mit diesem kann man die Konkurrenzregel etwas unschärfer gestalten: Ist $\gamma = 1$, so muss ein Punkt eine Aktivität größer als der Durchschnitt haben, um nicht von der Regel unterdrückt zu werden. Ist jedoch $\gamma < 1$, so wird der berechnete Wert für die Durchschnittsaktivität herabgesetzt, sodass auch Punkte mit einer Aktivität leicht unter der tatsächlichen durchschnittlichen Aktivität von der Konkurrenzregel nicht beeinträchtigt werden.

In Abbildung 3.3 ist ein beispielhafter Verlauf der Arbeit des AM zu sehen. Zu Beginn (Iteration 0) besitzt das AM noch keinerlei Aktivität,

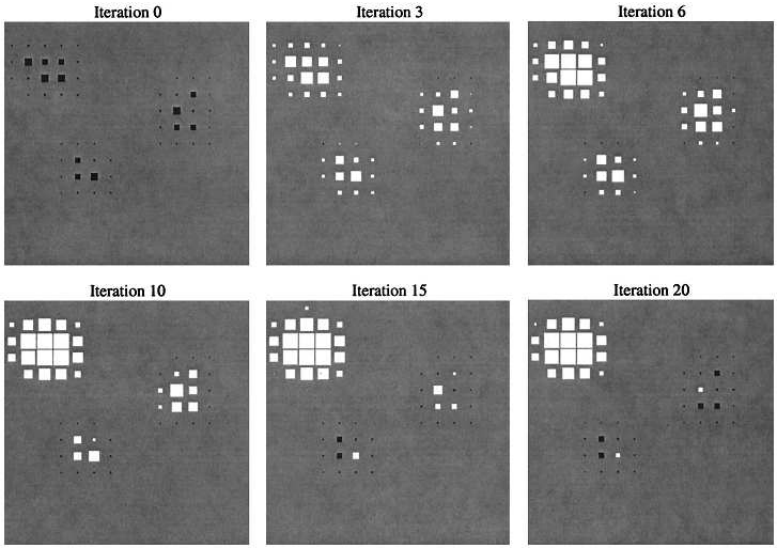


Abbildung 3.3: Beispiel für die Arbeitsweise des AM. Aus [12]

lediglich der exogene Input ist als schwarze Kästchen dargestellt. In den nachfolgenden Iterationen ist zu sehen, welchen Bildpunkten das AM eine Aktivität zuweist (weiße Kästchen, je größer desto höher die Aktivität). Zunächst werden alle Regionen selektiert, in denen sich Features befinden. Nach einer Weile jedoch wird nur die größte Region ausgewählt, während die kleineren unterdrückt werden, sodass zum Schluss nur noch ein Bereich aktiv ist, in welchem das System nun agieren würde.

In seinen Simulationen hat Mozer für die drei Parameter, welche die Updategleichung beeinflussen, festgelegte Werte verwendet. Die beiden Gewich-

tungsfaktoren wurden auf $\mu = 0.125$ und $\theta = 0.5$ gesetzt. Da der Parameter γ immer von der Menge der Aktivität, also der gefundenen Features abhängig ist, kann dieser nicht explizit festgelegt werden. Deshalb erstellte Mozer eine Formel zur Berechnung des Wertes, welche die Features berücksichtigt:

$$\gamma = min\left[1.00, max\left(0.75, \frac{exo_0}{\gamma'}\right)\right]$$

Der Zähler exo_0 des Bruches ist die Summe des exogenen Inputs. Der Parameter γ' im Nenner wird als Metaparameter bezeichnet und variiert je nach Simulation. Der Wert liegt nach Mozers Angaben in seinen Experimenten typischerweise zwischen 70 und 650.

3.3 Läsionierung des Systems

Für die Erkennung von Buchstaben ist das Modell von Mozer wie oben beschrieben ausreichend. Um jedoch das Verhalten von Patienten mit Hirnschädigungen zu simulieren, muss MORSEL ebenfalls in seiner Funktion beeinträchtigt werden. Ich werde mich in diesem Abschnitt weiterhin auf den Artikel [12] stützen, da Mozer in diesem genau den Fall der Funktionsstörung beschrieben hat, welcher ausschlaggebend für diese Arbeit ist.
Wie in Abschnitt 2.4 erläutert, zielte das Experiment von Hildebrandt et.al. aus [9] nicht auf eine offensichtliche Störung des visuellen Systems bei der Erkennung von Objekten, sondern auf die grundlegendere Funktion der menschlichen Aufmerksamkeit ab, welche verdeckt agiert und nur über Umwege analysiert werden kann. Um nun nachzuprüfen, ob die gewonnenen Thesen auch im Modell bestätigt werden können, muss eben der für die Aufmerksamkeit zuständige Teil aus MORSEL mit einer Störung modifiziert werden. Mozer hat dies bereits durchgeführt und mit dieser Änderung eine Auswahl von Experimenten simuliert (vgl. [12, S. 164]). Die Lokalisation und Art der künstlich erzeugten Störung basiert auf [13] und [14] und ist in Abbildung 3.4 schematisch dargestellt. Die Pfeile kennzeichnen die

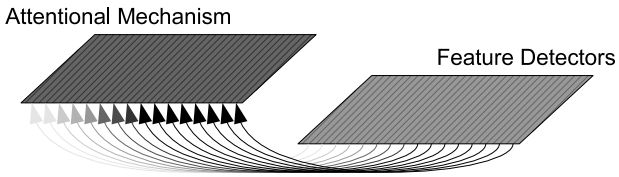


Abbildung 3.4: Läsion des AM in MORSEL. Adaptiert von [12]

Verbindungen der Feature Detektoren zum AM, also den exogenen Input.

Über diese Verbindungen wurde nun ein gradueller Defekt gelegt, angedeutet durch die Helligkeit der Pfeile. Dieser Defekt wird so interpretiert, dass den einzelnen Verbindungen ein Wert zugeordnet wird, welcher die Wahrscheinlichkeit angibt, mit welcher über die jeweilige Verbindung eine Information übertragen werden kann. So würde in dem abgebildeten Beispiel der linke Teil des AM nur sehr lückenhafte Informationen von den Feature Detektoren erhalten, da die Verbindungen dort eine geringe Übertragungswahrscheinlichkeit haben. Nach rechts wird diese Wahrscheinlichkeit immer größer, sodass ein Gradient entsteht. Mozer hat diesen Gradienten mathematisch als abschnittsweise lineare Funktion modelliert (veranschaulicht in Abbildung 3.5). Der Graph der Funktion besteht aus drei linearen Abschnit-

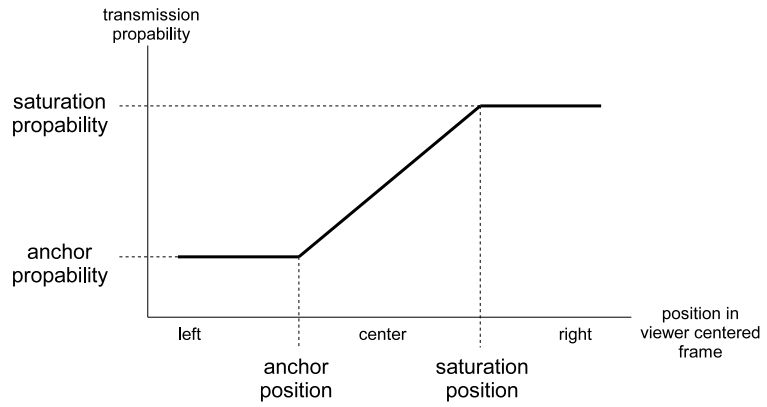


Abbildung 3.5: Modellierung des Gradienten. Adaptiert von [12]

ten, welche durch die Punkte *anchor* und *saturation* getrennt sind. Zwischen diesen beiden Punkten wird linear interpoliert, außerhalb von ihnen wird der Graph auf der y-Koordinate des Punktes weitergeführt. Dabei repräsentiert die x-Achse die horizontalen Position eines Features, und die y-Ache die Wahrscheinlichkeit, mit welcher die Information über dieses Feature übertragen wird. Diese Modellierung von Mozer erlaubt es, die Stärke und Form des graduellen Defektes genau einzustellen. So würden bei der Simulation eines nicht-läsionierten Systems die Wahrscheinlichkeiten jeweils nahe bei 1 liegen, sodass sich eine Gradientensteigung von 0 ergäbe. Ein stark defektes System würde sehr unterschiedliche Werte für die Wahrscheinlichkeiten von *anchor* und *saturation* erhalten, wodurch sich eine starke Gradientensteigung ergeben würde. Die Positionen der abschnittstrennenden Punkte können ebenfalls zur feineren Einstellung angepasst werden.

Kapitel 4

Anforderungen an JMorsel

In diesem Kapitel werden die Anforderungen an die zu erstellende Software beschrieben. In Anlehnung an das zu replizierende Modell werde ich das Produkt **JMorsel** nennen. Das J steht für die gewählte Programmiersprache Java, und obwohl nur ein Ausschnitt von MORSEL implementiert wird, habe ich den gesamten Titel vergeben, um Freiraum für nachfolgende Erweiterungen zu lassen.
Die Anforderungen an JMorsel ergeben sich zum großen Teil aus den vorherigen Kapiteln, da die Hauptaufgabe darin besteht, den aufmerksamkeitsbezogenen Teil aus MORSEL zu rekonstruieren. Im Folgenden werde ich die wesentlichen Punkte wiederholen, um einen Überblick zu schaffen, wobei ich mich auf die Terminologie der vorhergehenden Kapitel stütze, ohne bestimmte Begriffe speziell zu referenzieren. Die Punkte sind in Kategorien eingeteilt, welche eine Einordnung der jeweiligen Anforderungen erlauben.

4.1 Eingaben

JMorsel soll folgende Eingaben erhalten:

- **Testdaten**. Diese sind äquivalent zu den in Abschnitt 2.4 verwendeten Objektarrays und liegen in Form von Bilddateien im Bitmap-Format vor.
- **Parameter**. Die Funktionsweise der Software soll mit Hilfe von Parametern modifizierbar sein. So wird es eine Eingabemöglichkeit für die Parameter der Updategleichung geben, sowie Einstellmöglichkeiten für die Ausprägung der Läsion.
- **Settings**. Es sollte eine Möglichkeit geben, Einstellungen zu laden, sodass die Parameter nicht bei jedem Experiment neu eingegeben werden müssen.

4.2 Ausgaben

- **Simulationsergebnisse**. Es wird eine dateibasierte Ausgabe geben, welche die Daten im ASCII-Format ablegt. Die erzeugten Dateien sollen von herkömmlichen Tabellenkalkulationen sowie von statistischer/mathematischer Analysesoftware ohne größeren Aufwand lesbar sein. Um die Daten allerdings weiter auswerten und nutzen zu können, müssen diese strukturiert werden. Da die Experimente auf die Analyse der Aufmerksamkeit für einzelne Items abzielen, müssen die numerischen Ergebnisse der Simulation so konvertiert werden, dass jeweils ein Datum pro Item erzeugt wird.

- **Settings**. Dieser Punkt ergibt sich aus der Anforderung 'Settings' der Eingabe. Um Einstellungen nicht bei jedem neuen Experiment manuell festlegen zu müssen, sollte es die Möglichkeit geben, einmal festgelegte Parameter zu sichern, um sie später wieder laden zu können.

4.3 Funktionale Anforderungen

Die Eingabedaten sollen in JMorsel wie folgt verarbeitet werden:

- **Generierung von Features**. Da die Eingaben als Bilddaten vorliegen, muss zunächst eine Featureanalyse erfolgen, um die für die Simulation notwendigen Daten zu erzeugen. Dabei sollen die in Abschnitt 3.1 eingeführten Featuretypen verwendet werden.

- **Simulation der Aufmerksamkeit**. Dies ist die wesentliche Anforderung an das Programm. Die eingegebenen Bilder, bzw. generierten Features sollen vom System mit einer regional beschränkten Aufmerksamkeit versehen werden, um so eine Selektion von Objekten zu ermöglichen. Die Simulation erfolgt dabei schrittweise, was weitere Einstellmöglichkeiten wie maximale Schrittanzahl und Geschwindigkeit nach sich zieht.

- **Läsionierung**. Der Anwender soll die Möglichkeit erhalten, das System an der in Abschnitt 3.3 beschriebenen Stelle zu läsionieren, um die Auswirkungen eines Defektes zu simulieren.

- **Settings**. Als Ergänzung zum Laden und Sichern von Parametereinstellungen wäre es sinnvoll, automatisch die zuletzt verwendeten Werte zu laden, um die Durchführung von mehreren Experimenten unter gleichen/ähnlichen Konditionen weiter zu erleichtern. Zusätzlich sollte es möglich sein, die Standardwerte wiederherzustellen.

4.4 Nichtfunktionale Anforderungen

- **Ansprechende GUI**. Die Software muss eine übersichtliche und funktionale Benutzungsoberfläche bieten, welche dem Anwender einerseits die Kontrolle über die Ein- und Ausgaben, die Simulation, sowie die Einstellmöglichkeiten gibt, und ihm andererseits die Simulation in visuell ansprechender Weise präsentiert. So wäre eine Live-Ansicht der Simulationsschritte und der erzeugten Aufmerksamkeit denkbar, sowie die Visualisierung der eingestellten Läsion.

- **Einfache Installation**. Da die Informatik in diesem Projekt eher als Werkzeug dient, und die Software in einer reinen Anwenderumgebung laufen wird, sollte die Installation und das Ausführen der Anwendung nicht zu kompliziert sein.

Kapitel 5

Entwurf

In diesem Abschnitt soll der detaillierte Entwurf von JMorsel dargestellt werden. Ich werde zunächst auf grundlegende Fragen wie die zu verwendenden Werkzeuge und Methoden eingehen, bevor ich zum konkreten Design der Software übergehe.

5.1 Werkzeuge und Methoden

Aufgrund meiner persönlichen Erfahrungen und Kenntnisse habe ich mich in Bezug auf die Programmiersprache für Java entschieden [19]. Diese Sprache ist aktuell und befindet sich in stetiger Weiterentwicklung. Sie ist durch ihren objektorientierten, modularen Aufbau in Verbindung mit der sich durch ihre strukturelle Klarheit ergebenden Intuitivität eine leicht zu beherrschende, und dennoch sehr mächtige Sprache mit beliebiger Komplexität. In Bezug auf Desktopanwendungen hat Java jedoch den Nachteil, dass immer eine *Virtual Machine* zur Ausführung notwendig ist, was der Anforderung der möglichst leichten Handhabbarkeit aus Abschnitt 4.4 widerspricht. Dennoch ist meine Entscheidung auf Java gefallen, da es mehrere Möglichkeiten gibt, dieses Problem zu umgehen. Eine davon werde ich nun vorstellen.

5.1.1 Entwicklungsumgebung: Eclipse

Eines der bekanntesten Entwicklungswerkzeuge für Java ist die Eclipse IDE (siehe [20]). Diese ist frei verfügbar und besitzt einen großen Funktionsumfang. Ich werde JMorsel mit Hilfe von Eclipse 3.2.2 entwickeln. Um Datenverlusten vorzubeugen, werde ich die Versionsverwaltung CVS nutzen (zur Beschreibung des Systems siehe z.B. [1]). Das entsprechende *Repository* ist auf einem Server der Carl von Ossietzky Universität Oldenburg eingerichtet und kann durch die in Eclipse integrierte Funktionalität direkt aus der Entwicklungsumgebung heraus verwendet werden.

5.1.2 Framework: Eclipse RCP

Ein Unterprojekt der Eclipse IDE ist das sogenannte *Eclipse Rich Client Platform* (RCP, siehe [22]). Dieses Framework ermöglicht es Entwicklern, ihre Anwendungen als *Rich Client* zu erstellen. Ein Rich Client bezeichnet einen Client, der sowohl die grafische Repräsentation, als auch die Verarbeitung der Daten lokal vornimmt. Er besteht meist aus einer Sammlung von Plugins, welche seine eigentlichen Funktionalitäten definieren. Das Eclipse RCP stellt eine Fülle von vorgefertigten Modulen zur freien Verwendung bereit, hier einige Beispiele (zur näheren Erläuterung siehe [5]):

- Dialoge
- Wizards
- Validierung von Eingaben
- Toolbars
- Internationalisierung
- Registries
- Progress monitoring
- GUI Builder
- Hilfesystem

Das Framework basiert auf der OSGi Plattform, welche eine Spezifikation für eine Service-orientierte Architektur darstellt (siehe [16]). Diese Spezifikation wurde durch das Unterprojekt *Eclipse Equinox* [21] umgesetzt und stellt die grundlegende Basis für die Eclipse Richt Client Architektur dar. Unter anderem bietet Equinox die Möglichkeit eines nativen Produktexports, sodass es möglich ist, eine mit dem Eclipse RCP erstellte Anwendung für bestimmte Betriebssysteme auszuliefern. Die notwendige *Java Runtime Environment* (JRE) lässt sich dabei in das Produkt integrieren, sodass seitens des Anwenders keine weitere Installationen oder Anpassungen nötig sind.
Die grafischen Elemente (*Widgets*), welche die GUI ausmachen, entstammen dem *Standard Widget Toolkit* (SWT). Diese Bibliothek existiert seit 2001 und ist eine Hybridlösung zwischen schwer- und leichtgewichtig (vgl. [24]). Es setzt auf einem kleinen nativen Kern auf, welcher in der Lage ist, die auf den Betriebssystem vorhandenen Widgets nativ zeichnen zu lassen. Auf der Java Seite werden für grafische Komponenten möglichst kleine *Wrapper* (Hüllenklasse) eingesetzt, welche auf den nativen Kern zugreifen. Dieser ist für das jeweilige Betriebssystem unterschiedlich, auf dem die Anwendung

läuft. Er ist für die meisten populären Systeme vorhanden, darunter Windows, Linux, BSD, Solaris, MacOS, QNX sowie diverse mobile Plattformen wie Windows CE etc. (vgl. [23]). Ist ein benötigtes Widget auf der Zielplattform nicht vorhanden, so wird es vom SWT gerendert, was den leichtgewichtigen Teil darstellt.

Die Eclipse IDE ist selbst ein Rich Client, welcher mit dem Eclipse RCP erstellt wurde, sodass sie ein gutes Beispiel für die Möglichkeiten dieses Frameworks darstellt. In Abbildung 5.1 ist schematisch ihre Struktur dargestellt. Die unterste Ebene besteht einerseits aus Equinox (der OSGi Platt-

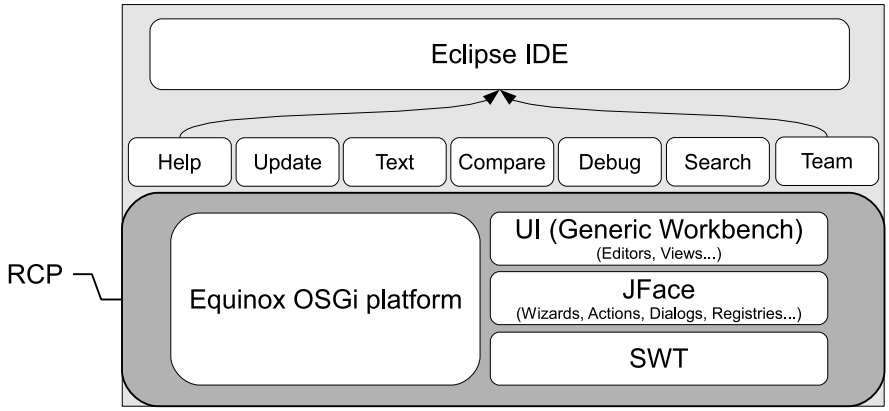


Abbildung 5.1: Der strukturelle Aufbau der Eclipse IDE. Adaptiert von [5]

form) und parallel aus einer Vereinigung von drei grafischen Paketen: Zunächst die Basisbibliothek SWT, welche die grundlegenden visuellen Elemente bereitstellt. Darauf setzt dann das JFace auf, welches SWT-Komponenten nutzt, um Elemente wie Wizards und Dialoge, aber auch abstraktere Funktionalitäten wie Actions und Registries anzubieten (eine *registry* ist im JFace ein Datenpool, welcher betriebssystemlastige Ressourcen wie Bilder und Farbobjekte verwaltet). Dieses wiederum wird vom Paket UI verwendet, um den *Workbench* zu erstellen, welcher aus Editoren, Views etc. besteht (vgl. [5]). Diese beiden Komponenten der untersten Ebene liefern somit den nötigen Kern, um die Eclipse IDE aufzubauen: In Equinox werden nun einzelne Bundles gestartet, welche jeweils eine Funktion der IDE darstellen und intensiv Gebrauch von den grafischen Paketen machen, dazu zählen die dargestellten Komponenten *Help*, *Update*, *Text* usw. In der Abbildung 5.2 ist dieser Zusammenhang noch einmal am Bild einer Eclipse IDE dargestellt.

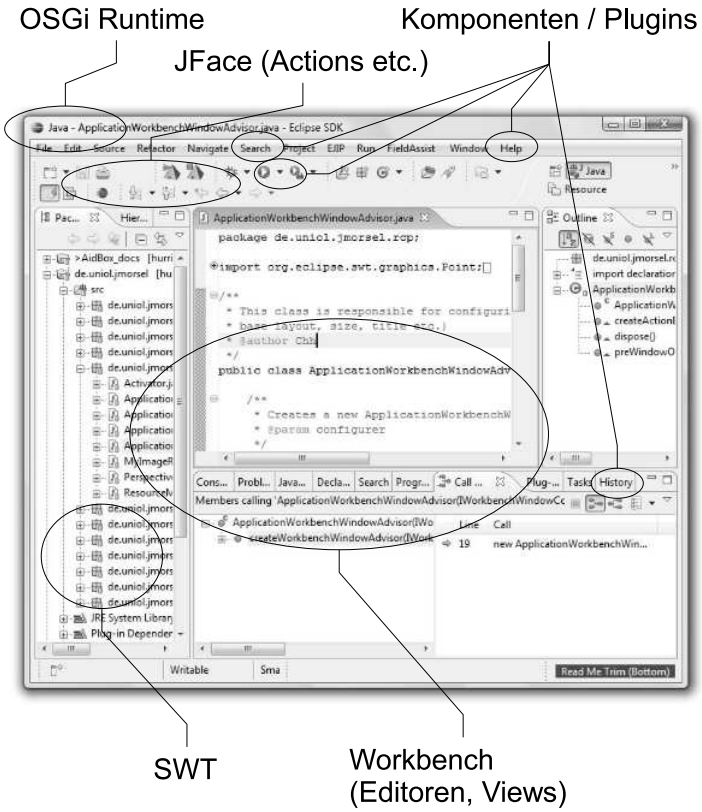


Abbildung 5.2: Die Eclipse IDE als Rich Client

Das Eclipse RCP Framework bietet dem Entwickler ein mächtiges Werkzeug zur Erstellung von Rich Client Architekturen. Man kann als Entwickler direkt die Sourcen der Eclipse IDE verwenden, um einen Rich Client zu erstellen. Minimal muss ein solcher Client aus den beiden in der Abbildung 5.1 mit „RCP“ markierten Paketen `org.eclise.core.runtime` (linker Teil) und `org.eclipse.ui` (rechter Teil) bestehen. Wahlweise können dann beliebige andere Bundles hinzugefügt werden, unter anderem alle in der Eclipse IDE enthaltenen Komponenten. Einige Vorteile der Eclipse RCP sind:

- Sehr schnelle Entwicklung einer professionell aussehenden Applikation
- Natives Look & Feel
- Kompilierung für diverse Plattformen: Windows, Linux, BSD, Solaris, MacOS, QNX...

- Konfiguration über XML-Dateien
- Workflow durch Implementierung von Actions

Das Framework gestaltet die Arbeitsfläche des Rich Clients prinzipiell nach einem Containerprinzip. Das bedeutet, dass es verschiedene Klassen von Containern gibt, welche visuelle Elemente enthalten können. Diese Container besitzen dann abhängig von ihrer Art Features, ohne dass der Entwickler diese selbst implementieren muss. Beispielsweise gibt es den Container *Editor*, welcher die Eigenschaft hat, eine Datenquelle wie etwa eine Datei als Basis zu besitzen und auf dieser zu arbeiten. So ist z.B. der mittlere Container in Abbildung 5.2, welcher einen Java-Quelltext anzeigt, vom Typ Editor, da er eine .java Datei darstellt und bearbeitet. Daneben gibt es *Views*. Diese Container sind abhängig von anderen Gegebenheiten der Anwendung. So ist etwa die *Outline* (in der Abbildung rechts) ein View, welcher sich immer auf den gerade aktiven Editor bezieht und die Struktur der geöffneten Java Datei anzeigt.
Die Besonderheit dieses Containerprinzips ist die dahinter liegende Grundfunktionalität. So lassen sich die einzelnen Views, Editoren etc. durch den Benutzer beliebig durch Drag'n'Drop in der Arbeitsfläche anordnen, vergrößern, deaktivieren etc. Der Entwickler muss diese Funktionen einfach nur in der entsprechenden Konfigurationsdatei seiner Anwendung aktivieren.
Die Eclipse IDE unterstützt den Entwickler bei der Erstellung eines RCP Projektes durch Wizards, welche die grundlegende Code-Struktur selbst generieren. Diese Struktur wird dann vom Entwickler mit Inhalt gefüllt und kann bei Bedarf erweitert werden. Für die Konfiguration der XML Dateien stellt Eclipse Hilfsmittel in Form von Editoren zur Verfügung, wodurch der XML-Code auf Wunsch des Entwicklers auch generiert werden kann. Dazu zählen Entwicklungsschritte wie das Einbinden neuer Views und Editoren sowie erweiterte Einstellungen wie Internationalisierung oder Branding des Produktes durch Namen, Versionsnummern, About-Informationen, Icons und ähnlichem.
Üblicherweise besteht ein Eclipse RCP Projekt aus folgenden Klassen:

- **Activator**, abgeleitet von *AbstractUIPlugin*: Diese Klasse definiert das eigentliche Bundle bzw. Plugin, welches man entwickelt. Sie ist für den gesamten Lebenszyklus des Plugins sowie Bundle-spezifische Dinge wie Preferences oder die Image Registry verantwortlich.
- **Application** implementiert *IPlatformRunnable*: Die Application Klasse stellt den Einstiegspunkt dar. Sie besitzt eine run()-Methode, welche zum Start der Anwendung vom OSGi Starter aufgerufen wird und das *Display* sowie den Workbench initialisiert. Die Klasse *Display* repräsentiert die Betriebssystemressource, auf der gezeichnet werden soll, also üblicherweise den Bildschirm.

- **ApplicationActionBarAdvisor**: Dieser Advisor erstellt und konfiguriert die ActionBar der Anwendung, also den Bereich, welcher die Menüs sowie die *Coolbar* enthalten soll. Die *Coolbar* ist eine Sammlung von Toolbars mit direkt zugänglichen Menükomponenten in Form von Buttons.

- **ApplicationWorkbenchAdvisor**: Der Workbench Advisor konfiguriert den Arbeitsbereich, indem Methoden zu speziellen Zeitpunkten des Anwendungs-Lebenszyklus aufgerufen werden (z.B. preWindowOpen() - wird direkt vor dem Anzeigen des Fensters ausgeführt) sowie der zu benutzende Window Advisor und die initiale Perspektive festgelegt wird.

- **ApplicationWorkbenchWindowAdvisor**: Der Window Advisor definiert das Layout des Fensters der Anwendung. Dazu zählen Größe, Anzeige der Statuszeile etc.

- **Perspective**: Eine Perspektive legt fest, welche Views und Editoren vom System angezeigt werden, sowie in welcher Größe und Position dies geschehen soll.

Daneben gibt es dann Editoren, welche wie bereits oben beschrieben auf externen Daten arbeiten, und Views, welche alle anderen Arten der visuellen Repräsentation vornehmen, sowie Actions: diese sind ein wichtiger Bestandteil des RCP Designs, denn sie stellen den Arbeitsfluss der Anwendung dar. Jedem Menüpunkt und Button sollte eine Action zugeordnet sein, in welcher dann die Ereignisbehandlung zu diesem Element implementiert wird.
Das Framework bietet auch hier dem Benutzer viele fertige Klassen zur Benutzung an. So gibt es Editoren für bestimmte Arten von Dateien, Views für Standardaufgaben wie etwa die Anzeige einer Baumstruktur und eine Fülle von standardisierten Actions, wie etwa *Save* und *Close*, welche sich auf einen Editor beziehen, oder *About* und *Help* zur Anzeige von erweiterten Informationen.

5.2 Design der GUI

Da ein Großteil der Anforderungen direkt vom Anwender über die GUI benutzbar sein sollen ist es sinnvoll, zunächst das grobe Design der Benutzungsoberfläche zu entwerfen. Dies erleichtert zum einen die Abdeckung der Anforderungen, und zum anderen die Strukturierung des Programmes auf Code-Ebene, da separate Bereiche in der GUI im Allgemeinen auch im Programmcode unter Beachtung der Abhängigkeiten separat implementiert werden können. Die im Programm verwendete Sprache wird Englisch sein. Durch den Internationalisierungsmechanismus des RCP ist es aber prinzipiell möglich, ohne zu großen Aufwand andere Sprachen einzufügen.

5.2.1 Hauptansicht

In Abbildung 5.3 ist skizzenhaft die Hauptansicht von JMorsel dargestellt. Im oberen Teil der Grafik sind zunächst die Titelzeile des Programms, die

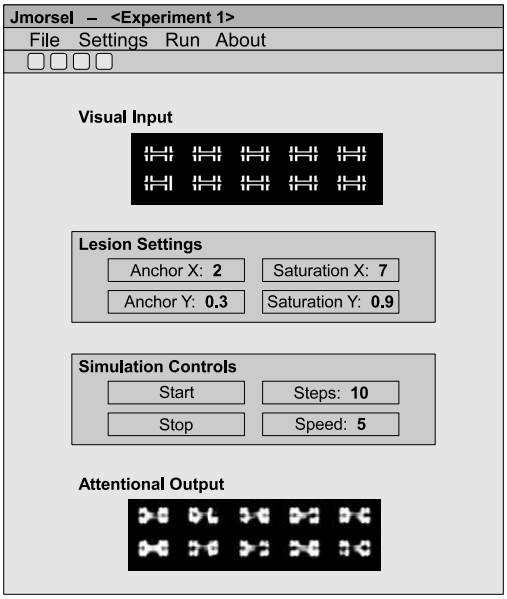


Abbildung 5.3: Aufbau der Hauptansicht der GUI

Menüleiste sowie eine Toolbar abgebildet. Alle Funktionen von JMorsel werden in den Menüs der Menüleiste zu finden sein. Ausgewählte Aktionen lassen sich aber auch über einen entsprechenden Button in der Toolbar ausführen. Darunter befindet sich der *Editor* (vgl. Abschnitt 5.1.2), welcher die Visualisierung der Simulation enthält. Dazu gehören folgende Elemente:

- **Visual Input**
  An dieser Stelle ist die Bilddatei dargestellt, welche in der Simulation verwendet werden soll.
- **Lesion Settings**
  Hier können die Parameter der Läsion definiert werden: Position sowie Übertragungswahrscheinlichkeit von *Anchor* und *Saturation* (siehe Abschnitt 3.2).
- **Simulation Controls**
  Dieser Bereich enthält Elemente zum Starten, Stoppen und Konfigurieren der Simulation. Zur Konfiguration gehören die Anzahl der zu simulierenden Schritte, sowie die Geschwindigkeit der Iterationen.

- **Attentional Output**
  Um die Arbeitsweise der Simulation direkt intuitiv erfassen zu können, werden hier zu jedem Iterationsschritt die aktuellen Aufmerksamkeitswerte gezeichnet.

5.2.2 View: Konfiguration der Updategleichung

Zusätzlich wird es einen *View* geben, welcher die Konfiguration der Parameter der Updategleichung erlaubt. Dieser ist exemplarisch in Abbildung 5.4 zu sehen.

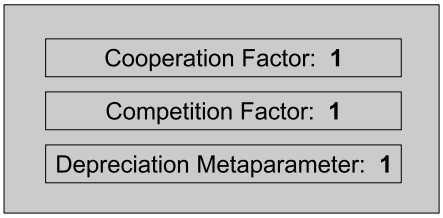


Abbildung 5.4: View: Konfiguration der Updategleichung

5.2.3 Menüpunkte

Die Menüs der Anwendung werden folgende Aktionen enthalten:

- **File, Load**
  Dieser Punkt dient dazu, eine Bilddatei zu laden und damit ein neues Experiment zu starten.
- **File, Export**
  Ermöglicht den Export der Simulationsergebnisse als numerische Werte in eine Datei. Diese Aktion wird vermutlich als *Wizard* realisiert werden, da zum Export erweiterte Einstellungen wie die Auswahl der Datei und des Dateiformates gehören. Zudem muss für die Bereitstellung der Daten das Simulationsergebnis in einzelne Zellen unterteilt werden, da ja nicht der numerische Wert jedes einzelnen Bildpunktes, sondern die summierten Werte pro dargestelltem Item von Interesse sind (vgl. Anforderung „Simulationsergebnisse“ in Abschnitt 4.2). Da dies jedoch von der Form der Eingabedaten abhängt kann dieser Prozess nur zum Teil automatisiert werden und Bedarf einer Kontrolle durch den Benutzer.
- **File, Exit**
  Beendet die Anwendung.

- **Settings, Load Settings**
  Lädt eine Datei, welche Einstellungen für das Programm enthält.
- **Settings, Save Settings**
  Speichert die aktuellen Einstellungen des Programmes in eine Datei.
- **Settings, Revert Settings**
  Setzt alle Einstellungen auf den Auslieferungszustand zurück.
- **Settings, Show Equation View**
  Zeigt den View zur Konfiguration der Updategleichung an.
- **Run, Detect Features**
  Führt die Feature-Erkennung durch, welche für die Simulation notwendig ist.
- **Run, Perform Simulation**
  Startet die Simulation. Dieser Punkt ist äquivalent zum „Start“-Button im Editor.
- **About**
  Zeigt Informationen über die Software, inklusive des Zustandes des OSGi-Systems und der Java-Umgebung an.

5.3 Softwaredesign

Der Aufbau von JMorsel richtet sich zu großen Teilen nach der vom RCP Framework vorgegebenen Struktur, da ein großer Prozentsatz der zu erstellenden Klassen für grafische Repräsentation zuständig sein wird.

5.3.1 Paketstruktur

Zunächst folgt eine Aufzählung der Pakete des Quelltextes. Die Paketnamen beginnen alle mit dem Präfix „`de.uniol.`“, welches die Zugehörigkeit der Anwendung zur Carl von Ossietzky Universität Oldenburg kennzeichnen soll.

- `de.uniol.jmorsel`
  Dieses Basispaket enthält jene Klassen, welche von anderen Klassen und Paketen intensiv genutzt werden und zentraler Bestandteil der Software sind.
- `de.uniol.jmorsel.connectors`
  Hier sind die sogenannten Konnektoren zu finden, welche Datenquellen mit Datensenken verbinden und Modifikatoren für numerische Daten darstellen. Sie stellen die Teile der Software dar, welche durch dynamische Verwendung zur Laufzeit eine Läsionierung des Systems ermöglichen.

- `de.uniol.jmorsel.morsel`
  In diesem Paket und seinen Unterpaketen sind die Klassen zu finden, welche die nachgeahmten Funktionalitäten aus dem Modell MORSEL darstellen.

- `de.uniol.jmorsel.morsel.am`
  Enthält die Implementierung des AM sowie eventuelle Erweiterungen dazu.

- `de.uniol.jmorsel.morsel.featuredetector`
  Da die einzulesenden Testdaten aus Bilddateien bestehen, müssen aus diesen zunächst visuelle Features extrahiert werden, welche dann in der Simulation verwendet werden. Dieses Paket enthält die für die Extraktion zuständigen Klassen.

- `de.uniol.jmorsel.rcp`
  Das Paket `rcp` enthält die in Abschnitt 5.1.2 vorgestellten Basisklassen einer RCP-Anwendung und ist generell mit seinen Unterpaketen für die visuelle Repräsentation zuständig.

- `de.uniol.jmorsel.rcp.actions`
  Hier befinden sich für die einzelnen Arbeitsschritte in der Anwendung die zugehörigen *Action*-Klassen.

- `de.uniol.jmorsel.rcp.dialogs`
  In JMorsel werden einige Einstellungen einen eigenen Dialog erfordern. Diese Dialoge werden hier zu finden sein.

- `de.uniol.jmorsel.rcp.editors`
  JMorsel wird ausschließlich Dateien des Typs `Bitmap` öffnen, sodass auch nur ein Editor erforderlich wird. Dieser ist aufgrund der Natur der Software jedoch sehr komplex und wird aus diversen Klassen zusammengesetzt sein, welche im Paket `editors` liegen.

- `de.uniol.jmorsel.rcp.jobs`
  Aktionen, welche einen komplexeren Prozess beinhalten, werden als *Job* entworfen. Ein Job meldet sich beim RCP Framework an und wird von diesem verwaltet. So lassen sich zum Beispiel abhängige Aufgaben koordinieren. Zudem stellt das RCP an dieser Stelle eine visuelle Rückmeldung der Aufgabe in Form von Fortschrittsbalken zur Verfügung.

- `de.uniol.jmorsel.rcp.views`
  Container, welche nicht unmittelbar dem Editor angehören und optional angezeigt werden können, fallen in die Kategorie *View*. Dieses Paket wird z.B. einen View zur Modifikation der Parameter der Updategleichung enthalten (siehe Abschnitt 5.2.2).

- `de.uniol.jmorsel.rcp.wizards`
  Ein Wizard dient dazu, den Benutzer schrittweise bei der Durchführung einer Aufgabe zu helfen. In JMorsel wird es einen Wizard geben, welcher die Exportfunktion der Simulationsergebnisse realisiert.

- `de.uniol.jmorsel.rcp.util`
  Dieses Paket enthält Helferklassen, welche keiner anderen Kategorie zugeordnet werden können und beispielsweise zur Berechnung von numerischen Daten oder zur Manipulation von Datensammlungen verwendet werden.

5.3.2 Beschreibung ausgewählter Klassen

An dieser Stelle sollen die grundlegenden Klassen beschrieben werden, welche die funktionale Struktur von JMorsel ausmachen. Die Auflistung ist nach Paketen unterteilt.

`de.uniol.jmorsel`

- `Experiment`
  Wird in JMorsel eine Bilddatei eingelesen, so startet damit ein neues Experiment. Die Klasse `Experiment` verkörpert dieses mit seinen zugehörigen Bilddaten, Simulationsergebnissen, allen getätigten Einstellungen sowie Metadaten wie Name oder Erstellungsdatum. Zur Laufzeit hält ein Objekt dieser Klasse Referenzen auf alle für dieses Experiment notwendigen Daten, und kann damit als zentrale Datenhaltung angesehen werden. Da sie keine statischen Methoden enthält, können durch erneute Instanziierung auch mehrere Experimente nebeneinander existieren.

- `IExperimentListener`
  Dieses Interface definiert einen *Listener*, der auf Ereignisse hört, welche das Experiment betreffen. Implementierende Klassen müssen sich im `Experiment` registrieren um benachrichtigt zu werden. So wird etwa nach jedem Simulationsschritt ein Ereignis ausgelöst, welches den registrierten Klassen mitteilt, dass neue Daten vorliegen, damit beispielsweise die GUI neu gezeichnet wird.

- `Settings`
  In Objekten dieser Klasse sind alle Einstellungen gespeichert, welche ein Experiment betreffen. Zusätzlich bietet diese Klasse Möglichkeiten, um die Einstellungen zu sichern, zu laden, oder zurückzusetzen.

- `TransmissionPropability`
  Diese Klasse repräsentiert eine Übertragungsfunktion wie in Abschnitt 3.3 definiert.

`de.uniol.jmorsel.connectors`

- `AbstractConnector`
  Dies ist eine abstrakte Basisklasse für Konnektoren (vgl. Abschnitt 5.3.1). Sie definiert Eingabedaten und schreibt die Implementierung einer Ausgabeberechnung vor.

- `GradientConnector`
  Dieser Konnektor enthält eine Instanz der Klasse `TransmissionPropability` und realisiert damit eine wahrscheinlichkeitsbasierte Übertragungsfunktion.

- `StraightConnector`
  Stellt die nicht-läsionierte Übertragung dar (konstante Wahrscheinlichkeit über das ganze Feld).

`de.uniol.jmorsel.morsel`

- `RasterAnalyzer`
  Diese Klasse ermöglicht die Rasterisierung von numerischen Werten in Feldern. Wird für den Export benötigt, siehe Abschnitt 4.2.

`de.uniol.jmorsel.morsel.am`

- `AttentionalMechanism`
  Stellt das AM aus MORSEL dar und beinhaltet die Funktionalität für die Durchführung und Berechnung von Simulationsschritten.

`de.uniol.jmorsel.morsel.featuredetector`

- `FeatureDetector`
  Dieser Detektor ist für die Erkennung von Features in den Eingabedaten zuständig.

`de.uniol.jmorsel.rcp`

- `ResourceManager`
  In dieser Klasse werden alle zur Laufzeit erstellten Farbobjekte verwaltet. Dies dient der Schonung von Betriebssystemressourcen.

Die anderen in diesem Paket enthaltenen Klassen wurden bereits in Abschnitt 5.1.2 vorgestellt.

`de.uniol.jmorsel.rcp.actions`

- `AttentionalMechanismAction`
  Führt die Simulation durch.

- `ExportAction`
  Exportiert die Simulationsergebnisse.

- `FeatureDetectionAction`
  Nimmt die Featureerkennung vor.

- `LoadSettingsAction`
  Lädt Einstellungen aus einer Datei.

- `NewExperimentAction`
  Liest eine Bilddatei ein, erstellt ein neues Experiment und öffnet den entsprechenden Editor.

- `PartListenerAction`
  Diese Klasse ist eine abstrakte Basisklasse für Aktionen, welche abhängig vom aktuellen Zustand des *Workbench* (also der geöffneten Editoren und Views) sind. So ist die `AttentionalMechanismAction` beispielsweise hiervon abgeleitet, da sie nur ausgeführt werden kann, wenn ein geöffnetes Experiment existiert.

- `RevertSettingsAction`
  Setzt alle Einstellungen auf ihre Standardwerte zurück.

- `SaveSettingsAction`
  Speichert die aktuellen Einstellungen in eine Datei.

- `ShowEquationViewAction`
  Öffnet bzw. schließt den View zur Konfiguration der Updategleichung.

`de.uniol.jmorsel.rcp.editors`

- `ExperimentEditor`
  Dieser Editor stellt die Hauptansicht von JMorsel dar und enthält die in Abschnitt 5.2.1 vorgestellten GUI-Elemente. Diese Klasse ist komponentenweise durch die nachfolgend aufgeführten Klassen aufgebaut.

- `ExperimentEditorInput`
  Stellt die Datenquelle für den `ExperimentEditor` dar und beinhaltet damit den Pfad zur geöffneten Datei sowie das assoziierte `Experiment`-Objekt.

- `ExperimentEditorComposite`
  Dies ist der grafische Teil des Editors. Diese Klasse ist aus dem SWT abgeleitet und enthält die grafischen Elemente. Neben den einfachen Elementen wie der Datenquelle und der Anzeige der Simulationsergebnisse enthält sie folgende komplexe Komponenten:

  - `LesionComposite`
    Stellt grafische Elemente zur Konfiguration der Läsion des Systems dar.

  - `ControlComposite`
    Beinhaltet Steuerungselemente für die Simulation.

`de.uniol.jmorsel.rcp.jobs`

- `AttentionalMechanismJob`
  Wird von der *AttentionalMechanismAction* aufgerufen und kontrolliert die einzelnen Schritte der Simulation.

- `FeatureDetectionJob`
  Wird von der *FeatureDetectionAction* aufgerufen und kontrolliert die einzelnen Schritte der Featureerkennung.

`de.uniol.jmorsel.rcp.views`

- `EquationView`
  Dieser View dient der Konfiguration der Updategleichung und ist durch das folgende *Composite* realisiert.

- `EquationComposite`
  Diese Klasse ist aus dem SWT abgeleitet und liefert die grafischen Elemente zur Manipulation der Parameter der Updategleichung.

`de.uniol.jmorsel.rcp.wizards`

- `ExportWizard`
  Dieser Wizard leitet den Benutzer durch den Vorgang des Datenexports.

- `ExportPageRaster`
  Stellt eine Seite des ExportWizards dar und dient der Konfiguration der zu exportierenden Daten.

- `ExportPageRasterComposite`
  Diese Klasse ist aus dem SWT abgeleitet und beinhaltet die grafischen Komponenten, welche Konfiguration der zu exportierenden Daten ermöglichen.

Kapitel 6

Implementierung der Grundversion

Dieses Kapitel beschreibt die Implementation von JMorsel in der Grundversion, welche den Anforderungen aus Kapitel 4 gerecht wird. Im Wesentlichen kann die Implementierung dem Entwurf (Kapitel 5) nachvollzogen werden, ich möchte hier nur einige spezielle Klassen und Methoden ansprechen sowie die Formate der Ausgaben der Anwendung darstellen.

6.1 Basisklassen

Die grundlegenden Klassen, welche eine lauffähige Anwendung ermöglichen, wurden vom Eclipse RCP Framework generiert. Diese wurden bereits in Abschnitt 5.1.2 beschrieben und liegen im Paket `de.uniol.jmorsel.rcp`. Einige Methoden mussten jedoch abgeändert werden, um bereits definiertes Verhalten zu korrigieren. Ich habe beispielsweise im gleichen Paket eine Klasse `MyImageRegistry` erstellt, welche einen Datenpool für geladene Bilddateien darstellt und im Gegensatz zu ihrer Elternklasse öffentlichen Zugriff auf die enthaltenen Daten ermöglicht. Diese musste dann in die Klasse `Activator` durch Überschreiben von Methoden eingebunden werden, damit sie vom Framework anerkannt und verwendet wird. Zusätzlich habe ich in der Klasse `Activator` den Zugriff auf die `Preferences` gelockert, um der Klasse `de.uniol.jmorsel.Settings` die Speicherung von Einstellungen mittels des vom RCP Framework bereitgestellten Mechanismus zu erlauben. Dies ermöglicht es der Anwendung, ihre Einstellungen persistent zu speichern, ohne dass der Benutzer eigens eine Datei dafür anlegen muss.

6.2 Konfigurationsdateien

Die Einbindung von Editoren und Views in das RCP Framework geschieht in zwei Schritten. Der erste beinhaltet die Erstellung der zuständigen Klas-

```
<extension
      point="org.eclipse.ui.editors">
   <editor
         class="de.uniol.jmorsel.rcp.editors.ExperimentEditor"
         default="true"
         extensions="bmp"
         icon="icons/spreadsheet-7.ico"
         id="de.uniol.jmorsel.rcp.editors.ExperimentEditor"
         name="ExperimentEditor"/>
</extension>
<extension
      point="org.eclipse.ui.views">
   <view
         class="de.uniol.jmorsel.rcp.views.EquationView"
         icon="icons/TableView.ico"
         id="de.uniol.jmorsel.EquationView"
         name="Equation View"/>
</extension>
```

Abbildung 6.1: Ausschnitt aus `plugin.xml`

sen, welche bestimmte Interfaces implementieren müssen. Der zweite Schritt besteht darin, diese Klassen dem Framework bekannt zu machen, indem sie in die entsprechende Konfigurationsdatei eingetragen werden. Abbildung 6.1 zeigt einen Ausschnitt der Datei `plugin.xml`, welche der Konfiguration der erstellten Anwendung dient. Im ersten Teil des Listings werden die Eigenschaften der Klasse `ExperimentEditor` festgelegt. Dazu wird zunächst der vollständige Name der Klasse angegeben. Nachfolgend sind dann die Parameter aufgeführt, welche die Verwendung dieser Klasse konfigurieren. So ist etwa definiert, dass dieser Editor der Standardeditor sein soll. Er kann Dateien vom Typ `BMP` öffnen und erhält das angegebene Icon als Schaltflächen- und Titelsymbol. Zusätzlich muss eine eindeutige ID spezifiziert werden, welche den Editor identifiziert. Anhand dieser ID können nun im Programmcode Instanzen dieses Editor erstellt und angesprochen werden.
Der zweite Teil ist in ähnliche Teile gegliedert und bewirkt das Einbinden der Klasse `EquationView`. Die komplette `plugin.xml` ist im Anhang A.1 auf Seite 67 zu finden, für nähere Erläuterungen der enthaltenen Elemente siehe [5] und [22].

6.3 Die `.settings` Datei

Die Funktion zum Speichern der Einstellungen schreibt diese als Schlüssel-Werte-Paare in eine Datei, sodass sie bei Bedarf mit einem Texteditor angepasst werden kann. Leider sind diese Daten in der Regel unsortiert, da

das `Preferences` Modul des RCP keine Sortierung vorsieht. In Anhang B.1 ist ein sortiertes und mit Kommentaren versehenes Exemplar der Standardeinstellungen abgebildet, auf welches sich die folgenden Erläuterungen beziehen.
Das Dokument beginnt mit zwei Kommentarzeilen, welche Metainformationen über die Datei enthalten. Dazu zählen der Titel der Datei sowie das Erstellungsdatum. Die dritte Zeile des Beispieldokumentes enthält den Wert `internal.timestamp`. Dieser hat keine semantische Bedeutung, sondern stellt einen Workaround für ein Problem mit dem RCP-`Preferences` Modul dar, da dieses die zur Laufzeit vorhandenen Wertepaare grundsätzlich nur dann in eine Datei speichert, wenn mindestens einer der Werte seit dem letzten Speichern verändert wurde. Der `timestamp` wird von der Anwendung immer vor dem Speichern neu gesetzt und beinhaltet den Wert, welcher von der Systemuhr per `System.getCurrentTimeMillis()` zurückgegeben wird, sodass in jedem Fall das Schreiben in eine Datei gewährleistet ist.
Die nachfolgenden Zeilen sind Konstanten, auf welche sich einige der Einstellungen beziehen. Eine Änderung dieser Werte hat keinen Einfluss auf das Programm, ist aber aus Übersichtlichkeitsgründen nicht empfohlen.
Anschließend sind die Parameter des AM, bzw. der Updategleichung dargestellt. Diese entsprechen mit einer Ausnahme den Werten aus Abschnitt 3.2. Und zwar habe ich den Parameter `biasWeight` hinzugefügt, welcher eine Gewichtung des Einflusses des exogenen Inputs erlaubt.
Die folgenden Werte beziehen sich auf den Export der Simulationsergebnisse. Zunächst ist der Typ des Exports als numerischer Wert angegeben. Die Bedeutung der Wertes ergibt sich aus dem Namen der dazu gehörigen Konstante. Anschließend ist die Aufteilung des visuellen Feldes in Zeilen und Spalten definiert, damit jedem dargestellten Objekt eine eigene Zelle zugewiesen werden kann. Die letzten beiden Werte dieser Kategorie konfigurieren die Form der Export-Datei.
Der nächste Abschnitt beinhaltet Einstellungen für die Dimensionen des simulierten visuellen Feldes. Diese wurden bisher nicht erwähnt, da sie zu den Erweiterungen zählen, welche in Kapitel 7 eingeführt werden.
Der letzte Bereich der Beispiel-Datei definiert die Art der eingestellten Läsion sowie ihre Parameter. Auch hier sind einige noch nicht erwähnte Begriffe zu finden, welche ebenfalls in Kapitel 7 erläutert werden.

Das RCP-`Preferences` Modul hat leider eine weitere Einschränkung. Es speichert bestimmte Wertepaare nicht ab, wenn ihr Wert dem Standardwert für den jeweiligen Datentyp entspricht. Das bedeutet, dass normalerweise `boolean`-Werte wie `Settings.includeSettingsInExport=false` nicht in der Datei auftauchen, da `false` der Standardwert für den Datentyp `boolean` ist. Ich habe diese Werte aufgrund der Übersichtlichkeit manuell in die Beispieldatei eingefügt.

6.4 Der Export der Simulationsdaten

Wie bereits bei den Anforderungen der Ausgabe in Abschnitt 4.2 bzw. bei der Beschreibung der Klasse `ExportRaster` in Abschnitt 5.3.2 erläutert, müssen die Daten des visuellen Feldes zunächst mittels eines Rasters in einzelne Zellen aufgeteilt werden, damit sie sinnvoll exportiert und weiterverarbeitet werden können. Abbildung 6.2 zeigt die entsprechende Seite des `ExportWizard`, in der diese Rasterisierung vorgenommen wird. Im rechten

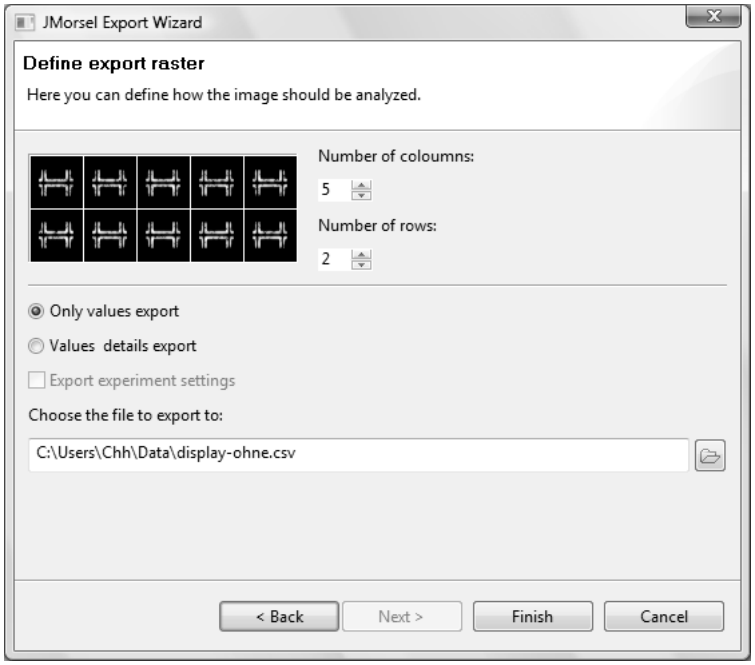


Abbildung 6.2: Die Rasterisierung des visuellen Feldes

oberen Teil befinden sich zwei sogenannte *Spinner*, mit denen sich die Anzahl der Zeilen und Spalten festlegen lassen, in die das visuelle Feld aufgeteilt werden soll. Bei dieser Modellierung wird davon ausgegangen, dass die einzelnen Objekte des visuellen Inputs homogen und symmetrisch über das Feld verteilt sind, sodass sie sich durch ein Rechteck-Raster in einzelne Zellen trennen lassen. Oben links ist das Feld mit eingezeichnetem Raster abgebildet. Es ist zu erkennen, dass es bei 2×5 Objekten mittels eines Rasters von zwei Zeilen und fünf Spalten möglich ist, das Feld so einzuteilen, dass jedes Objekt jeweils in einer Zelle liegt.
Im unteren Teil des abgebildeten Fensters lassen sich die Optionen für das Format der zu exportierenden Daten konfigurieren. Der Export kann in zwei Varianten erfolgen. Beide speichern die Daten im `CSV` Format, welches einzelne Felder durch Steuerzeichen voneinander trennt. Die übliche Schreibweise

ist, dass Felder durch ein Komma getrennt werden, Zahlen beinhalten dabei den Punkt als Dezimaltrennzeichen. Da dies jedoch bei den hiesigen Tabellenkalkulationen zu Problemen führt und der entsprechenden Anforderung aus 4.2 widersprechen würde, exportiert JMorsel die Daten mit dem Semikolon als Feldtrenner und dem Komma als Dezimaltrennzeichen.
Die beiden implementierten Varianten unterscheiden sich in ihrer Ausführlichkeit. Die lange Variante ist exemplarisch in Anhang B.2 dargestellt und wird durch den Button *Values details export* ausgewählt. Die ersten beiden Zeilen der Datei definieren jeweils nur ein Feld, dessen Inhalte durch Anführungzeichen als Text definiert sind. Sie enthalten Informationen bezüglich des Erstellungsdatums, der eingelesenen Bilddatei und der Einstellung des Export-Rasters (also der Zelleneinteilung des visuellen Feldes). Die dritte Zeile beschreibt die nachfolgenden Werte durch Spaltenüberschriften. Nach einer Leerzeile folgen dann blockweise die exportierten Simulationsergebnisse. Jeder Block enthält die Zellen einer Zeile des visuellen Feldes. Das erste Feld ist dabei aus ästhetischen Gründen leer gelassen. Das zweite enthält die Zellen-Nummer, das dritte die aufsummierten numerischen Werte der Zelle und das letzte deren relativen Anteil an der Summe aller Werte. Optional können am Ende noch die verwendeten Einstellungen des Experimentes an die Datei angehängt werden, dies wird durch den Button *Export experiment settings* aktiviert.
Die zweite Variante fällt wesentlich kürzer aus und ist für die direkte Weiterverarbeitung in Tabellenkalkulationen etc. vorgesehen. Sie wird durch den Button *Only values export* ausgewählt, eine Beispieldatei dazu ist in Anhang B.3 zu sehen. Sie enthält nur eine Zeile, und zwar die relativen Werte aller Zellen nebeneinander aufgelistet. Die Zellen sind dabei in der Reihenfolge Zeile → Spalte angeordnet, sodass im Beispiel die ersten fünf Werte die Zellen der ersten Spalte des visuellen Feldes darstellen (von links nach rechts), und die Werte sechs bis zehn die zweite Zeile (ebenfalls von links nach rechts). Das letzte Feld beinhaltet die Summe der absoluten Werte aller Zellen, sodass sich daraus die gleichen Informationen errechnen lassen, welche in der detaillierten Variante enthalten sind. Der Anwender muss sich lediglich merken, in wie viele Zeilen und Spalten er das visuelle Feld aufgeteilt hat.
Der unterste Teil der Abbildung gibt die Zieldatei des Exportes an. Mit dem auf der rechten Seite des Textfeldes angebrachten Button wird ein Dateiauswahldialog geöffnet. Als Voreinstellung ist im Textfeld eine Datei angegeben, welche im selben Verzeichnis wie die geöffnete Bilddatei liegt und den gleichen Namen trägt, jedoch mit der Endung `.csv`. Wird hier eine Datei eingetragen, welche bereits existiert, färbt sich der Hintergrund des Feldes rot, und es erscheint eine Warnung, welche den Anwender darauf hinweist, dass die vorhandene Datei überschrieben wird.
Mit Auswahl des Buttons *Finish* werden die Daten dann mit gewählten Konfiguration exportiert.

6.5 Die Klasse `NumericInput`

Die Eingabe der Parameter der Läsion sowie der Gewichte der Updategleichung sollte in JMorsel ursprünglich über die Klasse `Spinner` des Paketes `org.swt.widgets` erfolgen (siehe Abbildung 6.3). Leider ist dieses Element

Abbildung 6.3: Das Eingabeelement `Spinner` des SWT. Aus [23]

weder in der Lage, negative Zahlen zu akzeptieren, noch ein Symbol zur Anzeige der Einheit einzufügen (wie etwa das Prozentzeichen „%“ bei der Eingabe der horizontalen Läsionsparameter). Daher musste ein eigenes Eingabeelement implementiert werden, welches diese Funktionen unterstützt. Diese Klasse `de.uniol.jmorsel.rcp.editors.NumericInput` basiert auf dem SWT Eingabeelement `org.eclipse.swt.widgets.Text`. Zusätzlich wurden diverse Überprüfungen hinzugefügt, welche die Eingabemöglichkeiten auf Fließkommazahlen mit einem optional folgenden Einheitssymbol beschränken. Der Hauptalgorithmus dieser Überprüfungen ist in Abbildung 6.4 dargestellt. Die abgebildete Methode `verifyText(VerifyEvent e)` entstammt der Klasse `org.eclipse.swt.events.VerifyListener`, welche dem Element `Text` als Listener zugeordnet wird. Das übergebene Objekt `e` enthält das Zeichen, welches der Benutzer zuletzt eingegeben und die Ausführung der Methode verursacht hat. Dieses ist über `e.character` auslesbar. Der boole'sche Wert `e.doit` gibt an, ob die Modifikation durch das eingegebene Zeichen durchgeführt werden soll oder nicht. Mittels `e.start` und `e.end` lässt sich der Bereich der im Element `Text` enthaltenen Zeichenfolge abfragen, welcher durch die Eingabe des Benutzers modifiziert werden soll.
Die Methode überprüft nun sequentiell mögliche Fehlerquellen und deaktiviert die Durchführung der Modifikation, falls eine der Überprüfungen wahr wird.

Neben dieser Methode existieren in der Klasse `NumericInput` weitere Algorithmen, welche dafür sorgen, dass das enthaltene Element `Text` nur numerische Daten akzeptiert und auch wieder zurückgibt.

6.6 Fortschrittsbalken

JMorsel enthält zwei Prozesse, welche unter Umständen viel Zeit in Anspruch nehmen. Der eine ist die Erkennung von Features in den Eingabedaten, und der andere ist die Simulation der Aufmerksamkeit. Wie in Abschnitt 5.3.1 bereits erwähnt existiert im RCP Framework die Möglichkeit,

```
public void verifyText(VerifyEvent e) {
    char c = e.character;
    String cS = new String() + c;
    // check char range
    if (!Character.isDigit(e.character)
            && !Character.isISOControl(e.character) && c != ','
            && c != '.' && c != '-'
            && !NumericInput.this.unit.contains(cS) && c != '\0') {
        e.doit = false;
    // check duplicate floating point sign
    } else if ((c == ',' || c == '.')
            && (((Text) e.getSource()).getText().contains(",") ||
                ((Text) e.getSource()).getText().contains("."))) {
        e.doit = false;
    // check duplicate negative sign
    } else if(c == '-' && e.start != 0) {
            e.doit = false;
    // check modification in front of negative sign
    } else if (((Text) e.getSource()).getText().contains("-")
            && e.start == 0 && e.end == 0) {
        e.doit = false;
    // check value range
    } else if(e.start >= 0 && e.end >= 0 && c != '\0') {
        try {
            // build resulting string
            StringBuffer sb = new StringBuffer();
            String t = ((Text) e.getSource()).getText();
            sb.append(t.substring(0, e.start));
            sb.append(e.text);
            sb.append(t.substring(e.end));
            // check range of formatted string
            e.doit = checkRange(Float.parseFloat(checkFormat(sb
                    .toString())));
        } catch (NumberFormatException e1) {}
    }
}
```

Abbildung 6.4: Die Methode `verifyText(VerifyEvent e)`

solche komplexeren Prozesse durch den `Job`-Mechanismus zu realisieren, um dem Anwender eine visuelle Rückmeldung über den Verlauf des Prozesses zu geben. Diese Rückmeldung erfolgt normalerweise durch einen eingeblendeten Fortschrittsbalken. In Abbildung 6.5 ist der Dialog zu sehen, welcher den Fortschritt der Featureerkennung darstellt.

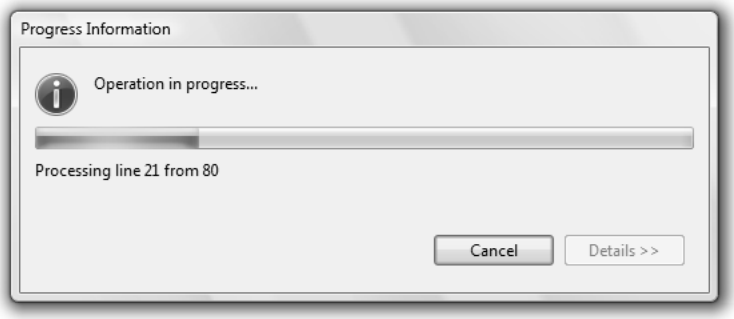


Abbildung 6.5: Fortschritt der Featureerkennung

Dieser Dialog wird jedoch erst nach einer kurzen Zeitspanne nach dem Start eines Jobs angezeigt. Dies liegt daran, dass die von diesem Mechanismus verwalteten Prozesse in ihrer Geschwindigkeit meist von der Hardware abhängig sind, auf der sie ausgeführt werden, und das RCP Framework nur für solche Prozesse eine visuelle Rückmeldung aufbaut, welche nicht „sofort“ im subjektiven Sinne fertiggestellt sind.
Die Visualisierung des Simulationsprozesses ist nicht als eigener Dialog realisiert, da dieser die Live-Ansicht der Simulationsergebnisse verdecken würde. Stattdessen wird am unteren Rand der Anwendung ein kleiner Fortschrittsbalken eingeblendet. Dies ist in Abbildung 6.6 dargestellt.

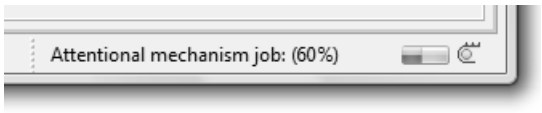


Abbildung 6.6: Fortschritt der Simulation

6.7 Featureerkennung

In Abschnitt 3.2 wurde die Arbeitsweise des AttentionalMechanism dahingehend erläutert, dass es als Eingabe die gefundenen Features der Featuredetektoren analysiert und auf dieser Basis die Verbindungen der Detektoren zum Pull-Out Netzwerk modifiziert. An dieser Stelle soll spezifiziert werden,

wie die Erkennung der Features in JMorsel realisiert ist. Die Features bestanden in MORSEL aus Liniensegmenten in bestimmten Orientierungen: 0° ('|'), 45° ('/'), 90° ('—') und 135° ('\'). Eine Möglichkeit, diese Features in einem gegebenen Bild zu finden, ist die Verwendung sogenannter gradientenbasierter Kantendetektoren, welche in der Lage sind, Sprünge in der Verteilung der Helligkeit eines Bildes zu erkennen. Ich werde an dieser Stelle lediglich den verwendeten Robinson-Operator vorstellen, eine ausführliche Schilderung der Grundlagen ist in [6] zu finden. Der Operator besteht aus je zwei Filtermatrizen für die verwendeten Featuretypen:

	—	\	/
$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$	$\begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & -2 \end{bmatrix}$
$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & -2 \\ 1 & 0 & -1 \\ 2 & 1 & 0 \end{bmatrix}$	$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}$

In der ersten Zeile ist der jeweilige Featuretyp angegeben, für den die darunter liegenden Matrizen gelten. Die Matrizen werden auch als *Kernel* bezeichnet und stellen Berechnungsvorschriften dar, welche jedem Pixel einen Wert zuweisen. Dieser Wert gibt die Güte der gefundenen Hell-Dunkel-Kante an. Anschaulich beschrieben wird ein Kernel „über das Bild gelegt", was als *Faltung* bezeichnet wird:

$$\underbrace{\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}}_{\text{Kernel}} \cdot \underbrace{\begin{bmatrix} p_{00} & p_{10} & p_{20} \\ p_{01} & p_{11} & p_{21} \\ p_{02} & p_{12} & p_{22} \\ p_{03} & p_{13} & p_{23} \end{bmatrix}}_{\text{Bilddaten}} = \underbrace{\begin{bmatrix} a \cdot p_{00} & b \cdot p_{10} & c \cdot p_{20} \\ d \cdot p_{01} & e \cdot p_{11} & f \cdot p_{21} \\ g \cdot p_{02} & h \cdot p_{12} & i \cdot p_{22} \end{bmatrix}}_{\text{Faltung}}$$

In obigem Beispiel wäre p_{11} das Zielpixel, für das der Wert der Faltung berechnet wird. Dieser Wert ergibt sich aus der Aufsummierung aller neun Produkte in der mit *Faltung* beschrifteten Matrix. Soll eine Faltung für ein Pixel durchgeführt werden, welches am Rand des Bildes liegt und somit keine acht Nachbarn besitzt, so wird für die fehlenden Pixel der Wert 0 angenommen.

Da in JMorsel allerdings keine Hell-Dunkel-Sprünge, sondern konkrete Kanten gefunden werden sollen, müssen die durch den Robinson-Operator ermittelten Daten weiter bearbeitet werden. In Abbildung 6.7 ist die Methode `calculateDetectionValues()` der Klasse `FeatureDetector` dargestellt, welche diese Berechnungen vornimmt. Die Methode ist in fünf Abschnitte unterteilt, welche jeweils durch eine Kommentarzeile kenntlich gemacht sind.

```
private void calculateDetectionValues() {
    if(isFeaturesDetected()) {
        int x = features.getDimension().x;
        int y = features.getDimension().y;
        float[][] ret = new float[y][x];

        // Sum all feature values
        for(int i = 0; i < FEATURES_TO_DETECT; i++) {
            for(int yy = 0; yy < y; yy++) {
                for(int xx = 0; xx < x; xx++) {
                    ret[yy][xx] += features.getDetectionValues(i)[yy][xx];
                }
            }
        }

        // Calculate arithmetic mean from all participants per pixel
        for(int yy = 0; yy < y; yy++) {
            for(int xx = 0; xx < x; xx++) {
                // Get amount of participants for this pixel
                int count = 0;
                for(int i = 0; i < FEATURES_TO_DETECT; i++) {
                    if(features.getDetectionValues(i)[yy][xx] > 0) {
                        count++;
                    }
                }
                // Divide by amount of participants
                if(count > 0) {
                    ret[yy][xx] /= count;
                }
            }
        }

        // Multiply with image to delete features outside of objects
        ret = multImageValues(ret);

        // Scale results to [0,0.2]
        ret = Util.scaleToZeroMaxValueInterval(ret, 0.2f);

        // Store result
        calculatedValues = ret;
    }
}
```

Abbildung 6.7: Die Methode `calculateDetectionValues()`

Im ersten Abschnitt (Zeilen 7-14) werden zunächst die berechneten Werte aller angewendeten Matrizen des Robinson-Operators pro Pixel aufsummiert, sodass sich ein Array ergibt, welches für jedes Pixel des Ausgangsbildes die kumulierten Werte der gefundenen Hell-Dunkel-Sprünge enthält. Der zweite Abschnitt (Zeilen 16-31) gewichtet diese Werte dann nach der Anzahl der Matrizen, welche für ein Pixel einen Wert $v > 0$ geliefert haben und damit an dem errechneten Wert des Pixels beteiligt waren. Abschnitt drei (Zeilen 33-34) multipliziert die Werte nun mit den tatsächlichen Farbwerten des Bildes, sodass nur an den Positionen Werte stehen bleiben, an denen sich im Ausgangsbild ein Objekt befindet. Dabei wird davon ausgegangen, dass der Hintergrund des Bildes Schwarz ist und die Pixel in dem Bereich den Farbwert 0 haben. Im folgenden Abschnitt (Zeilen 36-37) werden die bis hier errechneten Werte in das Intervall $[0, 0.2]$ skaliert, was den von Mozer in [12] verwendeten Grenzen entspricht. Der letzte Abschnitt (Zeilen 39-40) speichert das Ergebnis schließlich in die Klassenvariable `calculatedValues`.

In Abbildung 6.8 ist noch einmal der Unterschied zwischen dem Robinson-Operator und dem oben vorgestellten Algorithmus an einem Beispielbild dargestellt. Es ist leicht zu erkennen, dass das sich durch den Robinson-

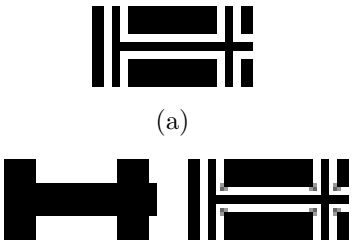


Abbildung 6.8: Featureerkennung des Bildes (a): nur Robinson-Operator (b), mit Nachbearbeitung (c)

Operator ergebende Bild für die vorliegende Anwendung unbrauchbar ist, da es die im Originalbild vorhandenen Kanten bzw. Features nur sehr grob und unzulänglich wiedergibt. Das nachbearbeitete Bild jedoch enthält die Kanteninformationen in der Form, dass ein Pixel umso heller ist, je eindeutiger ihm im Originalbild ein Feature des Objektes in Form einer Kante zugeordnet werden kann.

Kapitel 7

Erweiterungen zu MORSEL

Ein weiteres Ziel der vorliegenden Arbeit ist eine Erweiterung des Systems MORSEL durch weiterführende Funktionen. Die im Folgenden aufgeführten Punkte entstanden zum Teil aus Überlegungen theoretischer Natur, aber auch aus praktischen Gründen, welche sich bei der Evaluation der Grundversion ergaben.

7.1 Vereinfachte Konfiguration der Updategleichung

In Abschnitt 3.2 wurde die sog. Updategleichung eingeführt. Diese besteht aus diversen Einzeltermen, welche jeweils durch einen Faktor gewichtet in die Gleichung einfließen. JMorsel bietet die Möglichkeit der Konfiguration dieser Gewichtungsparameter, wie in Abschnitt 5.2.2 vorgestellt. Diese Form der Eingabe ist jedoch sehr unintuitiv, weshalb der entsprechende *View* durch visuelle Einbettung der Parameter in die Updategleichung erweitert wurde. Abbildung 7.1 zeigt die neue Ansicht des Views. Der obere Teil beinhaltet

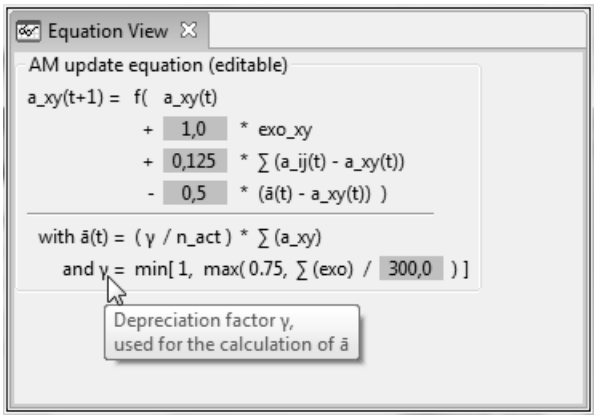


Abbildung 7.1: View: Konfiguration der Updategleichung (erweitert)

untereinander stehend die einzelnen Terme der Gleichung. Die einstellbaren Parameter sind als Eingabeelemente realisiert, welche positive und negative Dezimalzahlen akzeptieren. Der letzte Term der Gleichung enthält selbst wieder einen Parameter, welcher im unteren Teil des Views angegeben werden kann.
Alle Elemente dieser Eingabemaske sind mit *Tooltips* ausgestattet, welche eine Erläuterung der einzelnen Gewichte und Terme liefern. Dies ist in der Abbildung anhand des dargestellten Mauszeigers demonstriert.

7.2 Visualisierung der Läsion

Da die Eingabe von Läsionsparametern als reine Ziffernwerte sehr abstrakt und schwer vorzustellen ist (siehe Abbildung 7.2), wurde eine Visualisierung der Läsion implementiert. In Abbildung 7.3 ist ein Ausschnit aus der GUI mit eingelesenem Bild und darüber gezeichneter Läsion abgebildet. Die Läsionsparameter entsprechen der Standardeinstellung von JMorsel und sind den Experimenten von Mozer aus [12] entnommen. Die Grafik ist so zu interpretieren, dass die Y-Werte der Kurve die Übertragungswahrscheinlichkeiten der Läsion darstellen (siehe auch Abschnitt 3.3).

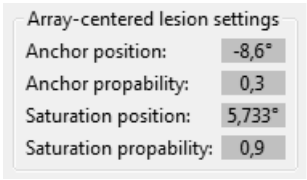


Abbildung 7.2: Eingabe der Läsionsparameter

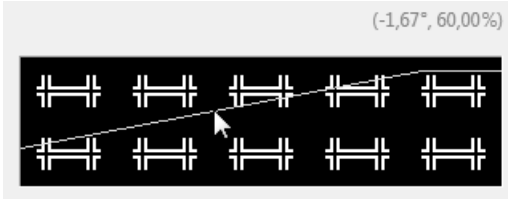


Abbildung 7.3: Visualisierung der Läsion und Anzeige der Koordinaten

7.3 Angabe der horizontalen Feldposition in Grad

Um die Experimente aus [9] besser nachvollziehen zu können, sollte die horizontale Feldposition nicht als absolute Pixelnummer oder prozentuale An-

gabe geschehen, sondern als Sehwinkelabweichung in Grad (°). Dabei wird die Mitte des visuellen Feldes als 0° definiert. Links davon wird die Abweichung in negativen, rechts davon in positiven Zahlen angegeben. Im oberen rechten Teil von Abbildung 7.3 ist in grauer Schrift die aktuelle Position des abgebildeten Mauszeigers zu sehen. Die Spitze des Zeigers befindet sich also auf -1,67° horizontaler Position. Die Gesamtspanne des visuellen Feldes im abgebildeten Beispiel ist 17,2°, womit der Zeiger am linken Rand des Feldes -8,6° und am rechten Rand 8,6° anzeigen würde. Diese Werte sind die in JMorsel integrierten Standardwerte und entsprechen den Zahlen aus [9]. Die vertikale Position des Zeigers ist als Prozentzahl angegeben (der untere Rand des Feldes entspricht 0%, der obere 100%). Diese prozentuale Angabe dient der Orientierung und kann zusammen mit dem gezeichneten Graphen der Läsion als Übertragungswahrscheinlichkeit interpretiert werden.
Da die Berechnung dieser Angaben jedoch von der simulierten Größe des Feldes sowie seinem Abstand zum imaginären Beobachter abhängt, musste zusätzlich eine Konfigurationsmöglichkeit geschaffen werden, um diese Randbedingungen festzulegen. Abbildung 7.4 zeigt den Dialog, welcher die Eingabe dieser Daten erlaubt. Der Anwender kann dabei wählen, ob er die

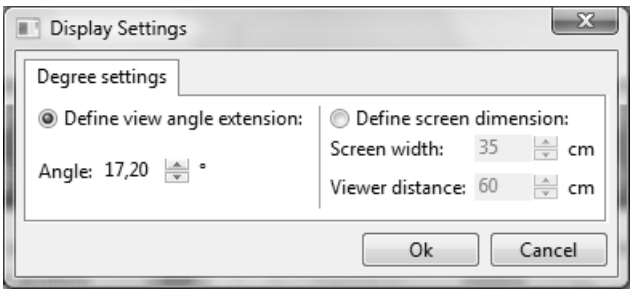


Abbildung 7.4: Einstellungen der Displaygröße

Gesamtspanne in Grad angibt, welche das visuelle Feld umfasst, oder ob er die Größe und den Abstand des Displays angibt. Im zweiten Fall berechnet JMorsel automatisch die Spanne, die das Feld umfasst.
Der Dialog befindet sich im Paket `de.uniol.jmorsel.rcp.dialogs` und besteht aus folgenden Klassen:

- `SettingsDialog`
  Ist von der Klasse `Dialog` des SWT abgeleitet und stellt ein konfigurierbares Dialogfenster dar.

- `SettingsComposite`
  Liefert den grafischen Inhalt des Dialoges, bestehend aus den Buttons zum Bestätigen und zum Abbrechen, sowie einem `TabFolder`, welches den Rahmen für verschiedene Registerkarten bereitstellt.

- `DegreeSettingsComposite`
  Ist in das `TabFolder` aus dem `SettingsComposite` eingebettet und realisiert eine Registerkarte zur Konfiguration der Parameter.

7.4 Neuer Läsionstyp: Rechteckkurve

Um die Läsion feingranularer definieren zu können, sollte der Gradient nicht wie bisher durch zwei, sondern durch vier Punkte definiert werden, sodass sich eine Art Rechteckkurve ergibt. In Abbildung 7.5 ist dieser Zusammenhang grafisch dargestellt und mit den in JMorsel verwendeten Bezeichnungen versehen. Die Gradientenläsion aus Abschnitt 3.3 wurde hier durch zwei

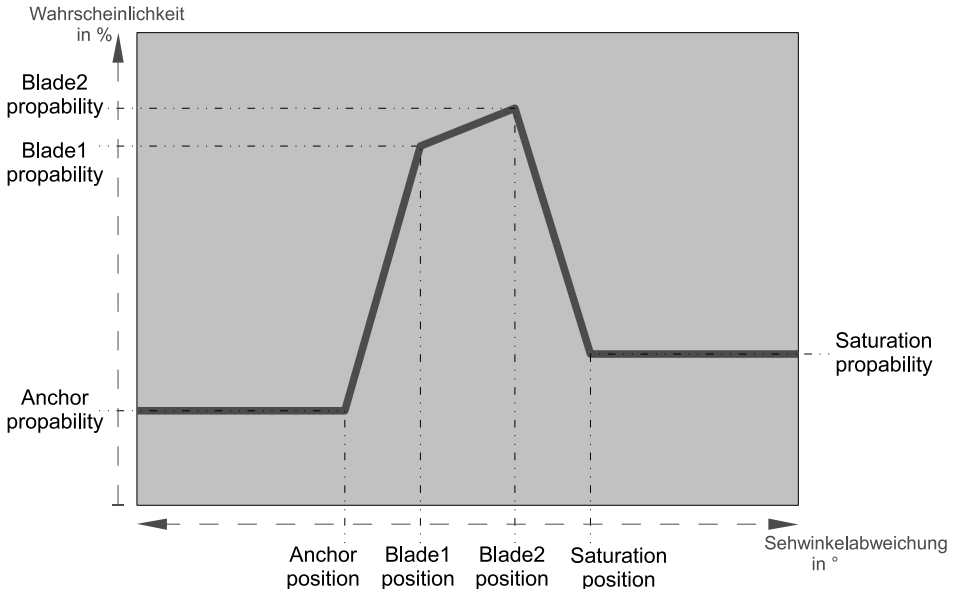


Abbildung 7.5: Läsionstyp: Rechteckkurve

Punkte im bisher linearen Abschnitt zwischen *anchor* und *saturation* erweitert. Diese sind mit *blade1* und *blade2* betitelt. Der sich ergebende Graph entspricht nicht genau einer Rechteckkurve, da der mittlere Schenkel nicht waagerecht ist, sowie die *anchor propability* und die *saturation propability* nicht auf einer Höhe stehen. Dies ist aber gewollt, da diese Kurvenform mehr Flexibilität bei der Läsionierung erlaubt.
Auf Code-Ebene wurde diese Läsion durch eine zusätzliche Konnektor-Klasse `RectangleConnector` im Paket `de.uniol.jmorsel.connectors` realisiert.

7.5 Zusätzliche objektzentrierte Läsion

Die von Mozer vorgestellte und bisher in JMorsel verwendete Läsion ist auf das gesamte visuelle Feld und damit auf rein beobachterzentrierte Koordinaten ausgelegt. Neben dem Versuch, die Ergebnisse des Experimentes von Hildebrandt et al. [9] mit dieser Läsion zu simulieren, könnte eine Modellierung nach objektzentrierten Gesichtspunkten Sinn ergeben, da sich die Fragestellung aus dem Experiment auf die Existenz einer solchen objektzentrierten Repräsentation bezog. Daher wurde in JMorsel zusätzlich die Möglichkeit geschaffen, eine Läsion auf Item-Ebene zu definieren. Der Typ der Läsion ist die im vorhergehenden Abschnitt 7.4 vorgestellte Rechteckkurve. In Abbildung 7.6 ist der Dialog zur Konfiguration dieser Läsion dargestellt. Der obere Bereich des Dialoges ist analog zu der Konfiguration der

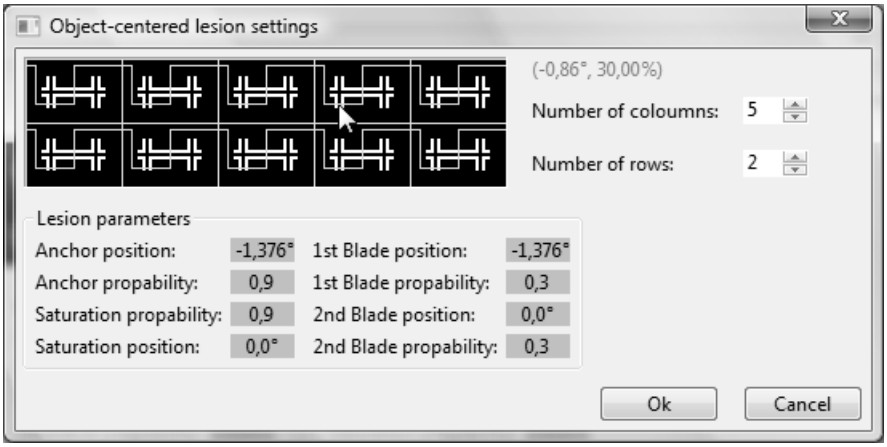


Abbildung 7.6: Konfiguration der objektzentrierten Läsion

Rasterisierung im `ExportWizard` aufgebaut (siehe Abschnitt 6.4). Im unteren Bereich befindet sich die Eingabe der Läsionsparameter. Diese sind im Vergleich zu Abbildung 7.2 um die in Abschnitt 7.4 eingeführten zusätzlichen Punkte *blade1* und *blade2* erweitert. In der darüberliegenden Grafik ist neben dem Raster auch die Läsion als Kurve eingezeichnet. Die durch die definierten Parameter entstehende Kurve erstreckt sich nicht global über das ganze Feld, sondern wird hier auf jede Zelle einzeln angewendet. Die Position des abgebildeten Mauszeigers verdeutlicht diese Funktion. Die Koordinaten des Zeigers sind rechts neben der Grafik angegeben und demonstrieren die Wirkungsweise des Rasters: der Zeiger befindet global gesehen etwa auf 60% der Höhe des Feldes, und horizontal gesehen auf der rechten Hälfte, was eine positive Sehwinkelabweichung als Koordinate ergeben müsste. Jedoch stehen die angegebenen Koordinaten bei $(-0{,}86°, 30{,}00\%)$, da sich die Koordinaten in diesem Dialog immer auf den Bereich einer einzelnen Zelle

beschränken. Die Spanne, welche das gesamte Feld umfasst, liegt bei 17,2° (vgl. Abschnitt 7.3). Bei den eingestellten fünf Spalten umfasst jede Zelle also genau $\frac{17{,}2^\circ}{5} = 3{,}44^\circ$ und erstreckt sich damit jeweils von $-1{,}72^\circ$ (linker Rand der Zelle) bis 1,72° (rechter Rand der Zelle). Der Mauszeiger steht bei $-0{,}86^\circ$, was einer Position etwas links von der Mitte der Zelle entspricht. Die Angabe der Y-Position = 30% bezieht sich ebenfalls auf die Zelle und demonstriert die Auswirkung der Läsionsparameter *1st Blade propability* und *2nd Blade propability*, da deren eingestellte Werte von 0,3 gerade 30% entsprechen.
Die Parameter der Läsion definieren also die Kurve für den Bereich einer einzelnen Zelle. Dabei wird zwischen den einzelnen Zellen nicht unterschieden, sodass die sich ergebende Kurve in gleicher Form auf jede Zelle angewendet wird. Somit wird eine Läsion geschaffen, welche sich auf jedes dargestellte Objekt gleich auswirkt und als objektzentriert angesehen werden kann.

Der Dialog liegt ebenfalls im Paket `de.uniol.jmorsel.rcp.dialogs` und wurde durch folgende Klassen realisiert:

- `ObjectLesionDialog`
  Ist von der Klasse `Dialog` aus dem SWT abgeleitet und stellt ein konfigurierbares Dialogfenster zur Verfügung.

- `ObjectLesionDialogComposite`
  Stellt den grafischen Inhalt des Dialoges dar und enthält die GUI Elemente.

In JMorsel können beide Läsionsformen entweder separat, oder auch parallel ausgewählt werden. Sind beide gleichzeitig aktiviert, so ergibt sich für jeden Punkt des visuellen Feldes eine Übertragungswahrscheinlichkeit, welche sich aus den multiplizierten Werten der beiden Läsionen ergibt. Abbildung 7.7 zeigt dies anhand der nun erweiterten Visualisierung der Läsionen.

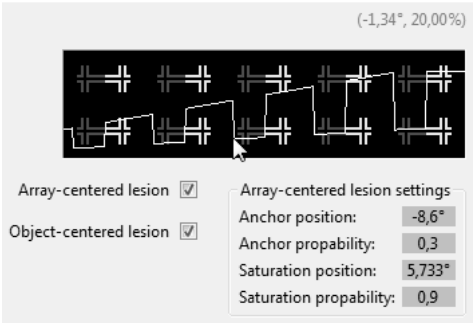


Abbildung 7.7: Visualisierung der kombinierten Läsionen

Die Läsionsparameter entsprechen den Standardwerten, um einen Vergleich

mit den Abbildungen 7.3 und 7.6 zu ermöglichen. Dabei ist der multiplikative Zusammenhang der Werte der Läsionskurven zu erkennen. Zusätzlich sind die dargestellten Objekte mehr oder weniger grau eingefärbt, um noch einmal den Einfluss der ihnen jeweils zugeordneten objektzentrierten Läsion zu verdeutlichen. Im unteren Teil der Abbildung sind zudem die Kontrollelemente zum Ein- und Ausschalten der beiden Läsionsformen, sowie die Parametereingabe der beobachterzentrierten Läsion dargestellt. Der Konfigurationsdialog für die objektzentierte Läsion wurde bereits in Abschnitt 7.5 vorgestellt und ist deshalb an dieser Stelle nicht erneut abgebildet.

Es mussten neue Konnektoren im Paket `de.uniol.jmorsel.connectors` implementiert werden, um diese Funktion in JMorsel zu integrieren:

- `PartialConnector`
  Realisiert einen Konnektor, welcher nur in einem definierten, rechteckigen Bereich agiert. Dieser Bereich wird durch Angabe der linken oberen sowie der rechten unteren Ecke als Punkte in prozentualen Koordinaten des visuellen Feldes angegeben. Eine Angabe der Punkte $(0, 0)$ und $(0.5, 1)$ würde also genau die linke Hälfte des visuellen Feldes definieren, da SWT-Koordinaten immer relativ zu der linken oberen Ecke (also $(0, 0)$) eines Elementes definiert werden. Der Konnektor erhält als Eingabe eine Instanz eines `AbstractConnector` und generiert seine Ausgabe in der Form, dass Werte innerhalb des definierten Bereiches ohne Änderung ausgegeben, Werte außerhalb jedoch auf 0 gesetzt und damit abgeschnitten werden.

- `MultiSumConnector`
  Realisiert einen Konnektor, welcher beliebig viele Instanzen der Klasse `AbstractConnector` als Eingabe erhält und deren Ausgaben für jeden Punkt aufsummiert.

- `SequentialConnector`
  Dieser Konnektor ermittelt seine Ausgabe im Gegensatz zu der Klasse `MultiSumConnector` nicht durch Aufsummierung, sondern multiplikativ. Er erhält genau zwei Instanzen des `AbstractConnector` als Eingabe und wendet diese sequentiell auf die zu verarbeitenden Daten an, indem zunächst die Ausgabe des ersten Konnektors ermittelt wird, welche dann als Eingabe des zweiten Konnektors dient. Dessen Ausgabe wird dann als Gesamtausgabe zurückgegeben. Dies entspricht mathematisch einer Multiplikation der Übertragungswahrscheinlichkeiten der Eingabekonnektoren.

Diese Konnektoren werden nun so angewendet, dass zunächst für jede Zelle ein für die jeweiligen Zellenkoordinaten definierter `PartialConnector` erstellt wird. Diese werden dann zusammen einem `MultiSumConnector` als Eingabe übergeben, welcher aus den Läsionen der Einzelzellen eine Gesamtläsion für das ganze Feld ermittelt. Dieser Konnektor wiederum wird

zusammen mit dem `GradientConnector` der beobachterzentrierten Läsion (falls vorhanden) als Eingabe für einen `SequentialConnector` verwendet, um daraus die in Abbildung 7.7 dargestellte kombinierte Läsion zu bilden.

7.6 Experiment benennen

Bei der Erstellung eines neues Experimentes mit dem Menüpunkt **File, New Experiment** wird nach der Auswahl der zu öffnenden Bilddatei ein Dialogfenster angezeigt, welches den Anwender dazu auffordert, dem neuen Experiment einen Titel zu geben. Als Grundeinstellung ist hier der Name der geöffneten Bilddatei, eingerahmt durch spitze Klammern, angegeben. Abbildung 7.8 zeigt den Eingabedialog. Der gewählte Name wird dann in der Titelleiste des Editors angezeigt (vgl. Abbildung 7.9).

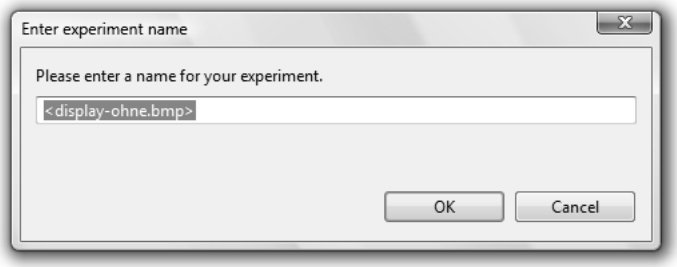


Abbildung 7.8: Eingabedialog für den Titel des Experimentes

7.7 Parallele Experimente

Wie in Abschnitt 5.1.2 beschrieben, liefert das RCP Framework ein mächtiges Containerkonzept für den Aufbau einer komplexen GUI. Dies lässt sich für die Realisierung von mehreren parallelen Experimenten nutzen. In Abschnitt 5.3.2 wurde erläutert, dass jedem `ExperimentEditor` eine Instanz der Klasse `ExperimentEditorInput`, und diesem jeweils ein `Experiment` zugeordnet ist. Mittels dieser Struktur ist es möglich, mehrere Editoren auf verschiedenen Dateien zu öffnen, und diese parallel in der Anwendung darzustellen und zu verwenden. Dazu wurde der Menüpunkt **File, Load** in **File, New Experiment** abgeändert. Abbildung 7.9 zeigt die Titelleisten mehrerer geöffneter Editoren. Der zur Zeit aktive Editor ist farblich hinterlegt. Zudem sind den Titelleisten jeweils Tooltips zugeordnet, welche den Pfad der geöffneten Bilddatei anzeigen.

Abbildung 7.9: Mehrere parallel geöffnete Experimente

7.8 Trennung der Hemisphären

In Experimenten, wie sie von z.B. Hildebrandt et al. [9] durchgeführt werden, wird grundsätzlich genau definiert, an welcher Stelle des Gehirns eine Läsion vorliegt. Dabei ist es insbesondere wichtig zu unterscheiden, in welcher Hemisphäre sich diese Läsion befindet, da die vorherrschende Meinung ist, dass eine funktionale Trennung zwischen den beiden Hirnhälften besteht. Zusätzlich zu dem in Abschnitt 2.2 aufgeführten Zitat aus [7, S. 212] heißt es in [7, S. 218]: „[...] Man muss deshalb annehmen, daß die zusammenfassende Verarbeitung sensibler und sensorischer Informationen in ein räumliches Schema, das nicht analog, sondern propositionell zu denken ist, zu den nicht wenigen Funktionen gehört, für welche eine funktionelle Asymmetrie der Großhirnhemisphären zugunsten der rechten Hirnhälfte besteht."
Aufgrund dieser Annahmen wurde in JMorsel ein Mechanismus zur Separation der beiden Hemisphären eingefügt, welcher eine individuelle Läsionierung und Analyse erlaubt. Dieser wurde auf GUI-Ebene durch Verdopplung der grafischen Elemente aus Abbildung 5.3 im Abschnitt 5.2.1 realisiert. Zusätzlich wurde eine Kontrollstruktur zur Kombination der Simulationsergebnisse geschaffen. Abbildung 7.10 zeigt die Elemente dieser Struktur. Der obere

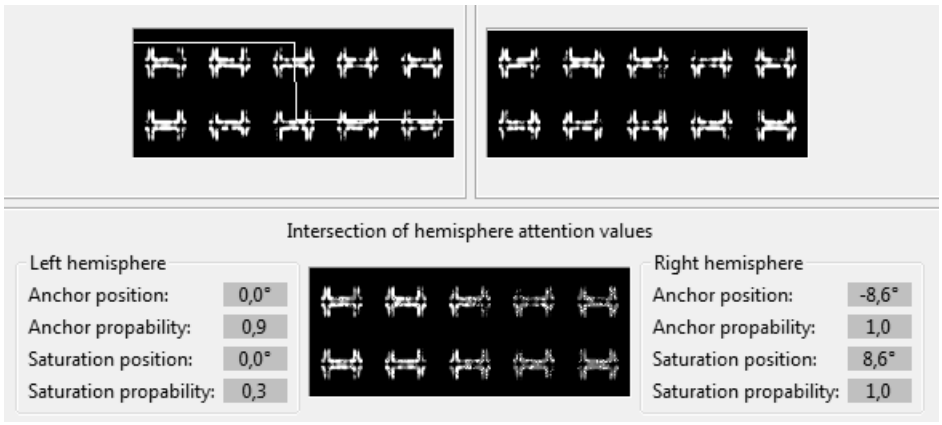


Abbildung 7.10: Kombination der Hemisphären

Bereich stellt exemplarische Simulationsergebnisse der beiden Hemisphären dar. Im unteren Bereich befinden sich Eingabemasken zur Definition der

Läsionen, welche die Verknüpfung zwischen den Hemisphären beeinträchtigen. Diese Masken entsprechen den Eingabemasken der beobachterzentrierten Läsion und verhalten sich äquivalent. Auch an dieser Stelle werden Konnektoren der Klasse `GradientConnector` verwendet, welche die Simulationsergebnisse der Hemisphären entsprechend filtern. Aus den Ausgaben der Konnektoren wird dann für jeden Punkt des visuellen Feldes das arithmetische Mittel gebildet und in dem zentriert zwischen den Eingabemasken positionierten Ausgabefeld dargestellt. Wie auch bei der Visualisierung der Läsion in Abschnitt 7.2 wird hier in das Feld der Quelldaten eine Kurve gezeichnet, welche die eingegebenen Parameter widerspiegelt.
In dem abgebildeten Beispiel beeinträchtigt die Läsion der Ausgabe der linken Hemisphäre die gesamte rechte Hälfte der Daten, während die rechte Hemisphäre nicht läsioniert ist: die Übertragungswahrscheinlichkeiten sind dort über das ganze Feld als $1 = 100\%$ definiert. Die Auswirkung dieser Parameter ist in der zentrierten Ausgabe zu erkennen: Objekte auf der rechten Seite des Feldes wurden nur geschwächt markiert, da für diesen Bereich die linke Hemisphäre kaum Daten geliefert hat.
Für diese Funktionalität mussten in JMorsel neben der Duplizierung der Editorinhalte und Aufteilung der Klasse `AttentionalMechanism` in zwei gleiche Teile lediglich die grafischen Elemente der im vorhergehenden Absatz erläuterten Kombinationsstruktur als neue Klasse implementiert werden. Diese befindet sich im Paket `de.uniol.jmorsel.rcp.editors`, heißt `ConnectionComposite` und wird als regulärer SWT Container in den Editor eingefügt.

7.9 Batch-Modus

Für die in [9] verwendeten visuellen Eingabedaten existieren 20 mögliche Positionen, an denen der kritische Reiz auftreten kann. Eine statistische Analyse der Ausgabedaten von JMorsel erfordert demzufolge eine große Menge von Simulationsdurchläufen auf diversen Eingabedaten. Es hat sich gezeigt, dass die manuelle Durchführung dieser Experimente sehr zeitaufwändig und mühsam ist. Daher wurde ein Stapelverarbeitungsmodus implementiert, welcher alle in einem angegebenen Verzeichnis enthaltenen Bilddateien mit vorgegebenen Einstellungen analysiert und die Simulationsergebnisse nacheinander in eine Ergebnisdatei schreibt. Dieser Modus ist über den Menüeintrag **File, Batch experimenting...** zu aktivieren. Bevor der Durchlauf startet, werden einige Dinge vom Benutzer abgefragt. Als erstes muss das zu verarbeitende Verzeichnis angegeben werden (Abbildung 7.11).
Anschließend wird abgefragt, welche Simulationsergebnisse exportiert werden sollen. Zur Auswahl stehen hier die linke Hemisphäre, die rechte Hemisphäre, oder die Kombination aus beiden (Abbildung 7.12).

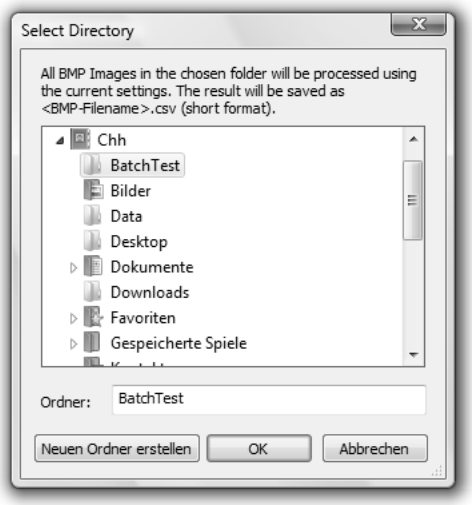


Abbildung 7.11: Auswahl des zu verarbeitenden Verzeichnisses

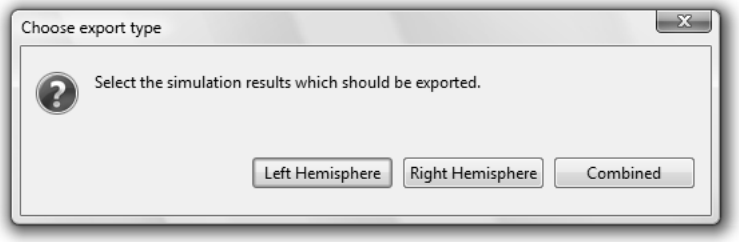


Abbildung 7.12: Auswahl der zu exportierenden Daten

Zuletzt muss noch angegeben werden, in welcher Datei die Ergebnisse gespeichert werden sollen. Voreingestellt ist hier die Datei „JMorselExport.csv“ in dem im ersten Schritt ausgewählten Verzeichnis (Abbildung 7.13). Der dargestellte „*Select destination file*“-Dialog ist betriebssystemabhängig und kann sich von System zu System in seinem Aussehen unterscheiden.

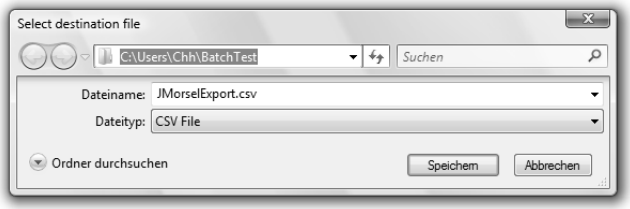


Abbildung 7.13: Auswahl der Zieldatei

Nach Angabe der Zieldatei führt JMorsel dann Simulationen für alle gefundenen Dateien durch. Der Durchlauf ist beendet, wenn kein geöffnetes Experiment mehr zu sehen ist. Während des Durchlaufes sind die Eingabeelemente der GUI nicht gesperrt, es wird jedoch empfohlen, nicht in die Simulationen einzugreifen.
Bemerkung: Im Batch-Modus wurde die Live-Ansicht der Simulationen deaktiviert, um eine möglichst schnelle Fertigstellung der Verarbeitung zu erreichen.

Die Realisierung des Batch-Modus erforderte die Implementation einer neuen Klasse `de.uniol.jmorsel.rcp.actions.BatchExperimentAction`, sowie hinsichtlich des automatisierten Ablaufes eine Anpassung der Klassen `AttentionalMechanismAction`, `NewExperimentAction` und `ExportWizard`. Die drei oben beschriebenen Dialoge werden zur Laufzeit dynamisch erstellt und basieren auf den Klassen `DirectoryDialog`, `MessageDialog` und `FileDialog` aus dem SWT.

Kapitel 8

Gesamtüberblick

In den vorhergehenden Kapiteln wurden bisher nur Ausschnitte der Anwendung gezeigt. In diesem Kapitel soll JMorsel als Ganzes vorgestellt und die Bedienung der Software anhand von Bildschirmfotos erläutert werden.

8.1 Installieren und Starten der Anwendung

In Abschnitt 5.1.2 wurde bereits erwähnt, dass das Eclipse RCP Framework einen Produktexport für diverse Plattformen bietet. Dabei wird eine Startdatei erstellt, welche auf dem Zielsystem direkt lauffähig ist und das Produkt startet. Die Zielplattform für JMorsel ist zur Zeit ausschließlich Windows, weshalb ich mich im Folgenden auf Beschreibungen für dieses System beschränken werde.
Im Auslieferungszustand befindet sich JMorsel in einem selbstextrahierenden Archiv, welches mit einer Testversion der Software WinRAR erstellt wurde (siehe [17]). Nach der Ausführung dieser Datei wird der Anwender nach einem Zielverzeichnis für JMorsel gefragt. In dem angegebenen Verzeichnis wird der Ordner *jmorsel* mit den Inhalten des Produktes erstellt. Die Software beansprucht aufgrund der integrierten Java Laufzeitumgebung etwa 90MB Speicher auf dem Datenträger. Nach dem Extraktionsvorgang kann JMorsel direkt durch Ausführen der Datei *jmorsel.exe* gestartet werden. Die Konfigurationsdateien der Anwendung (wie etwa die automatisch gespeicherten Experimenteinstellungen) werden in dem Ordner *workspace* abgelegt, welcher beim ersten Start im Programmverzeichnis angelegt wird. Während die Equinox-Plattform das Plugin „JMorsel“ lädt (das Plugin-Konzept wurde in Abschnitt 5.1.2 vorgestellt) wird der in Abbildung 8.1 dargestellte Ladehinweis angezeigt. Kurz darauf erscheint dann die Hauptansicht von JMorsel. Diese ist aufgrund der Komplexität der in einem Experiment dargestellten grafischen Elemente auf die initiale Größe von 785×720 Pixel festgelegt. Die Größe kann durch ziehen mit der Maus an den Rändern des Fensters angepasst werden.

Abbildung 8.1: Ladehinweis von JMorsel

Der Anwender kann nun über den Menüeintrag **File, New experiment** oder den entsprechenden Button der Toolbar eine Bilddatei laden und damit ein neues Experiment beginnen. Alternativ kann auch der Stapelverarbeitungsmodus aktiviert werden (siehe Abschnitt 7.9).

8.2 Beschreibung der GUI

Nachdem der Anwender JMorsel gestartet und über den Menüeintrag **File, New experiment** oder dem entsprechenden Button der Toolbar eine Bilddatei geladen hat, erscheint der in Kapitel 5 eingeführte Editor mit den in Kapitel 7 vorgestellten Erweiterungen. Abbildung 8.2 zeigt die typische Ansicht von JMorsel mit einem exemplarisch durchgeführten Simulationslauf. Direkt unter der Titelleiste befinden sich die Menüs der Anwendung mit den in Abschnitt 5.2.3 vorgestellten Einträgen. Darunter befindet sich die Toolbar mit folgenden Buttons (von links nach rechts):

- **New experiment**
  Lädt eine Bilddatei und startet ein Experiment in einem neuen Editor.
- **Export results**
  Öffnet den ExportWizard zum exportieren der aktuellen Simulationsergebnisse.
- **Show equation view**
  Dieser Button verhält sich wie ein An-Aus-Schalter. Er bewirkt die Anzeige des Views zur Konfiguration der Updategleichung, welcher initial am rechten Fensterrand positioniert wird. Der View kann mit der Maus an beliebige Stellen verschoben werden, sogar außerhalb des Programmfensters (vgl. Abbildung 8.3). Während der View sichtbar ist, erhält der Button ein Aussehen, welches den Status *Eingeschaltet* darstellt. Wählt man den Button nun erneut, oder klickt auf das Kreuz neben dem Titel des Views, so erhält der Button wieder das Aussehen, welches den Status *Ausgeschaltet* darstellt, und der View wird geschlossen.
- **Display Settings**
  Öffnet den in Abschnitt 7.3 vorgestellten Dialog zur Konfiguration der Größe des visuellen Displays.

Abbildung 8.2: Typische Ansicht von JMorsel

- **Load settings...**
  Lädt Einstellungen aus einer `.settings` Datei.
- **Save settings...**
  Speichert die aktuellen Einstellungen in eine `.settings` Datei.
- **Revert settings**
  Setzt die Einstellungen auf die Standardwerte zurück.
- **Perform simulation**
  Wird das Dreieckssymbol angezeigt, so hat dieser Button die gleiche Auswirkung wie der *Start*-Button im `ControlComposite` des Editors (vgl. Abschnitte 5.2.1 und 5.3.2). Während die Simulation läuft, wird ein Quadratsymbol angezeigt, und eine Aktivierung des Buttons würde dem *Stop*-Button entsprechen.

Anschließend folgen die sog. *Reiter* der geöffneten Editoren. Der Bereich darunter stellt den Inhalt des aktiven Editors dar. Die einzelnen Komponenten wurden bereits in den Kapiteln 5 und 7 vorgestellt. Anzumerken ist lediglich, dass die Kontrollelemente der Simulation aufgrund der Teilung der Hemisphären nach unten verschoben wurden.

Abbildung 8.3 zeigt die Möglichkeit der freien Positionierung der einzelnen Container in JMorsel. Im Hauptfenster sind zwei geöffnete Editoren zu

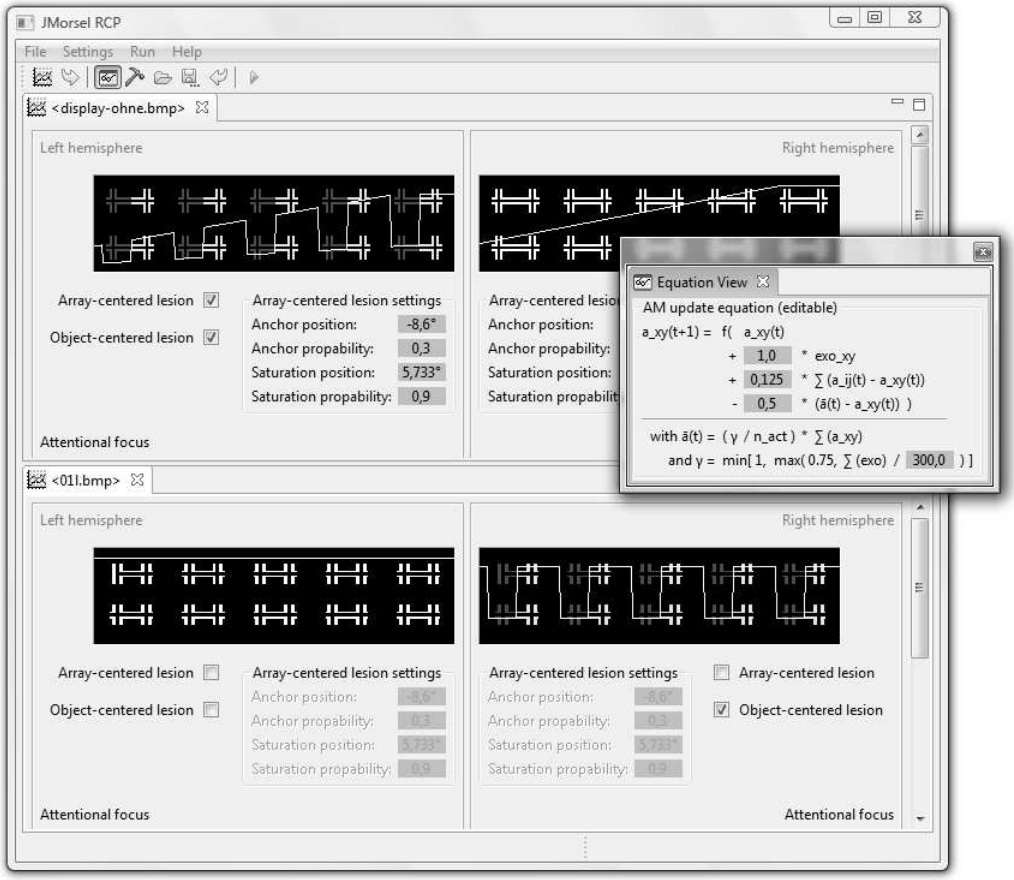


Abbildung 8.3: Frei positionierbare Container in JMorsel

sehen (<*display-ohne.bmp*> und <*01l.bmp*>), welche vertikal übereinander geschoben wurden, sodass sie sich das Hauptfenster teilen und ein direkter visueller Vergleich der eingestellten Parameter möglich ist. Überlappend darüber befindet sich der View zur Konfiguration der Updategleichung. Dieser wurde nach außerhalb des Programmfensters verschoben und erhielt dadurch ein eigenes Fenster, welches sich ebenfalls frei bewegen lässt.

Kapitel 9

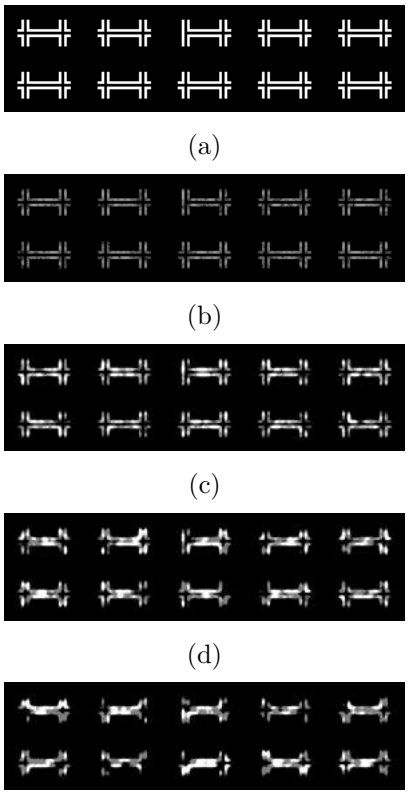
Zusammenfassung

9.1 Evaluation der Software

Die Anwendung besteht in der zur Zeit des Druckes aktuellen Version aus 6893 Zeilen Code in 66 Klassen. Aufgrund dieser Komplexität konnte die geforderte Evaluation hinsichtlich der Reproduzierung der Ergebnisse der Experimente von Hildebrandt et al. nicht mehr durchgeführt werden. Bei den durchgeführten Testläufen zeichnete sich allerdings ein Trend ab, welcher in Bezug auf diese Anforderung gewertet werden kann. In Abbildung 9.1 ist das Ergebnis eines typischen Simulationslaufes unter den nichtläsionierten Standardeinstellungen mit mehreren Simulationslängen dargestellt. Die Abbildungen 9.2 und 9.2 zeigen die dazugehörigen Exportdaten in grafischer Form. Die Grafiken beinhalten jeweils eine Kurve für den Durchlauf mit 5, 10, 15 und 20 Schritten. Die obere Grafik zeigt die Ergebnisse für die Elemente 1 bis 5 (obere Zeile des Eingabebildes), die untere Grafik zeigt die Ergebnisse für Elemente 6 bis 10 (untere Zeile des Eingabebildes).
Das Eingabebild enthielt bei Item 3 (obere Zeile, mittleres Item) auf der linken Seite den kritischen Reiz. Meiner Ansicht nach ist recht deutlich zu erkennen, dass dieser Reiz in keiner der Simulationen gesondert markiert wurde. Längere Laufweiten der Simulation brachten keine nennenswerten Veränderungen. Ich habe diverse Kombinationen von Parametern in der Updategleichung getestet, konnte aber keine Möglichkeit finden, den Reiz separat zu detektieren. Ich denke, dies liegt daran, dass sich der Zielreiz im Vergleich zu Mozers Experimenten nicht stark genug von den übrigen Objekten abhebt.

9.2 Ausblick

JMorsel implementiert nur einen recht geringen Teil des von Mozer vorgestellten Modells. Es wäre daher denkbar, weitere Komponenten aus MORSEL bzw. BLIRNET zu implementieren, um ein vollständigeres System zu erhal-



(a) (b) (c) (d) (e)

Abbildung 9.1: Eingabebild (a) mit Simulationsergebnisse für (b) 5, (c) 10, (d) 15 und (e) 20 Simulationsschritten

ten. Dieses würde auf andere Fragestellungen abzielen, könnte aber auch für die vorliegenden Experimente von Nutzen sein, da der von Hildebrandt et al. vorgeschlagene Ansatz der Simulation über die Aufmerksamkeitsarchitektur nur eine von mehreren Möglichkeiten ist, die Ausfälle von läsionierten Nervensystemen zu erklären.
Zusätzlich ist JMorsel an sich ausbaufähig. Bei meiner Arbeit fiel mir immer wieder auf, dass die Simulation einer seriellen visuellen Suche nur bedingt durch Mozers Modell des AttentionalMechanism zu realisieren ist. Ich könnte mir vorstellen, dass eine Spezifizierung der Software auf Suchaufgaben hilfreich wäre. Dies könnte beispielsweise durch einen visuellen Puffer umgesetzt werden, welcher sich nach einer Rasterisierung des visuellen Feldes die bisher betrachteten Objekte merkt und jeweils miteinander verlgeicht, um den kritischen Reiz ausfindig zu machen. Dies könnte jedoch die generellen Gültigkeit des Modells einschränken.
Als weiteren Ansatzpunkt sehe ich die Arbeitsweise der Updategleichung. In MORSEL war diese dafür zuständig, die Aufmerksamkeit im visuellen

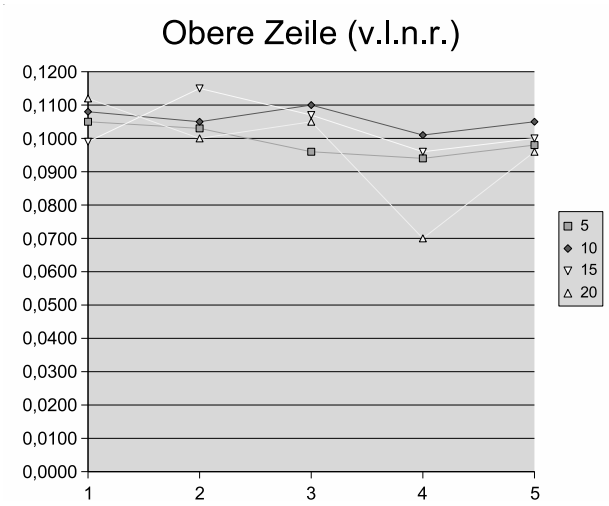


Abbildung 9.2: Export der Simulationsergebnisse: obere Zeile

Feld auf einen zusammenhängenden Bereich zu beschränken. Dies konnte in JMorsel mit den gegebenen Testdaten nicht erreicht werden, was meiner Meinung nach an der Größe des visuellen Feldes bzw. der Menge der darauf abgebildeten Objekte liegt. Mit einer sehr langen Simulationsdauer ließ sich dieser Effekt zwar nachbilden, jedoch sind nach einer so einer Simulation kaum noch Informationen über die Position oder Form der abgebileten Items zu erkennen, was der gestellten Aufgabe nicht entgegenkommt. Ich denke, die Updategleichung müsste an die neuen Gegebenheiten angepasst werden, indem etwa ein weiterer Term eingeführt wird, welcher die Aktivität featurebasiert regelt. Dieser Term könnte zum Beispiel so arbeiten, dass er Regionen hervorhebt, welche in einem großen Verbund zu einem einzelnen Feature gehören, und entgegengesetzt Regionen unterdrückt, in welchen nur wenige Bildpunkte einem Feature angehören. Dies würde die Aufgabenstellung unterstützen, da der in den Testdaten vorhandene kritische Reiz durch eine „fehlende Lücke“, also so gesehen durch ein größer ausgedehntes Feature definiert wird.

9.3 Fazit

Die Entwicklung von JMorsel war ein Feedback-basierter Prozess und es war eine Reihe von aufeinander aufbauenden Prototypen notwendig, um das Endprodukt zu erstellen. Da in dem vorliegenden Fall der Anwender bzw. Auftraggeber aus einem anderen Fachbereich als der Entwickler stammte,

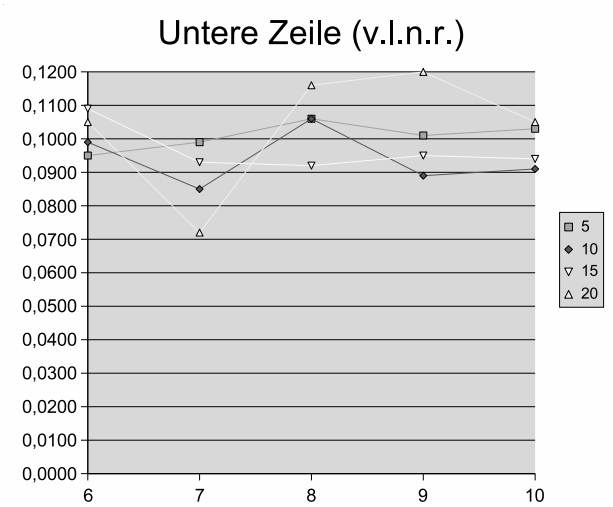


Abbildung 9.3: Export der Simulationsergebnisse: untere Zeile

mussten oftmals Anforderungen erweitert oder angepasst werden. Diese Anpassungen ergaben sich aus dem Test des jeweils aktuellen Protoypen und wurden meist bei der Benutzung der GUI entdeckt. Im Laufe der Zeit hat sich ein kontinuierlicher Kommunikationsprozess eingestellt, welcher die Entwicklung der Software stark erleichterte.
Ich habe versucht, eine möglichst anwenderfreundliche Software mit vielen grafischen Hilfestellungen und Prozessvisualisierungen zu entwickeln. Zusätzlich habe ich auf Erweiterbarkeit sowie Strukturiertheit geachtet. Die Arbeit an JMorsel hat mein Interesse an dem zugrunde liegenden Thema noch weiter verstärkt und ich werde mich weiterhin damit beschäftigen. Mir ist deutlich geworden, wie viel Arbeit die modellhafte Rekonstruktion selbst eines so „kleinen“ Ausschnittes des menschlichen Nervensystems mit sich bringt, und wie viel in diesem Bereich noch getan werden kann.

Ich möchte an dieser Stelle Prof. Dr. Helmut Hildebrandt danken, welcher mir insbesondere im Bereich der Psychologie und Neurophysiologie Anregungen und Hilfestellungen gab.

Anhang A

Konfigurationsdateien

A.1 plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension id="application"
         point="org.eclipse.core.runtime.applications">
      <application>
         <run
            class="de.uniol.jmorsel.rcp.Application">
         </run>
      </application>
   </extension>
   <extension point="org.eclipse.ui.perspectives">
      <perspective
            class="de.uniol.jmorsel.rcp.Perspective"
            fixed="true"
            id="de.uniol.jmorsel.perspective"
            name="Perspective">
      </perspective>
   </extension>
   <extension point="org.eclipse.ui.editors">
      <editor
            class="de.uniol.jmorsel.rcp.editors.ExperimentEditor"
            default="true"
            extensions="bmp"
            icon="icons/spreadsheet-7.ico"
            id="de.uniol.jmorsel.rcp.editors.ExperimentEditor"
            name="ExperimentEditor"/>
   </extension>
   <extension point="org.eclipse.ui.views">
      <view
            class="de.uniol.jmorsel.rcp.views.EquationView"
            icon="icons/TableView.ico"
            id="de.uniol.jmorsel.EquationView"
            name="Equation View"/>
      <view
            allowMultiple="false"
            class="de.uniol.jmorsel.rcp.views.LightweightConsoleView"
            id="de.uniol.jmorsel.ConsoleView"
            name="System Output"/>
   </extension>
   <extension id="product"
         point="org.eclipse.core.runtime.products">
      <product
            application="de.uniol.jmorsel.application"
            name="JMorsel VisualAttention Analyzer">
      </product>
   </extension>
</plugin>
```

A.2 jmorsel.product

```
<?xml version="1.0" encoding="UTF-8"?>
<?pde version="3.1"?>
<product name="JMorsel VisualAttention Analyzer"
      id="de.uniol.jmorsel.product"
      application="de.uniol.jmorsel.application"
      useFeatures="false">
    <aboutInfo>
        <text>
JMorsel VisualAttentionAnalyzer
(c) 2007 Christian Hinrichs
Christian.Hinrichs@online.de
        </text>
    </aboutInfo>
    <configIni use="default"/>
    <launcherArgs>
    </launcherArgs>
    <windowImages/>
    <launcher name="jmorsel">
        <win useIco="true">
            <ico path="/de.uniol.jmorsel/icons/spreadsheet-7.ico"/>
            <bmp/>
        </win>
   </launcher>
   <plugins>
        <plugin id="com.ibm.icu"/>
        <plugin id="de.uniol.jmorsel"/>
        <plugin id="org.eclipse.core.commands"/>
        <plugin id="org.eclipse.core.contenttype"/>
        <plugin id="org.eclipse.core.expressions"/>
        <plugin id="org.eclipse.core.jobs"/>
        <plugin id="org.eclipse.core.runtime"/>
        <plugin id="org.eclipse.core.runtime.compatibility.auth"/>
        <plugin id="org.eclipse.core.runtime.compatibility.registry"
          fragment="true"/>
        <plugin id="org.eclipse.equinox.common"/>
        <plugin id="org.eclipse.equinox.preferences"/>
        <plugin id="org.eclipse.equinox.registry"/>
        <plugin id="org.eclipse.help"/>
        <plugin id="org.eclipse.jface"/>
        <plugin id="org.eclipse.osgi"/>
        <plugin id="org.eclipse.swt"/>
        <plugin id="org.eclipse.swt.win32.win32.x86"
          fragment="true"/>
        <plugin id="org.eclipse.ui"/>
        <plugin id="org.eclipse.ui.workbench"/>
   </plugins>
</product>
```

Anhang B

Dateien der Ausgabe

B.1 Settings-Datei

```
#JMorsel Settings File
#Thu Jul 05 15:21:14 CEST 2007
internal.timestamp=1183641674313

#-- Definierte Konstanten
Settings.LESION_MODE_DISABLE=1
Settings.LESION_MODE_GRADIENT=2
Settings.LESION_MODE_RECTANGLE=3
Settings.EXPORT_TYPE_LEFT=4
Settings.EXPORT_TYPE_RIGHT=5
Settings.EXPORT_TYPE_COMBINED=6

#-- Parameter der Updategleichung
Settings.maxSteps=5
Settings.biasWeight=1.0
Settings.cooperationWeight=0.125
Settings.competitionWeight=0.5
Settings.activityThresholdParameter=300.0

#-- Export-Einstellungen
Settings.exportType=6
Settings.columns=5
Settings.rows=2
Settings.detailedExportFormat=false
Settings.includeSettingsInExport=false

#-- Einstellungen des visuellen Feldes
Settings.screenDistance=60
Settings.screenWidth=35
Settings.viewAngleExtension=17.2
Settings.viewAnglePreferred=true

#-- Parameter der Läsionen
Settings.leftLesionArrayEnabled=false
Settings.rightLesionArrayEnabled=false
Settings.leftLesionObjectEnabled=false
Settings.rightLesionObjectEnabled=false
Settings.nonLesionPropability=0.9
Settings.lesionArrayLeft.anchor.y=0.3
Settings.lesionArrayLeft.blade1.y=0.3
Settings.lesionArrayLeft.blade2.y=0.3
Settings.lesionArrayLeft.saturation.x=0.8333333
Settings.lesionArrayLeft.saturation.y=0.9
Settings.lesionArrayRight.anchor.y=0.3
Settings.lesionArrayRight.blade1.y=0.3
Settings.lesionArrayRight.blade2.y=0.3
Settings.lesionArrayRight.saturation.x=0.8333333
Settings.lesionArrayRight.saturation.y=0.9
Settings.lesionConnectionLeft.anchor.y=1.0
Settings.lesionConnectionLeft.blade1.y=1.0
Settings.lesionConnectionLeft.blade2.y=1.0
Settings.lesionConnectionLeft.saturation.x=1.0
```

```
Settings.lesionConnectionLeft.saturation.y=1.0
Settings.lesionConnectionRight.anchor.y=1.0
Settings.lesionConnectionRight.blade1.x=1.0
Settings.lesionConnectionRight.blade1.y=1.0
Settings.lesionConnectionRight.blade2.x=1.0
Settings.lesionConnectionRight.blade2.y=1.0
Settings.lesionConnectionRight.saturation.x=1.0
Settings.lesionConnectionRight.saturation.y=1.0
Settings.lesionObjectLeft.anchor.x=0.1
Settings.lesionObjectLeft.anchor.y=0.9
Settings.lesionObjectLeft.blade1.x=0.1
Settings.lesionObjectLeft.blade1.y=0.3
Settings.lesionObjectLeft.blade2.x=0.5
Settings.lesionObjectLeft.blade2.y=0.3
Settings.lesionObjectLeft.saturation.x=0.5
Settings.lesionObjectLeft.saturation.y=0.9
Settings.lesionObjectRight.anchor.x=0.1
Settings.lesionObjectRight.anchor.y=0.9
Settings.lesionObjectRight.blade1.x=0.1
Settings.lesionObjectRight.blade1.y=0.3
Settings.lesionObjectRight.blade2.x=0.5
Settings.lesionObjectRight.blade2.y=0.3
Settings.lesionObjectRight.saturation.x=0.5
Settings.lesionObjectRight.saturation.y=0.9
```

B.2 Export-Datei (detailliert)

```
"JMorsel Export - Experiment 05.07.2007 13:56:38 <display-ohne.bmp>"
"Raster was 2 rows and 5 columns"
;field number;absolute strength;percentaged fraction

;0;173,711;0,096
;1;187,967;0,104
;2;172,765;0,096
;3;182,173;0,101
;4;177,122;0,098

;5;182,594;0,101
;6;187,471;0,104
;7;175,797;0,097
;8;193,96;0,107
;9;175,004;0,097
```

B.3 Export-Datei (kurz)

```
0,096;0,104;0,096;0,101;0,098;0,101;0,104;0,097;0,107;0,097;1808,566
```

Anhang C

Inhalte der CD-ROM

File System: Joliet

Mode: Single-Session (CD-ROM)

C.1 Bachelorarbeit

Pfad: /

Readme.txt	Beschreibung der Inhalte der CDROM
Ba-Chh.pdf	Diese Bachelorarbeit (PDF-File)
jmorsel_mitJRE6.exe . .	Die Anwendung JMorsel im Auslieferungszustand

C.2 Verzeichnisse

Pfad: /

de.uniol.jmorsel/	Quelldateien der Software als Eclipse 3.2.2 Projekt
Ressourcen/	Kopien der in dieser Arbeit referenzierten Internetressourcen
Testdaten/	Testdaten zur Verwendung in der Software

C.3 Dokumentation

Pfad: /de.uniol.jmorsel/

doc_public/	Javadoc der Quelldateien (nur `public`)
doc_private/	Javadoc der Quelldateien (alles)

Literaturverzeichnis

[1] Bar, M. und. K. Fogel: *Open Source Development with CVS*. Paraglyph, 2003.

[2] Behrmann, M. und. D. C. Plaut: *The interaction of spatial reference frames and hierarchical object representations: evidence from figure copying in hemispatial neglect.*. Cogn Affect Behav Neurosci, 1(4):307–329, Dec 2001.

[3] Bisiach, E. und. C. Luzzatti: *Unilateral neglect of representational space.*. Cortex, 14(1):129–133, Mar 1978.

[4] Damasio, A. R., H. Damasio und. H. C. Chui: *Neglect following damage to frontal lobe or basal ganglia.*. Neuropsychologia, 18(2):123–132, 1980.

[5] Daum, B.: *Rich-Client-Entwicklung mit Eclipse 3.1*. dpunkt.verlag, 2005.

[6] Haralick, L. G. S. R. M.: *Computer and Robot Vision*. Addison-Wesley, 1991.

[7] Hartje, W. und. K. Poeck: *Klinische Neuropsychologie*. Thieme Georg Verlag, 1982.

[8] Hildebrandt, H.: *Der visuo-räumliche unilaterale Neglect*. Seminarfolien, siehe http://www.psychologie.uni-oldenburg.de/helmut.hildebrandt/23912.html (letzter Zugriff: 21.06.2007).

[9] Hildebrandt, H., C. Schütze, M. Ebke, F. Brunner-Beeg und. P. Eling: *Visual search for item- and array-centered locations in patients with left middle cerebral artery stroke.*. Neurocase, 11(6):416–426, Dec 2005.

[10] Julesz, B.: *Textons, the elements of texture perception, and their interactions.*. Nature, 290(5802):91–97, Mar 1981.

[11] Mozer, M. C.: *The Perception of Multiple Objects: A Connectionist Approach (Neural Network Modelling and Connectionism)*. The MIT Press, 1991.

[12] Mozer, M. C.: *Frames of reference in unilateral neglect and visual perception: a computational perspective.*. Psychol Rev, 109(1):156–185, Jan 2002.

[13] Mozer, M. C. und. M. Behrmann: *Reading with attentional impairments: A brain-damaged model of neglect and attentional dyslexias*. Connectionist approaches to natural language processing, S. 409–460, 1992.

[14] Mozer, M. C., P. W. Halligan und. J. C. Marshall: *The end of the line for a brain-damaged model of unilateral neglect*. Journal of Cognitive Neuroscience, 9:171–190, 1997.

[15] Müsseler, J. und. W. Prinz: *Allgemeine Psychologie (Spektrum Lehrbuch)*. Spektrum Akademischer Verlag, 2002.

[16] OSGiAlliance: *OSGi FAQ*. OSGi Alliance, 05 2007. http://www.osgi.org/about/faqs.asp?section=1#q6.

[17] Roshal, A.: *WinRAR archiver, a powerful tool to process RAR and ZIP files*. win.rar GmbH, 07 2007. http://rarlabs.com/.

[18] Schandry, R.: *Biologische Psychologie*. BeltzPVU, 2003.

[19] SunMicrosystemsInc: *Java Technology*. Sun Microsystems, Inc., 05 2007. http://java.sun.com/.

[20] TheEclipseFoundation: *Eclipse - an open development platform*. The Eclipse Foundation, 2007. http://www.eclipse.org/.

[21] TheEclipseFoundation: *Equinox*. The Eclipse Foundation, 05 2007. http://www.eclipse.org/equinox/.

[22] TheEclipseFoundation: *Rich Client Platform*. The Eclipse Foundation, 05 2007. http://wiki.eclipse.org/index.php/Rich_Client_Platform.

[23] TheEclipseFoundation: *SWT: The Standard Widget Toolkit*. The Eclipse Foundation, 05 2007. http://www.eclipse.org/swt/.

[24] Ullenboom, C.: *Java ist auch eine Insel*. Galileo Computing, 2007. http://www.galileocomputing.de/openbook/javainsel6/.

[25] Zell, A.: *Simulation Neuronaler Netze.*. Oldenbourg, 1994.