Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III

„Xulu“

Entwicklung einer generischen Plattform zur Implementierung von Simulationsmodellen am Beispiel der Landnutzungsmodellierung

Diplomarbeit

bei Prof. Dr. Armin B. Cremers

vorgelegt von Martin O.J. Schmitz
Matrikelnr. 131 85 74
schmitzm@bonn.edu

Bonn, 30. November 2005

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Das Gleiche gilt für beigegebene Zeichnungen, Kartenskizzen und Abbildungen.

Bonn, den 30. November 2005

Martin Schmitz

Inhaltsverzeichnis

Erklärung i

Inhaltsverzeichnis iii

Abbildungsverzeichnis vii

Tabellenverzeichnis ix

Abkürzungsverzeichnis xi

Einleitung 1

1 Landnutzungsmodellierung 5
1.1 Zweck und Ziele der Landnutzungsmodellierung 5
1.2 Von den Daten zur Modellierung 6
1.3 Modellierungsansätze 7
1.3.1 Das Untersuchungsgebiet 7
1.3.2 Modellkonzepte 8
1.4 Zusammenfassung 11

2 Landnutzungsmodellierung in der Praxis 13
2.1 IMPETUS 13
2.2 Zellulare Automaten mit MAPMODELS 16
2.2.1 Die MAPMODELS-Anwendung 16
2.2.2 Zellulare Automaten 18
2.2.3 Bewertung von MAPMODELS für die CA-Implementierung . . 21
2.3 CLUE 24
2.3.1 Eingabedaten 25
2.3.2 Modellablauf 27
2.3.3 Bewertung 28
2.4 Technische Probleme derzeitiger LUC-Modellierung 29
2.5 Zusammenfassung 30

3 Anforderungen an die flexible Modellierungsplattform XULU 31
3.1 Anforderungen des Anwenders . . . 32
3.2 Anforderungen des Modell-Entwicklers . . . 34
3.3 Zusammenfassung . . . 36

4 Entwurf von XULU 37
4.1 Übersicht über den Aufbau von XULU . . . 38
4.2 XULU-Registry:
Flexibilität durch dynamische Plugins . . . 40
4.3 XULU-Event-Manager:
Flexibilität durch den Anwender . . . 43
4.4 Modell-Einbettung in XULU . . . 45
4.4.1 Ressourcen-Philosophie . . . 45
4.4.2 Modell-Steuerung . . . 46
4.4.3 Der Weg zum Modell-Ergebnis . . . 48
4.4.4 Modell-Schnittstelle . . . 49
4.4.5 Multi-Modell-Szenarien . . . 49
4.5 XULU-Visualisierung . . . 52
4.6 XULU-Datenpool und Datenbeschaffung . . . 54
4.7 Datenmanipulation in XULU . . . 55
4.8 XULU-Datentypen . . . 56
4.8.1 Objekt-Aufbau und Erzeugung durch Factorys . . . 56
4.8.2 Zugriffskontrolle . . . 59
4.8.3 Schnittstelle zu den Datentypen . . . 60
4.8.4 Entwurf neuer Datentypen . . . 61
4.9 GUI-Komponenten . . . 64
4.10 Zusammenfassung . . . 65

5 Implementierung von XULU 67
5.1 XULU-Skripte . . . 68
5.2 Aufbau der XULU-Applikation . . . 69
5.3 Plugin-Verwaltung durch die XULU-Registrierung . . . 72
5.4 Event-Management . . . 74
5.4.1 Event . . . 74
5.4.2 Handler . . . 75
5.4.3 Event-Handler . . . 76
5.4.4 Ablauf der Event-Managers . . . 76
5.5 Datenverwaltung . . . 79
5.5.1 Datentypen . . . 79
5.5.2 Objekt-Erzeugung durch Factorys . . . 84
5.5.3 Der Datentyp Raster . . . 87
5.6 Modell-Schnittstelle und -Steuerung . . . 89
5.6.1 Modell-Ressourcen . . . 91
5.6.2 Genereller Modell-Ablauf . . . 93
5.7 Zusammenfassung . . . 94

6 Evaluation von XULU anhand des CLUE-Modells **97**
6.1 Implementierung des CLUE-Modells 98
6.1.1 Integration in die Ressourcen-Philosophie 98
6.1.2 Modell-Ablauf 103
6.1.3 GUI 106
6.2 Anwendung 108
6.2.1 Modell-Instanz laden 108
6.2.2 Daten laden 109
6.2.3 Ressourcen zuordnen und Modell initialisieren 110
6.2.4 Modell-Ergebnisse verarbeiten 110
6.3 Bewertung 112
6.3.1 Phase der Modell-Programmierung 112
6.3.2 Modell-Steuerung 113
6.3.3 Effizienz-Vergleich mit dem Original-CLUE 113
6.3.4 Zusammenfassung 115

Zusammenfassung und Ausblick **117**

A XULU-Registry-Datei **I**

B Dateiformat für Datenpool-Skripte **III**

C Dateiformat ARCINFOASCIIGRID **V**

D Dateiformat für dynamische XULU-Objekte **VII**

E CLUE-Datensatz für das IMPETUS-Gebiet **XI**

F Test der JAVA-Polymorphie **XIII**

G SDL – Specification and Description Language **XV**

Literaturverzeichnis **XVII**

Abbildungsverzeichnis

1.1 Typische Vertreter von *Driving Forces*. 7

2.1 Übersicht über die IMPETUS Untersuchungsgebiete (Quelle: [12]) . . . 14
2.2 Das IMPETUS-A-Projekt in Benin (Quelle: [12]) 14
2.3 Das IMPETUS-Untersuchungsgebiet im Quémé-Einzugsgebiet von Benin (Quelle: [32]) 15
2.4 Ein MAPMODELS-Flowchart (Quelle: [24]) 17
2.5 BORGWARDTs Modellstruktur zur Simulation von Siedlungs- und Feldflächen (Quelle: [3]) 19
2.6 Das *MapModel* für eine Iteration der Siedlungsexpansion (Quelle: [3]) . 22
2.7 Überblick über die grobe CLUE-Struktur. Der obere Bereich läuft außerhalb der CLUE-Anwendung ab. (Quelle: [37]) 25

3.1 Die Modellierungsplattform trennt die modellspezifische Semantik von allgemeinen modellunabhängigen Funktionalitäten 34

4.1 Die Kernkonzepte der Modellierungsplattform XULU 37
4.2 Eine (auch intern) komponentenbasiert aufgebaute Modellierungsplattform ermöglicht zentrale Erweiterung und den unabhängigen Austausch einzelner Komponenten. 39
4.3 Die zwei Teile der Modellierungsplattform: *Anwendung* und *Plugins*. . 41
4.4 *Dynamisches Binden*: Im Programmcode wird nur mit den durch das Interface spezifizierten Methoden (1 bis N) gearbeitet, ohne Kenntnis von der konkreten Implementierung. 42
4.5 Eine zentrale Ereignis-Steuerung erhöht wesentlich die Übersichtlichkeit der internen Abläufe. 44
4.6 Die Modell-Semantik (rechts) wird strikt von den Anwendungsfunktionen der Plattform (links) getrennt und beschränkt sich im Wesentlichen auf die Algorithmik. 47
4.7 Der Verlauf von XULU-Programmstart bis hin zum Modell-Ergebnis erfolgt in verschiedenen Phasen. 48
4.8 Zwei unabhängige Modelle, die auf gegenseitigen Modellausgaben operieren. 51
4.9 Die Art der Visualisierung kann nicht an einen Objekttyp gebunden werden. Der Anwender trifft diese Entscheidung manuell anhand der Objekt-Semantik. 53

4.10 Vergleich eines flächendeckend verteilten mit einem stark gestreuten Raster. 57
4.11 Ein möglicher struktureller Aufbau von *Dorf*-Datentypen. 62
4.12 Alle Plattform-Komponenten bestehen aus 2 Teilen: Funktion und GUI. 64

5.1 Das Hauptfenster der XULU-Modelling-Platform enthält für jede Komponente ein eigenes Unter-Fenster. 70
5.2 Der Aufbau der Xulu-Modelling-Platform. Die farbig hinterlegten Klassen stellen instanziierbare Komponenten dar; die gestrichelt umrandeten die Schnittstellen (Interfaces, abstrakte Klassen). 71
5.3 Das *Observer Pattern* (Quelle: [9]) 74
5.4 Ein Event-Handler verbindet ein Ereignis mit einer davon unabhängigen Funktion zu einer Reaktion. 75
5.5 Das Eintreffen eines Ereignisses im Event-Manager. 77
5.6 Der interne Ablauf des Event-Managers (ohne die Definition neuer Event-Handler). 78
5.7 Ableiten von *mehreren* Klassen-Implementierungen ist in JAVA nicht möglich (links); von *einer* Klasse-Implementierung und mehreren Interfaces schon (rechts). 80
5.8 Der Aufbau eines XULU-Objekts. 82
5.9 Der Aufbau einer *Property* (Eigenschaft) eines XULU-Objekts. 83
5.10 Um eines neues Raster zu erzeugen, müssen eine Reihe von Parametern spezifiziert werden. 85
5.11 Der Ablauf eines Datenimport-Vorgangs als SDL-Diagramm. 86
5.12 Der Aufbau der Raster-Datentypen und -Factorys 90
5.13 Eine Ressourcen-Objekt *kapselt* ein Daten-Objekt zwischen Datenpool und Modell. 92
5.14 Das Steuerungs-Fenster wird für jedes Modell auf Basis der geforderten Ressourcen erstellt. 93
5.15 Der Steuerung des Modellierungsablaufs durch den Modell-Manager. . 95

6.1 Das von XULU dynamisch generierte Kontroll-Fenster für das CLUE-Modell. 99
6.2 Ein CLUE-Zeitschritt in Pseudocode. 105
6.3 Die modell-spezifische GUI des CLUE-Modells. 106
6.4 Im Fenster des Modell-Managers sind alle geladenen Modelle aufgelistet. 108
6.5 Um ein neues Raster zu erzeugen, müssen eine Reihe von Parametern spezifiziert werden. 110
6.6 Die Ergebnisse des XULU-CLUE für das erste und zweite simulierte Jahr. 111
6.7 Die Ergebnisse des Original-CLUE für das erste und zweite simulierte Jahr. Veränderungen sind u.a. in den markierten Bereichen erkennbar. . 111
6.8 Bei einem Nachbarschaftsradium von 2 Zellen, werden die schattierten Bereiche sowohl für die Zelle D4, als auch für E4 betrachtet. 115

G.1 Die in dieser Arbeit verwendeten SDL-Symbole. XV

Tabellenverzeichnis

2.1 Die zwei Varianten des CLUE-Modells 24
2.2 Einige wichtige Parameter des CLUE-Modells 26

3.1 Anforderungen des *Anwenders* an die Modellierungsplattform XULU . . 33
3.2 Beispiele für *allgemeine* XULU-Datentypen 35
3.3 Beispiele für *modellspezifische* XULU-Datentypen 35
3.4 Anforderungen des *Modell-Entwicklers* an die Modellierungsplattform XULU 36

4.1 Mögliche benutzerdefinierte Event-Handler bestehen aus einem Ereignis (links) und einer Aktion (rechts). 44
4.2 Übersicht über die Anforderungen an ein Modell. 50
4.3 Übersicht über die Anforderungen an ein XULU-Daten-Objekt. 61

5.1 Plugin-Komponenten, die von der XULU-Registry dynamisch (in Listen) verwaltet werden 72
5.2 Für benutzerdefinierte Event-Handler zur Verfügung stehende Ereignisse 76
5.3 Die Eigenschaften eines `SingleGrid`. Das `MultiGrid` unterscheidet sich davon nur dadurch, dass es statt der skalaren Property „Grid“ eine Liste von Rastern verwaltet. 88
5.4 Der Unterschied zwischen *lokaler* und *globaler* Ressourcen-Konsistenz. 92

6.1 Ressourcen-Definitionen durch den Content-Manager des CLUE-Modells. 102
6.2 Aus dem CLUE-Datensatz erzeugte XULU-Objekte. 109
6.3 Für das CLUE-Modell benötigte XULU-Objekte. 109

Abkürzungsverzeichnis

ABM	**A**gent **B**ased **M**odelling
API	**A**pplication **P**rogramming **I**nterface
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
BWL	**B**etriebs**w**irtschafts**l**ehre
CA	**C**ellular **A**utomaton (Zellularer Automat)
CLUE	The **C**onversion of **L**and **U**se and its **E**ffects
DHM	**D**igitales **H**öhen-**M**odell
GIS	**G**eo-**I**nformations-**S**ystem
GUI	**G**raphical **U**ser **I**nterface (Grafische Benutzeroberfläche)
IMPETUS	**I**ntegratives **M**anagement **P**rojekt für einen **e**ffizienten und **t**ragfähigen **U**mgang mit **S**üßwasser in Westafrika
JVM	**J**ava **V**irtual **M**achine
LUC	**l**and **u**se / land **c**over (Landnutzung/Landbedeckung)
LUCC	**l**and **u**se / land **c**over **c**hange (Landnutzungsänderung)
N.N.	**N**ormal **N**ull
OO	**O**bjekt**o**rientierung, **o**bjekt**o**rientiert
RSRG	**R**emote **S**ensing **R**esearch **G**roup (der Universität Bonn)
SDL	**S**pecification and **D**escription **L**anguage
TU	**T**echnische **U**niversität
UI	**U**ser **I**nterface (Benutzerschnittstelle)
WFS	**W**eb **F**eature **S**ervice
WMS	**W**eb **M**ap **S**erver
XML	**E**xtensible **M**arkup **L**anguage

Einleitung

Einleitung und Motivation

Modellierung zum Zweck der Simulation kommt in den verschiedensten Bereichen der Wissenschaft zum Einsatz[1]. Chemische Prozesse, die Spieltheorie in der BWL, Vorhersagen im Bereich der Meteorologie und die Ausbreitung von Epidemien sind nur einige wenige Beispiele, bei denen sich Forschung und Wissenschaft das Mittel der Modellierung zu Nutze machen.
An allen Stellen, an denen reale Feldversuche zu kostenintensiv, zu langwierig oder zu gefährlich sind, versucht man diese durch ein Modell nachzubilden. Häufig wird Modellierung auch eingesetzt, um die Auswirkungen verschiedener Auswahlmöglichkeiten zu analysieren und nach der Methode *trial-and-error* die geeignetste Variante zu ermitteln (*Decision Support*). An dieser Stelle sind reale Feldversuche oft unmöglich. Man denke beispielsweise an die Standortfrage für ein Hochhaus im Bereich der Städteplanung. Gleiches gilt für alle Einsatzgebiete, in denen man an Zukunftsszenarien interessiert ist. Hier ist Modellierung unumgänglich. Dies gilt speziell auch für die *Landnutzungsmodellierung*, die dieser Diplomarbeit als Anwendungsgebiet dienen soll. Dabei handelt es sich um ein sehr aktuelles Thema, das im Rahmen der zunehmenden globalen Veränderungen besonders für Entwicklungsländer immer mehr an Bedeutung gewinnt. Modellierung und Simulation ermöglichen es, zukünftige Entwicklungen – wie Entwaldung, Wasserverknappung oder Versorgungsengpässe – frühzeitig abzuschätzen. Dabei geht es häufig um Zeiträume von mehreren Jahren und Jahrzehnten. Auf Basis solcher Prognosen können somit rechtzeitig Gegenmaßnahmen eingeleitet werden. Die Landnutzungsmodellierung versucht dabei in zunehmendem Maße, Aspekte aus verschiedenen Wissenschaften zu verbinden (z.B. Fernerkundung und Sozialwissenschaften). Für die Simulation von Landnutzung gibt es keine einheitliche Vorgehensweise, wie ein Modell arbeiten sollte. Im Laufe der Jahre haben sich sehr viele unterschiedliche Ansätze entwickelt. Eine Einteilung in Modell-Klassen ist nur schwer möglich, da Modelle oft auf verschiedenen Ansätzen beruhen, die sich zudem überschneiden.

Anhand der zwei Anwendungen CLUE-S und MAPMODELS, die von der RSRG[2] im Rahmen von IMPETUS eingesetzt werden, verdeutlicht diese Arbeit die verschiedenen Faktoren, welche die Neu- und Weiterentwicklung von Landnutzungsmodellen

[1] Im Folgenden ist mit dem Begriff *Modell*, bzw. *Modellierung* immer ein Simulationsmodell gemeint.
[2] „**R**emote **S**ensing **R**esearch **G**roup“ der Universität Bonn

erschweren. Ein wesentlicher Punkt besteht darin, dass Modell-Applikationen zur Zeit entweder sehr eng mit der jeweiligen Modell-Semantik verknüpft sind, oder nur sehr einfache Modell-Strukturen zulassen, die für die Entwürfe moderner Landnutzungsmodellierung nicht ausreichen. Selbst die Umsetzung von Modell-Ideen erfordert sehr häufig die Implementierung einer kompletten Anwendung (samt Benutzeroberfläche, Datenverwaltung, usw.).

Das Ziel dieser Arbeit besteht deshalb darin, eine allgemeine und modell-unabhängige Plattform zu entwickeln, welche die Implementierung neuer Modelle und Modell-Ideen in vielerlei Hinsicht vereinfacht.
Die Anforderungen, die an eine solche Plattform gestellt werden, sind sehr unterschiedlich, je nachdem aus welchem Blickwinkel man sie betrachtet (Modell-Anwender, Modell-Entwickler, Plattform-Entwickler). Teilweise sind bereits die Anforderungen aus *einem* Blickwinkel schwer zu vereinbaren. Beispielsweise hat der Modell-Entwickler einerseits Interesse daran, dass die Plattform viele Aufgaben generisch löst, um ihm bei der Modell-Implementierung möglichst viel Arbeit zu ersparen. Andererseits möchte er im Entwurf der Modell-Algorithmik nicht durch die Vorgaben der Plattform eingeschränkt werden. Die Modell-Schnittstelle darf somit nicht zu restriktiv sein.
In einem ersten Teil dieser Arbeit werden aus diesen (teilweise konträren) Anforderungen die Kernpunkte herausgearbeitet, die beim Entwurf der Plattform eine Rolle spielen. Ein wichtiger Aspekt besteht in einem geeigneten Tradeoff für die angesprochene Modell-Schnittstelle.
Um eine Vereinfachung der Modell-Implementierung zu erreichen, muss diese auf ein geringes Maß (im Idealfall den Modell-Algorithmus) reduziert werden. Im Fokus des Entwurfs steht deshalb auch die Vereinheitlichung der Modell-Handhabung – wie Modell-Steuerung oder -Parametrisierung –, um diese gleichermaßen für alle Modelle einsetzen zu können. Gleiches gilt für die Datenverwaltung und -visualisierung. Die Vereinheitlichung bildet zudem die Basis dafür, dass einmal implementierte Modell-Ansätze wiederverwendet und miteinander kombiniert werden können.
Darüberhinaus stellt Erweiterbarkeit eine wichtige Richtlinie dar, der beim Plattform-Design gefolgt wird. Diese bezieht sich insbesondere auch auf den Anwendungsbereich der Plattform. Die Landnutzungsmodellierung soll dieser Arbeit nur als *exemplarisches* Einsatzgebiet dienen. Prinzipiell soll die Plattform so generisch entworfen sein, dass sie auch in anderen Bereichen (z.B. Wirtschaftssimulationen) verwendet werden kann.
Dies ist ein Hauptgrund dafür, das in dieser Arbeit entworfene Konzept der ***eX**tendable **U**nified **L**and **U**se Modelling Platform* (kurz: XULU) sehr stark auf der Verwendung von Plugins aufzubauen. Sowohl auf Modellseite, als auch auf Anwendungsseite bleibt XULU somit flexibel erweiterbar.
Ein weiterer zentraler Aspekt wird darüberhinaus sein, dass sich Weiterentwicklungen (z.B. bezüglich Effizienz) *global* durch die Plattform realisieren lassen, also ohne alle bestehenden Modelle (*lokal*) abändern zu müssen.

Um die geschilderten Gesichtspunkte in einer allgemeinen Plattform zu verbinden, wird beim Entwurf und der Implementierung von XULU ein objektorientierter Ansatz verfolgt. An verschiedenen Stellen wird auf bestehenden Entwurfsmustern (*Design Patterns*) aufgebaut. Die Umsetzung in der Programmiersprache JAVA bie-

tet neben den Vorzügen der Betriebssystem-Unabhängigkeit (MICROSOFT WINDOWS, LINUX, MACINTOSH) den Vorteil, dass auf eine Reihe bereits implementierter Software-Lösungen aufgebaut werden kann (z.B. GEOTOOLS).

Eine Evaluation der XULU-Plattform wird anhand einer Implementierung des CLUE-Modells durchgeführt. Diese zeigt, dass der Aufwand zur Implementierung eines Modells durch die Verwendung der XULU-Plattform erheblich reduziert wird. Gleichzeitig stehen für die Modell-Anwendung komfortablere Anwendungs- und Steuerungsfunktionen zur Verfügung, als bei der modell-spezifisch implementierten Original-Applikation CLUE-S, obwohl diese in XULU modell-unabhängig realisiert sind.
Desweiteren verdeutlicht die Evaluation die bereits angesprochene Herausforderung, die konträren Standpunkte von Modell-Anwender und -Entwickler in einer allgemeinen Plattform zu vereinbaren. Die durch XULU für den Anwender errungene Flexibilität bezüglich der Modell-Parametrisierung führt zu Beeinträchtigungen der Benutzerfreundlichkeit während der Modell-Erprobungsphase.
Die durch die Evaluation gewonnenen Erkenntnisse werden dazu genutzt, konkrete Erweiterungsmöglichkeiten zu identifizieren und zu diskutieren, die solche Interessenskonflikte *global* für alle Anwendungsgebiete der XULU-Modelling-Platform ausgleichen.

Aufbau der Arbeit

KAPITEL 1 liefert einen allgemeinen Einblick in das Thema Landnutzungsmodellierung, welches dieser Arbeit als Anwendungs-Beispiel dient. Ansatzweise werden verschiedene Konzepte der Modellierung beschrieben. KAPITEL 2 beleuchtet die zwei exemplarischen Modellierungsapplikationen MAPMODELS und CLUE-S genauer und kristallisiert dabei die Probleme heraus, die sich insbesondere durch eigenständige, modell-spezifische Anwendungen ergeben. Darauf aufbauend fasst KAPITEL 3 die Anforderungen an eine generische Modellierungsplattform zusammen. Dies geschieht einerseits aus dem Blickwinkel des Modell-Entwicklers und andererseits aus Sicht des Anwenders. KAPITEL 4 präsentiert den Entwurf der generischen Plattform XULU, in die Modelle verschiedenster Art integriert werden können. Dabei werden die Konzepte erläutert, die bei der Implementierung von XULU verfolgt werden, um den in Kapitel 3 geschilderten Anforderungen zu genügen. KAPITEL 5 beleuchtet die wichtigsten Aspekte der XULU-Implementierung, sowie die Mittel der Umsetzung in JAVA. Anhand von CLUE beschreibt KAPITEL 6 die Integration eines konkreten Modells in die XULU-Plattform. Anschließend dient dieses authentische Fallbeispiel einer Bewertung, in wie weit die realisierte Anwendung den ursprünglichen Anforderungen gerecht wird. Das abschließende Kapitel fasst die in dieser Diplomarbeit erzielten Ergebnisse zusammen und liefert einen Ausblick, wie XULU durch zukünftige Projekte sinnvoll erweitert werden kann.

Kapitel 1

Landnutzungsmodellierung

Dieses Kapitel soll einen kurzen Einblick in das Thema Landnutzungsmodellierung geben. Hierzu werden einige generelle Ansätze erläutert und gleichzeitig wichtige Fachbegriffe eingeführt, die im Rahmen der Diplomarbeit von Bedeutung sind.

1.1 Zweck und Ziele der Landnutzungsmodellierung

Die Modellierung von Landnutzung/Landbedeckung (LUC) [1] diente in der Vergangenheit vor allem dazu, Gründe und Erklärungen für die Entstehung *aktueller* LUC zu finden. Man ging der Frage nach „Welche Entwicklungen waren verantwortlich für die derzeitige Situation?“, um solche Entwicklungen in Zukunft möglichst früh zu erkennen. Seit einigen Jahren gehen die Bestrebungen der LUC-Modellierung vermehrt in die Richtung, das aus der Vergangenheit Gelernte auf die Zukunft zu projizieren und ganz konkret Szenarien zukünftiger Landnutzung (samt den damit verbundenen Folgen) zu modellieren. Die Projektion aktueller Entwicklungen auf mehrere Jahre und Jahrzehnte in die Zukunft ermöglicht zwar keine exakten Vorhersagen, jedoch kann bereits die Prognose grober Tendenzen und Konsequenzen des aktuellen Handelns zu einer guten Abwägung über die Notwendigkeit eines Entgegenlenkens verhelfen (*Decision Support*). Ein Beispiel ist die stetige Abholzung großer Waldgebiete oder das zunehmende Verschwinden von Feuchtgebieten. Eine ungefähre Vorhersage, in welchen Regionen diese Tendenzen in 20 Jahren akut werden und welche Auswirkungen (⇒ Wasserversorgung, Mangel an Brennstoff, ...) dadurch entstehen, ermöglicht ein entsprechendes Gegenlenken zum jetzigen Zeitpunkt. Dabei ist man zunehmend bestrebt, verschiedene Wissenschaften – insbesondere Fernerkundung (*remote sensing*) und Sozialwissenschaften (*social science*) – miteinander zu verbinden und die gegenseitigen Wechselwirkungen bei der Modellierung zu berücksichtigen [15].
Zwar erscheint auf den ersten Blick kein unmittelbarer Zusammenhang zwischen der direkten Landnutzung (z.B. Siedlungsfläche, Erdnuss- oder Cashew-Anbau, Brachland, Savanne oder dichter Wald) und sozioökonomischen Problemen – wie sozialen Spannungen – zu bestehen. Betrachtet man die Problematik jedoch genauer, so liegen die Ursachen für soziale Spannungen sehr häufig in der Ressourcen-Verknappung, welche direkt auf die LUC-Entwicklung zurückgeführt werden kann (zu wenig Ackerland oder zu wenig Siedlungsfläche für die gesamte Bevölkerung einer Region).

[1] „LUC“ steht für die englische Bezeichnung „land use and land cover“.

1.2 Von den Daten zur Modellierung

LUC-Modellierung basiert auf der Auswertung und Analyse empirischer Daten. Bevor diese für die Modellierung genutzt werden können, müssen diese zunächst erfasst und zum Teil nachbearbeitet werden:

Zustandserfassung ⇒ Klassifizierung ⇒ Einsatz in der Modellierung

Während der Zustandserfassung werden reale Umweltdaten über das Untersuchungsgebiet gesammelt. Hierzu werden Methoden der ...

- Fernerkundung (z.B. Luft- oder Satellitenaufnahmen)
- Naherkundung (z.B. Bodenproben, terrestrische Fotos)
- statistischen Erfassung (z.B. Bevölkerungszahlen, -dichten)

... eingesetzt. Die Fernerkundung läßt eine großflächige Erfassung zu, weshalb ihr eine besondere Bedeutung zufällt. Als Folge bauen LUC-Modelle sehr häufig auf einer rasterförmigen Einteilung des Untersuchungsgebiets auf, da Satellitenaufnahmen in der Regel als Rasterdaten vorliegen. Zudem sind Raster mathematisch einfacher zu analysieren als Vektordaten.

Landnutzung (LUC) und Landnutzungsänderung (LUCC) werden sehr häufig als funktionaler Zusammenhang von bestimmten biologischen und sozio-ökonomischen Variablen beschrieben [35]. Diese (empirisch messbaren) Einfluss-Variablen werden als *Driving Forces* (wörtlich: „treibende Kräfte“) bezeichnet und können zum Teil direkt[2] aus der Datenerfassung übernommen werden. Je nach zu modellierender Problematik werden unterschiedliche *Driving Forces* gewählt. Abbildung 1.1 zeigt einige typische Vertreter.

Häufig ist jedoch für die Analyse auch eine Nachbearbeitung der erfassten Daten notwendig. Zum Beispiel geht insbesondere die zeitliche LUC-Modellierung von einer gegebenen *LUC-Konfiguration* aus. Diese beschreibt einen konkreten Zustand des Untersuchungsgebiets. Da dieses meist sehr groß ist, geschieht die Bestimmung der initialen LUC-Konfiguration nicht manuell, sondern durch die Auswertung von Fernerkundungsdaten [15, 34]. Diese liegen in der Regel als spektrale Informationen über die Landbedeckung vor[3]. Hierüber wird im Rahmen der *Landnutzungsklassifizierung* (kurz: *Klassifizierung*) jedes Landstück (z.B. Rasterzelle) des Untersuchungsgebiets in ein *diskretes* Schema vorgegebener LUC-Typen (z.B. Siedlung, Ackerland, Straße,

[2] bzw. nach Aggregation entsprechend der Einteilung des Untersuchungsgebiets

[3] An dieser Stelle sei auf den Unterschied zwischen Landbedeckung (*land cover*) und Landnutzung (*land use*) hingewiesen (vgl. [5, 34]). Dieser ist im Zusammenhang mit der *Klassifikation* wesentlich. In dieser Arbeit abstrahiere ich jedoch von diesem Unterschied und verwende den Begriff *Landnutzung* als Synonym sowohl für Landnutzung, als auch für Landbedeckung.

• Höhe über N.N. • Hangneigung • Bodenbeschaffenheit • Niederschlag • Erreichbarkeit von Wasser • ...	**Ökosystemare Parameter**
• Bevölkerungsdichte • Nähe zu Industriegebieten • Nähe zu Wohngebieten • Nähe zu Infrastruktur • Einkommen • ...	**Sozio-Ökonomische Parameter**

Abbildung 1.1: Typische Vertreter von *Driving Forces*.

Brachland) eingeteilt. Die Schwierigkeit bei der Klassifizierung besteht darin, allgemein gültige und möglichst eindeutige funktionale Zusammenhänge zwischen den biologischen/spektralen Parametern und den LUC-Klassen zu finden. Erst hierdurch wird eine weitestgehend automatische Klassifizierung ermöglicht.

1.3 Modellierungsansätze

Die ersten Ansätze der LUC-Modellierung erfolgten bereits im 19ten Jahrhundert durch George Perkins und Johann Heinrich von Thünen [5, Kapitel 2.2]. Jedoch sollte man diese Ansätze vielmehr als Studien, denn als Modelle bezeichnen. Die ersten richtigen LUC-Modelle entstanden in den 70er Jahren des letzten Jahrhunderts. Aufgrund steigender Rechnerleistungen setzte in den 90er Jahren eine rasche Entwicklung ein, so dass sich im im Laufe der Zeit sehr viele unterschiedliche Modellierungsansätze entwickelt haben. An dieser Stelle möchte ich jedoch nur auf einige grundlegende Ansätze eingehen, um ein Verständnis für die gängige Vorgehensweise der LUC-Modellierung zu schaffen. Eine ausführlichere Betrachtung findet sich in [5], [1] und [35]. BRIASSOULIS nimmt in [5] sogar eine Strukturierung in Modellklassen vor.

1.3.1 Das Untersuchungsgebiet

In der Regel wird das betrachtete Untersuchungsgebiet in eine Menge von Zonen eingeteilt, für die jeweils eine eigene Landnutzung modelliert wird. Die Partitionierung (Aufteilung) kann weitestgehend beliebig vorgenommen werden und auch einen semanti-

schen Ursprung besitzen (z.B. Stadtbezirke). Weit verbreitet ist jedoch eine gleichmäßige Zonenaufteilung in Form eines Rasters, also in geometrisch gleich große (quadratische) Zellen. Deshalb verwende ich im Folgenden zumeist den Begriff *Zelle*, anstatt von allgemeinen *Zonen* zu sprechen.

Die überwiegende Anzahl an Modellen arbeiten mit diskreten Landnutzungen. Hierbei wird vorab eine feste Menge von *Landnutzungstypen*[4] gewählt, die für das Untersuchungsgebiet betrachtet werden (z.B. Siedlung, Wald, Brachland, Ackerland, städtische Bebauung, Industrie). Während des Modellablaufs werden jeder Zelle dann diese LUC-Typen zugewiesen.
Oft wird dabei eine 1:1-Beziehung vorgenommen, d.h. jeder Zelle j wird *genau ein* LUC-Typ i zugeordnet. Die genaue Lokalisation innerhalb einer Zelle wird nicht modelliert (das Modell ist also nur eingeschränkt *spatial explicit*). Dies erleichtert zwar die Interpretation der entstehenden Ergebnisse, jedoch ist es semantisch schwierig – insbesondere bei einer groben Einteilung mit geographisch großen Zellen – einer Zelle genau einen LUC-Typ zuzuweisen. Außerdem können bei dieser Vorgehensweise sehr grobe semantische Übergänge („harte Grenzen“) zwischen den Zellen entstehen (z.B. von *dichter Wald* zu *dichte städtische Bebauung*). Je feiner die Einteilungen (des Untersuchungsgebiets und/oder der LUC-Typen) vorgenommen werden (z.B. Zellen $< 1\ km^2$), desto unkritischer werden diese Nachteile und umso besser ist meist auch das Modell-Ergebnis. In der Regel können diese Parameter jedoch nicht beliebig gewählt werden, sondern sind durch die technischen Möglichkeiten der Datenerfassung (z.B. maximale Auflösung der Satelliten, Verfügbarkeit sozio-ökonomischer Informationen) und der Datenauswertung (Rechnerleistung) begrenzt. Es ist immer ein geeigneter Tradeoff zwischen Kosten und Nutzen einer feineren Auflösung zu finden.

Statt jeder Zelle j *genau einen* LUC-Typ zuzuweisen, modellieren viele Modelle das prozentuale Vorkommen für *jeden* betrachteten Landnutzungstyp i. Besonders für Zonen, in denen mehrere LUC-Typen relativ gleichverteilt auftreten, bietet diese Vorgehensweise Vorteile. Das prozentuale Vorkommen kann auch als Wahrscheinlichkeit dafür aufgefasst werden, dass die Landnutzung i in einer Zelle j vorkommt (*LUC-Wahrscheinlichkeit* P_{ij}).

1.3.2 Modellkonzepte

Ein Modell generiert seine Ausgabe in der Regel nicht auf einen Schlag, sondern Schritt-für-Schritt. Hierbei gibt es verschiedenste Ansätze, wobei nicht jedes Modell darauf abzielt eine konkrete LUC-Konfiguration zu erzeugen.
BRIASSOULIS identifiziert 8 Kategorien, in denen sich LUC-Modelle unterscheiden können [5, Kapitel 4.2]:

a) **Zweck, für den ein Modell erstellt wurde:**
z.B. Beschreibung realen Verhaltens; Auffinden von semantischen Zusammenhängen; Modellierung zukünftiger LUC-Konfigurationen

[4] auch *Landnutzungsklassen* genannt

b) **Zugrunde liegende Theorie:**
z.B. mikro- oder makro-ökonomisch

c) **Räumliche Auflösung (Aggregation):**
lokal, regional, interregional, national, global

d) **Grad der räumlichen Explizitheit:**
geo-referenziert (*full spatial explicit*) oder unabhängig vom genauen Ort

e) **Betrachtete LUC-Typen:**
z.B. urban/städtisch, agrarwirtschaftlich oder forstwirtschaftlich

f) **Aspekt der untersuchten LUC-Veränderung:**
z.B. Verstädtung, Abholzung, Austrocknung

g) **Zeit-Aspekt:**
statisch, quasi-statisch, dynamisch

h) **Modellierungstechnik:**
statisch, programmbasiert, *spatial interaction*, Simulation, *integrated model*

Eine eindeutige Zuordnung eines Modells in diese Kategorien ist jedoch nur schwer möglich. Dies stellt BRIASSOULIS sogar selbst fest [5, Kapitel 4.2].
Im Hinblick auf eine Implementierungsplattform für verschiedenste Modelle fällt den strukturellen Merkmalen a) *Zweck*, d) *Räumliche Explizitheit*, g) *Zeitaspekt* und h) *Modellierungstechnik* eine besondere Bedeutung zu. Im Folgenden möchte ich exemplarisch 4 Ansätze erläutern, die sich vor allem in diesen unterscheiden[5]:

Statistic Modelling
keine LUC-Konfiguration; kein Zeitablauf; eingeschränkte Georeferenz

Static Land Use Modelling
LUC-Konfiguration über allgemein gültige Regeln; Zeitablauf über Änderung *lokaler* Parameter; georeferenziert

Land Use Change Modelling (LUCC)
LUC-Konfiguration über allgemein gültige Regeln; Zeitablauf über Änderung *globaler* Parameter; georeferenziert

Agent-Based Modelling (ABM)
LUC-Konfiguration über Interaktion von Agenten; individuelles Verhalten der Agenten (keine allgemein gültigen Regeln); Modell-Verhalten stark dynamisch (vom aktuellen Modell-Zustand abhängig)

[5] Die angeführten Ansätze sollen *keine* sich ausschließenden Modell-Klassen darstellen.

Einfache statistische Modelle gehen von einer *gegebenen* LUC-Konfiguration aus und ermitteln signifikante Zusammenhänge (*Beta*s) zwischen der Landnutzung und den ebenfalls vorgegebenen Einflussfaktoren. Zum Beispiel liefert die *Multiple Regression* als Ergebnis den statistischen Einfluss (α_i und $\beta_{k,i}$) der *Driving Forces k* auf die jeweilige Landnutzung *i*. Hierzu wird folgende Bedingung gestellt [5, Kapitel 4.3]:

$$LUT_{i,j} = \alpha_i + \beta_{1,i} \cdot X_{1,j} + \ldots + \beta_{k,i} \cdot X_{k,j} + \ldots + \beta_{m,i} \cdot X_{m,j} + \varepsilon \qquad (1.1)$$

mit		
	i	ein LUC-Typ ($1 \leq i \leq n$)
	j	eine Zelle des Untersuchungsgebiets
	k	ein Einflussfaktor ($1 \leq k \leq m$)
	$LUT_{i,j}$	Menge/Landfläche in Zelle j, die für Typ i genutzt wird
	$X_{k,j}$	Ausprägung des Einflussfaktors k in Zelle j
	$\beta_{k,i}$	**Einfluss von Faktor *k* auf LUC *i*** (Regressionsparameter)
	α_i	Regressionskonstante für LUC i
	ε	Störterm

Statistische Modelle liefern keine neue Landnutzung und auch keine Begründung für die zugrunde liegende Landnutzung, sondern können lediglich Wechselbeziehungen zwischen der Landnutzung und den *Driving Forces* aufdecken. Sie dienen deshalb sehr häufig anderen Modellen als Eingabe (oder Teil-Modell). Ein simples, aber **statisches LUC-Modell** kann wie folgt aussehen:

- Zunächst werden die statistischen Regressionsparameter (β's und α's) über ein Prototyp-Untersuchungsgebiet (empirischer Beispieldatensatz) ermittelt.

- Anschließend kann für ein anderes Untersuchungsgebiet (empirische $X_{k,j}$) oder ein Zukunftsszenario (geschätzte $X_{k,j}$) eine konkrete LUC-Konfiguration berechnet werden (Formel 1.1 diesmal mit $LUT_{i,j}$ als Unbekannte).

Wenn man für mehrere (aufeinander folgende) Zeitpunkte die *Driving Forces* vorgibt, erhält man zwar ein zeitliches Modell, jedoch bleibt diese Art von Modellen sehr statisch, da die Veränderung der Landnutzung ausschließlich auf der Veränderung der *lokalen Driving Forces* basiert. Man kann hierbei also nicht wirklich von der Modellierung von *land use/cover change* (LUCC) sprechen, da die vorangegangene Landnutzung unberücksichtigt bleibt und immer eine komplett neue LUC-Konfiguration ermittelt wird. Ein weiterer Nachteil ist, dass eine lokale[6] Vorhersage, welche *Driving Forces* sich wie verändern, sehr schwierig, wenn nicht sogar unmöglich ist.

An diesem Punkt setzten **LUCC-Modelle** an. Sie berücksichtigen zwar auch, welche Landnutzung an welcher Stelle am geeignetsten ist[7], jedoch konzentrieren sie sich primär darauf, *an welchen Stellen* am ehesten ein Wechsel der LUC auftritt, wenn

[6] für jede Zelle des Untersuchungsgebiets!

[7] z.B. mit Hilfe eines Regressionsmodells, wie es zuvor erläutert wurde

sich bestimmte *globale* Faktoren verändern (z.B. Population, Klima, Bedarfsanforderungen). Die räumliche Umgebung der einzelnen Zellen (Nachbarzellen) spielt dabei eine wichtige Rolle. Die lokalen *Driving Forces* werden dagegen weitestgehend (über die Zeit) konstant gehalten[8]. Bekannte Vertreter der LUCC-Modelle sind das CLUE-Modell [37, 36, 6] und *Zellulare Automaten* [23], welche in Kapitel 2.3 und 2.2 näher beschrieben werden.

Ein weiterer Modellierungsansatz, der vor allem in den letzten Jahren immer mehr in den Vordergrund tritt, ist die **agentenbasierte LUCC-Modellierung** [20, 21]. Diese stellt keinen komplett neuen Ansatz dar, sondern baut auf den gängigen zellularen Ansätzen auf (z.B. *Zellulare Automaten* oder *Markov Modelle*), die dazu genutzt werden, räumliche Prozesse (z.B. Ausbreitungen oder Beeinflussungen) darzustellen. In der Realität wird die LUC-Entscheidung jedoch nicht nur von lokalen *Driving Forces* beeinflusst, sondern in nicht unerheblichem Maße auch durch das individuelle Verhalten von menschlichen Entscheidungsträgern, sog. *Agents*. Ein Agent kann z.B. einen einzelnen Farmer darstellen, ein ganzes Dorf oder auch eine Region. In rein zellularen Modellen resultiert aus identischen *Driving Forces* in der Regel identische Landnutzung. Durch die Hinzunahme individueller Agents wird dieser Determinismus aufgebrochen. Verschiedene Agents haben ein unterschiedliches Verhalten und unterschiedliche Präferenzen (z.B. Gewinnmaximierung oder lediglich Deckung des Eigenbedarfs) und können die lokalen *Driving Forces*, ihr internes Wissen und externe Informationen unterschiedlich interpretieren. Zudem kann die Entscheidungsfindung der Agents durch gegenseitige Interaktion beeinflusst werden.
Agenten-basierte LUCC-Modellierung bietet einen hohen Grad an Flexibilität und Modellierungsspielraum. Sie ist dennoch nicht unumstritten, da sie dadurch sehr komplex ist und stark von dem „einprogrammierten“ Agenten-Verhalten abhängt:

> „[...] when you design something you have direct (partial or total) control on the outcome, whereas when you analyze something that's “out there”, you can only hope that you guessed correctly.“ H.COUCLELIS, [20, Kap. 1.3]

1.4 Zusammenfassung

Das vorangegangene Kapitel hat aufgezeigt, warum LUC-Modellierung ein sehr aktuelles Thema ist (insbesondere für Entwicklungsländer), und dass sich im Laufe der Zeit sehr viele verschiedene Ansätze der Modellierung entwickelt haben. Ein festes Schema, wie Modelle aufgebaut sind oder vorgehen, gibt es nicht. Dies zeigt sich beispielsweise beim Vergleich statistik-basierter Modelle mit dem recht jungen Ansatz der agentenbasierten Modellierung. Auch eine Einteilung in Modell-Klassen gestaltet sich schwierig, da sich verschiedene Ansätze häufig überschneiden. Dies ist besonders im Hinblick auf das Ziel dieser Arbeit von Bedeutung, eine allgemeine Modellierungsplattform zu

[8] Dies ist ein Aspekt, der gerade bei langen Modellierungszeiträumen kritisch zu betrachten ist!

schaffen, in der *alle* Techniken eingesetzt werden können, auch solche, die über den Bereich der LUC-Modellierung hinaus gehen.

Kapitel 2

Landnutzungsmodellierung in der Praxis

Dieses Kapitel stellt kurz das IMPETUS-Projekt vor, um einen thematischen Hintergrund aufzuzeigen, auf den an vielen Stellen der Diplomarbeit Bezug genommen wird. Darüberhinaus werden einige Ansätze der LUC-Modellierung genauer beschrieben, indem als Beispiel zwei Modelle/Modellapplikationen vorgestellt werden, die von der RSRG für IMPETUS eingesetzt werden: MAPMODELS und CLUE.
Zum einen werden hierdurch die gängigen Arbeitsweisen moderner LUC-Modelle verdeutlicht, zum anderen aber auch die Probleme heraus kristallisiert, die den Einsatz bestehender Modellapplikationen erschweren und die Motivation für die Entwicklung einer flexiblen Modellierungsplattform bilden. Das CLUE-Modell wird zudem im späteren Verlauf der Arbeit für die Evaluation der Plattform herangezogen.

2.1 IMPETUS

IMPETUS steht für „**I**ntegratives **M**anagement **P**rojekt für einen **e**ffizienten und **t**ragfähigen **U**mgang mit **S**üßwasser in West-Afrika“ [12]. Es besteht aus mehreren Teilprojekten (siehe Abb. 2.1). Ziel ist es, ausgehend von der Untersuchung zweier Flußeinzugsgebiete in Benin (Ouémé) und Marokko (Draa), langfristige Veränderungen des hydrologischen Kreislaufs zu analysieren und daraus Zukunftsszenarien zu konstruieren. Dabei werden Querverbindungen zwischen verschiedensten Bereichen der Geo- und Sozialwissenschaften hergestellt (vergleiche Abb. 2.2). Der RSRG-Fachbereich 09 um Prof. Dr. Menz befasst sich dabei mit der Vegetationsdynamik in Benin/Westafrika (Teilprojekt A3). Benin zählt zu den ärmsten Ländern der Erde. Haupt-Exportprodukt ist die Baumwolle[1]. Ungefähr zwei Drittel der Bevölkerung arbeiten in der Land- und Forstwirtschaft. Durch ein hohes Bevölkerungswachstum (ca. 3,23%) verstärkt sich der Druck auf die Umwelt-Ressourcen. Der Industrie- und Dienstleitungssektor ist noch nicht in der Lage, außerhalb der Landwirtschaft genügend neue Arbeitsplätze bereitzustellen. Ökologische Probleme, wie der Rückgang des Waldes, die Überfischung in den

[1] Auf dem Weltmarkt spielt der beninische Anteil an der Baumwollproduktion jedoch nur eine untergeordnete Rolle.

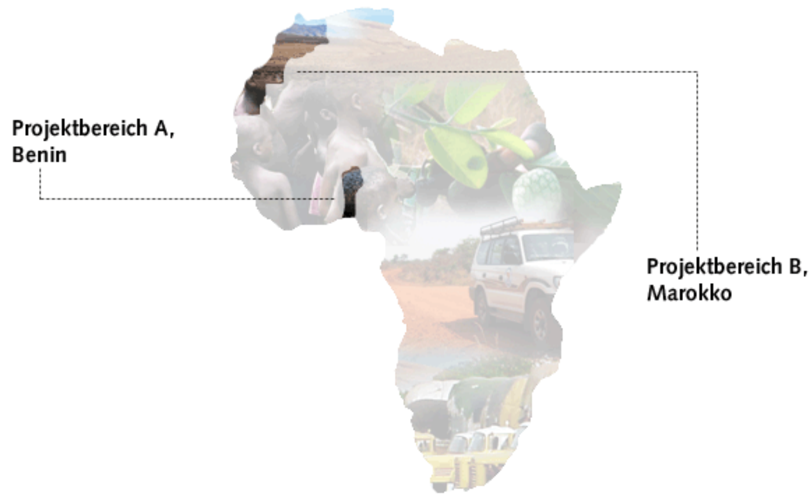


Abbildung 2.1: Übersicht über die IMPETUS Untersuchungsgebiete (Quelle: [12])

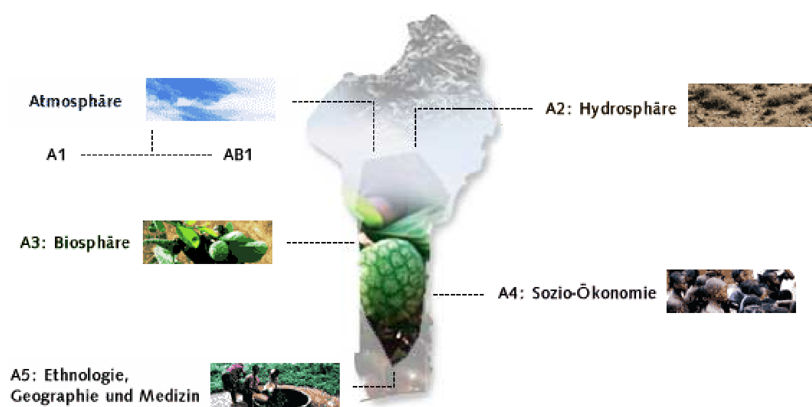


Abbildung 2.2: Das IMPETUS-A-Projekt in Benin (Quelle: [12])

Lagunengebieten, die Degradation der landwirtschaftlich nutzbaren Böden, die zunehmende Luftverschmutzung in den Städten und die Küstenerosion nehmen daher zu [2]. Zur Zeit untersucht die RSRG vorwiegend die Veränderung von Siedlungs-, Feld- und

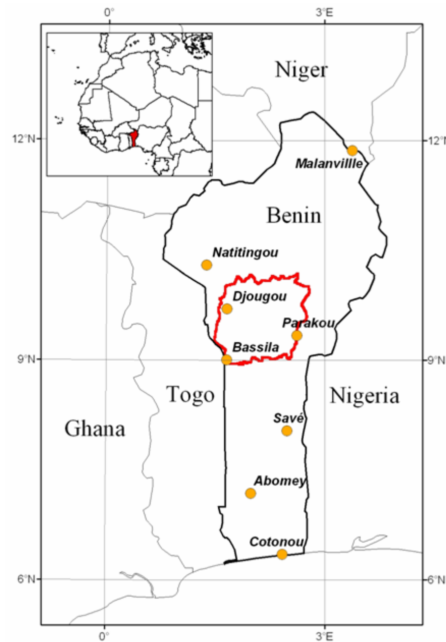


Abbildung 2.3: Das IMPETUS-Untersuchungsgebiet im Quémé-Einzugsgebiet von Benin (Quelle: [32])

Waldflächen im Einzugsbereich des Flusses Quémé (siehe Abb. 2.3). Die Modellierungsideen basieren dabei auf den Ergebnissen eigener Studien, die in den vergangenen Jahren zum großen Teil auch vor Ort durchgeführt wurden. Dabei wurde z.B. festgestellt, dass verschiedene Typen von Siedlungen (sog. *Altsiedel-* und *Neusiedeldörfer*) identifiziert werden können, die sich im Verhalten der Siedlungsexpansion und der Feld-Bewirtschaftung wesentlich unterscheiden [31, 30].
Die Daten für die Modellierung des Untersuchungsgebiets werden zu einem großen Teil aus Fernerkundungsdaten gewonnen (z.B. Klassifizierung der initialen LUC). Die Umgebung erfordert jedoch, dass gewisse Daten auch lokal erfasst werden. Als wichtiges Fortbewegungs- und Transportmittel werden von der Bevölkerung z.B. Fahrrad und Moped genutzt. Eine bedeutende Rolle spielen somit auch mitunter sehr kleine Pfade und Feldwege, welche über Fernerkundungsdaten jedoch nicht zu erkennen sind, so dass auf manuell erfasste Daten oder Luftaufnahmen aus geringer Höhe zurückgegriffen wird.

Um die LUC-Situation des Untersuchungsgebiets über die kommenden Jahre und Jahrzehnte zu simulieren, setzt die RSRG verschiedene Modell-Ansätze ein. Zwei dieser Modelle (statistik-basierend, Zellulare Automaten) werden anschließend in diesem Kapitel näher erläutert. Alle Techniken basieren jedoch auf global[2] gültigen Regeln. Unterschiedliche Dorf-Typen oder spezielle Regeln bei der Entstehung komplett neu-

[2] für das gesamte Untersuchungsgebiet

er Siedlungen, wie sie im Untersuchungsgebiet festgestellt wurden, können in den bestehenden Modellen (bzw. Modellapplikationen) nicht berücksichtigt werden. Zudem können die einzelnen Modell-Ansätze nur unabhängig voneinander eingesetzt werden.
Ziel der RSRG ist es jedoch, die gewonnenen Erkenntnisse zusammenhängend in einem gemeinsamen (neuen) Modell zu verbinden, um z.B. auch gegenseitige Wechselwirkungen berücksichtigen zu können. Hierfür fehlt bislang jedoch noch die geeignete Implementierungsplattform.

2.2 Zellulare Automaten mit MAPMODELS

MAPMODELS ist eine auf Flowcharts[3] basierende Programmiersprache zur Erstellung räumlicher Analysemodelle. Sie wurde durch die Gruppe um LEOPOLD RIEDL am Institut für Stadt- und Regionalforschung (SRF) der TU-Wien entwickelt und basiert auf dem ESRI-Programm ARCVIEW und der darin integrierten objektorientierten Programmiersprache AVENUE.

2.2.1 Die MAPMODELS-Anwendung

MAPMODELS ist eine Extension für die kommerzielle GIS-Anwendung ARCVIEW[4] [8]. Die Aufgaben der Datenverwaltung und -Visualisierung, sowie Daten-Import und -Export werden so komplett durch die übergeordnete Anwendung ARCVIEW bereitgestellt. Diese bietet darüberhinaus ein umfangreiches Funktionen-Sortiment zur Datenbearbeitung und räumlichen Analyse von Raster- und Vektordaten (z.B. Klassifizierung, Distanzberechnung, Umwandlung von Vektor- in Rasterdaten).
Die in ARCVIEW integrierte Programmiersprache AVENUE liefert für die ARCVIEW-Extensions eine Schnittstelle, um auf all diese Funktionalitäten zuzugreifen. So lassen sich umfangreiche Abläufe durch AVENUE-Code darstellen, abspeichern und als Makro beliebig oft (auch auf unterschiedlichen Daten) ausführen.

MAPMODELS wurde so konzipiert, dass ein Anwender, ohne explizite Kenntnisse einer (textuellen) Programmiersprache, in der Lage ist, Regeln für die räumliche Analyse zu erstellen. Die Benutzung von MAPMODELS erfordert lediglich Grundkenntnisse in analystisch-kartographischer Modellierung [26].
Regeln werden in MAPMODELS durch graphische Flowcharts ausgedrückt, welche einen gerichteten, azyklischen Graphen darstellen:

Knoten	$\mapsto$	Funktion, die auf die Daten angewandt wird
Kanten	$\mapsto$	Die Daten selbst
Kantenrichtung	$\mapsto$	Datenfluss

[3] auch „Datenflussgraphen“ genannt
[4] Bemerke: MAPMODELS funktioniert nur mit ARCVIEW 3.0 oder höher, *nicht* aber mit ARCGIS.

Abbildung 2.4 zeigt exemplarisch das Aussehen eines MAPMODELS-Flowchart[5]. Im Beispiel werden alle Zellen eines *digitalen Höhenmodells* (DHM) identifiziert, die oberhalb von 1000m liegen und deren Hangneigung 25% nicht überschreitet.

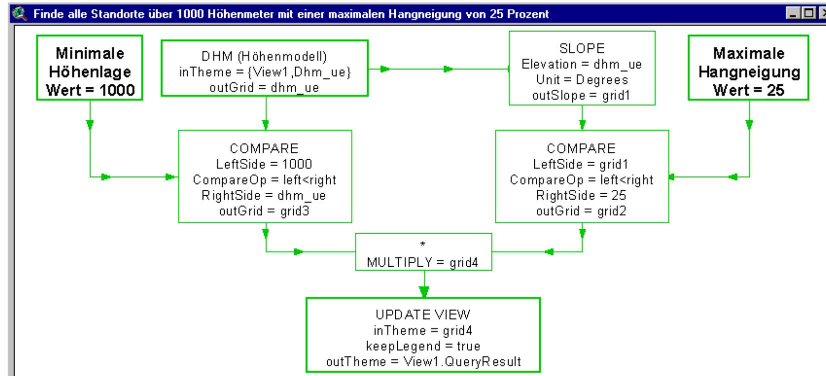
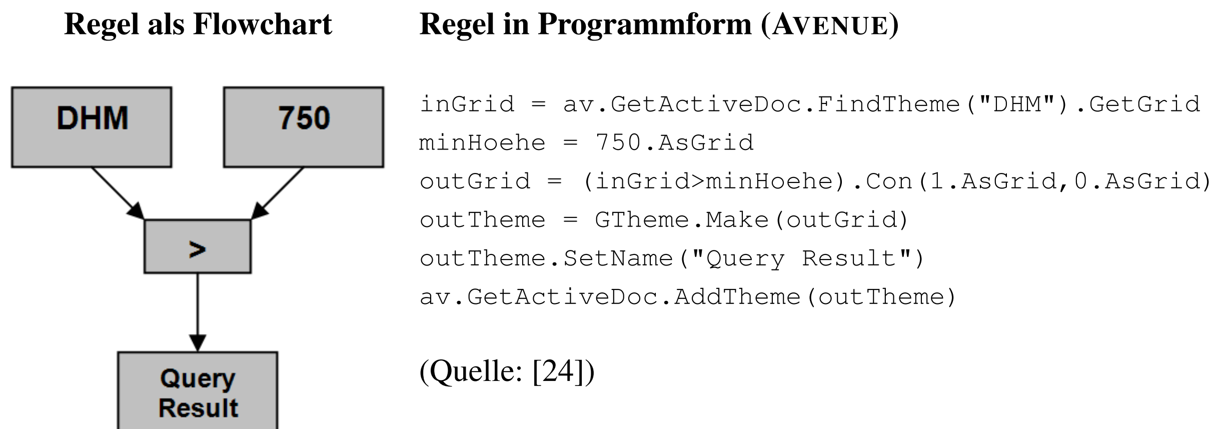


Abbildung 2.4: Ein MAPMODELS-Flowchart (Quelle: [24])

Die Flowcharts werden durch ein graphisches Interface per Drag&Drop zusammengesetzt. Die Steuerung der Operatoren – z.B. Ändern von Konstanten oder Vergleichsoperatoren – erfolgt ebenfalls über graphische Schnittstellen. Hierdurch entsteht für den Anwender eine wesentlich einfachere und bildhaftere Syntax, als dies bei einer textuellen Programmiersprache der Fall ist:



Regel als Flowchart | **Regel in Programmform (AVENUE)**

```
inGrid = av.GetActiveDoc.FindTheme("DHM").GetGrid
minHoehe = 750.AsGrid
outGrid = (inGrid>minHoehe).Con(1.AsGrid,0.AsGrid)
outTheme = GTheme.Make(outGrid)
outTheme.SetName("Query Result")
av.GetActiveDoc.AddTheme(outTheme)
```

(Quelle: [24])

Von der technischen Seite gesehen, setzt das MAPMODELS-Interface lediglich die Flowcharts in ausführbaren (kompilierbaren) AVENUE-Code um, in dem jedem Knoten ein festes Skript zugeordnet ist. In diesem variieren lediglich die Parameter für die ein- und auslaufenden Kanten. Theoretisch steht also die gesamte Mächtigkeit von ARCVIEW (AVENUE) für MAPMODELS zur Verfügung. In seiner Grundversion enthält MAPMODELS bereits eine Vielzahl von Funktionsknoten zur Geo-Analyse (z.B. `Reclassify`, `Local_Compare`, `If`, `FokalSum`, `Global_Statistics`, ...). In seiner

[5] Im folgenden bezeichne ich ein solches Flowchart auch kurz als „ein MapModel“

Funktionen-Struktur orientiert es sich dabei an der *MapAlgebra* von TOMLIN [24, 33]. Eine vollständige Liste der MAPMODELS-Funktionsbibliothek ist der MAPMODELS-Dokumentation zu entnehmen [22]. Der Anwender kann diese jedoch beliebig durch selbstprogrammierten AVENUE-Code erweitern [22].

2.2.2 Zellulare Automaten

Mit Hilfe der *Zellularen Automaten* (CA[6]) lassen sich eine Vielzahl von diskreten und zeitabhängigen Prozessen modellieren. Ein CA ist wie folgt charakterisiert [23]:

- Ein CA besteht aus einer regelmäßigen Anordnung von (nicht-überlappenden) Zellen. Häufig wird ein Raster verwendet.
- Jede Zelle ist durch einen Zustand z aus einer endlichen Zustandsmenge Z beschrieben.
- Die Zustandsänderung erfolgt in diskreten Zeitschritten $t = 1, 2, \ldots$
- Die Zustandsänderung $t \mapsto t + 1$ einer Zelle erfolgt unter Berücksichtigung der Zustände zum Zeitpunkt t der Zelle selbst, sowie einer definierten, endlichen Umgebung.
- Die Definition der Umgebung ist lokal (relativ zur betrachteten Zelle) und für alle Zellen gleich.

RIEDL beschreibt in [23] einige einfache Ansätze, wie MAPMODELS als CA eingesetzt werden kann. Diese Ansätze basieren jedoch alle auf einer „unbeschränkten" Simulation, d.h. jede Ausführung des MapModels entspricht genau einem Zeitschritt, und bei Hintereinanderausführung mehrerer Schritte, entwickelt sich das Szenario „unkontrolliert" weiter. Wie bereits in Abschnitt „LUCC-MODELLE" (1.3.2) erwähnt, verfolgen LUC-Modelle jedoch häufig die Strategie, globale Vorgaben für einen Zeitschritt zu setzen (z.B. bestimmte Bedarfe an LUC), um pro Zeitschritt ein Teilmodell solange iterieren zu lassen, bis die Vorgabe erfüllt ist. Das oben beschriebene CA-Konzept wird hierzu erweitert zu einem *Gezwungenen Zellularen Automaten* (vgl. [3]). Das Hauptaugenmerk liegt dabei nicht mehr auf der Frage „Was passiert?" (dies ist durch die globale Vorgabe ja bereits vorbestimmt), sondern wendet sich vielmehr der Frage zu „Wo äußert sich die Veränderung?".

Dieser Vorgehensweise folgt auch JANA BORGWARDT in ihrer Diplomarbeit für die RSRG [3], um die LUC-Entwickung in einem Teil des IMPETUS-Gebiets zwischen 1991 und 2010 zu simulieren. Für die Implementierung nutzt sie die beschriebene ARCVIEW-Extension MAPMODELS.
Bei dem betrachteten Untersuchungsgebiet handelt es sich um ca. die Hälfte eines

[6] CA = Cellular Automaton

60×30km großen Gebiets, welches in ein Raster mit Auflösung 28,5m eingeteilt ist ($\Rightarrow$ ca. 1,1Mio. Rasterzellen). Ausgangspunkt bildet eine LUC-Klassifikation für das Jahr 1991, welche 14 LUC-Typen umfasst (u.a. verschiedene Wald- und Savannenarten, Wasser, Siedlung dicht, Siedlung locker, Feld). In ihrem Modell bildet BORGWARDT jedoch nur den Wechsel in die LUC-Typen *Siedlung* und *Feld* ab. Ist einer Zelle dabei einmal der Typ *Siedlung* zugeordnet, so bleibt sie in der Folge unverändert. Das Modell sieht jährliche Zeitschritte vor, wobei der Bedarf (angestrebte Fläche) der beiden LUC-Typen *Siedlung* und *Feld* für jeden Zeitschritt extern vorgegeben wird. Er errechnet sich auf Basis von Erfahrungswerten und einem aus Vergangenheitswerten geschätzten jährlichen Bevölkerungswachstum von 7,7%:

$$Pop(t) = 107{,}7\% \cdot Pop(t-1) = Pop_{1991} \cdot 1{,}077^t \tag{2.1}$$

$$Bedarf_{siedlung}(t) = 107{,}7\% \cdot Fläche_{siedlung}(t-1) \tag{2.2}$$

$$Bedarf_{feld}(t) = 0{,}5ha \cdot Pop(t) \tag{2.3}$$

Abbildung 2.5 zeigt die gesamte von BORGWARDT entworfene Modellstruktur. Auf die Entstehung komplett neuer Siedlungen[7] „aus dem Nichts“ mit entsprechenden initialen Feldflächen, sowie die Betrachtung von Brachflächen, möchte ich im Folgenden nicht näher eingehen. Zwar werden sie auch durch MAPMODELS-Flowcharts realisiert, jedoch stehen sie in keinem unmittelbaren Zusammenhang zum Konzept der *Zellularen Automaten*.

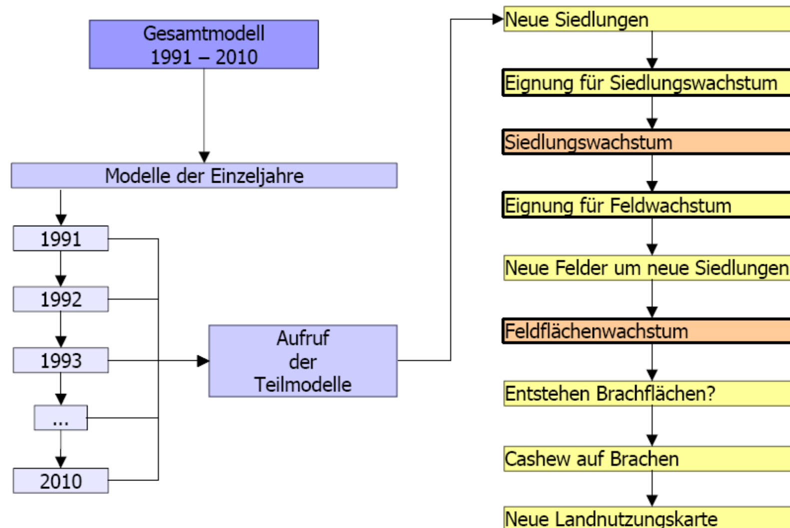


Abbildung 2.5: BORGWARDTs Modellstruktur zur Simulation von Siedlungs- und Feldflächen (Quelle: [3])

Die Expansion der Siedlungsflächen läuft für jedes simulierte Jahr folgendermaßen ab:

[7] an bestimmten Stellen entlang von Strassen

1. Über die *Driving Forces* ...

    - Hangneigung
    - Galeriewälder
    - Forét classée
    - Dorfgebietsgrenzen
    - Landwirtschaftliche Eignung
    - Distanz zur nächsten Strasse
    - Distanz zur nächsten Siedlung
    - Aktuelle Landnutzung

    ... wird die prinzipielle Eignung einer jeden Zelle für den LUC-Typ *Siedlung* bestimmt (als absoluter Wert). Dabei dienen die ersten vier *Driving Forces* als Ausschlussfaktoren (*Inhibitoren*), die weiteren 4 als Bewertungsfaktoren. Der überwiegende Teil der Parameter stellt Konstanten dar, die über den gesamten Modellablauf unverändert bleiben[8]. Beim Faktor „Distanz zur nächsten Siedlung“ verhält es sich anders. Aufgrund der sich im Modellablauf ändernden LUC (insb. der Siedlungen), wird dieser Parameter für jeden simulierten Zeitschritt neu berechnet.

2. Ausgehend von dem berechneten Eignungsraster findet die Siedlungs-Expansion nach dem Prinzip der *Gezwungenen Zellularen Automaten* statt:
    Für jede Zelle wird zunächst die sog. *Von-Neumann-Nachbarschaft* des Radiuses 1 betrachtet, welche die 4 direkt benachbarten Zellen (Nord, Ost, Süd und West) beschreibt[9]. Nur wenn mindestens eine der Nachbarzellen bereits für *Siedlung* genutzt wird, darf die betrachtete Zelle prinzipiell für die LUC *Siedlung* in Betracht gezogen werden. Hierbei kommen die fokalen Funktionen von MAPMODELS zum Einsatz (`NbrHood Circle` = „Neighborhood Circle“ und `Focal Statistics`), welche zusammen mit einem Vergleichsoperator (`Local Compare`) ein binäres Raster erzeugen. Eine logische UND-Verknüpfung mit dem Eignungsraster eliminiert die Eignungswerte aller Zellen, die nicht für *Siedlung* in Frage kommen. Anschließend werden von den verbleibenden Zellen – über einen Eignungs-Schwellwert[10] – die „geeignetsten“ identifiziert und in *Siedlung* umgewandelt.
    Aus technischen Gründen werden anschließend die bereits in vorherigen Schritten als *Siedlung* modellierten Zellen durch eine logische ODER-Verknüpfung nochmals mit in das Ergebisraster aufgenommen.

3. Die Gesamtfläche des Typs *Siedlung* wird mit dem Bedarf des betrachteten Jahres verglichen. Ist dieser noch nicht erreicht, wird mit Schritt 2 fortgefahren (Iteration). Andernfalls endet die Siedlungsexpansion an dieser Stelle.

[8] Sie können also bereits in einem Vorbereitungsschritt in Rasterform gebracht werden
[9] Ausnahme: Am Rand beschreibt die *Von-Neumann-Nachbarschaft* nur 2 oder 3 Zellen
[10] Der Schwellwert bleibt über den gesamten Modellablauf konstant

Das MAPMODELS-Flowchart für die Siedlungsexpansion (Schritt 2 und 3) ist in Abbildung 2.6 dargestellt. An dieser Stelle sei jedoch bereits darauf hingewiesen, dass die Iterationsschleife darin nicht explizit dargestellt ist, da sie nur über einen Umweg zu implementieren war. Im Abschnitt 2.2.3 wird näher auf diese Problematik eingegangen.

Die Modellierung der Feldflächen verläuft technisch sehr ähnlich zu der der Siedlungsflächen. Es werden jedoch etwas andere *Driving Forces* und eine größer definierte *Von-Neumann-Nachbarschaft* für den CA verwendet [3]. Die Frage der Konkurrenzsituation ...

> „In welchen LUC-Typ wird eine Zelle umgewandelt, wenn sie sowohl für *Siedlung* als auch für *Feld* geeignet ist?“

... wird automatisch durch den seriellen Modellablauf – erst die Expansion der Siedlungen und danach die der Felder (siehe Abb. 2.5) – gelöst. Zusammen mit der bereits erwähnten Tatsache, dass *Siedlung*-Zellen nicht mehr für eine Umwandlung in Betracht gezogen werden, entsteht eine Bevorzugung des LUC-Typs *Siedlung*. Diese ist nicht zufällig, sondern durchaus gewollt, da die Siedlungsentwicklung direkt die Bevölkerungsentwicklung widerspiegelt. Der Bedarf an *Feld*-Flächen ist erst eine Folge daraus und deshalb der Siedlungsexpansion unterzuordnen.

2.2.3 Bewertung von MAPMODELS für die CA-Implementierung

Als *Extension* von ARCVIEW kann MAPMODELS auf der darin enthaltenen Visualisierung und Datenverwaltung aufsetzen, was den Vorteil hat, dass dieser (umfangreichen) Thematik bei der (Weiter-) Entwicklung und Benutzung von MAPMODELS nahezu keine Beachtung geschenkt werden muss.
MAPMODELS wurde von L. RIEDL so konzipiert, dass ein Anwender ohne explizite Programmier- und AVENUE-Kenntnisse in der Lage ist, häufig benötigte ARCVIEW-Operationsfolgen über (parametrisierte) „Makros“ zusammenzufassen. Durch die übersichtliche Gestaltung der Flowcharts via Drag&Drop und den großen Pool vordefinierter (geo-analytischer) Funktionsknoten (siehe [22]), erfüllt MAPMODELS dieses Bestreben sehr gut. Darüberhinaus können erfahrene Anwender die Bibliotheken um neue Funktionsknoten erweitern und dabei den kompletten AVENUE-Sprachschatz ausschöpfen.
Wie RIEDL in [23] darstellt, lassen sich mit MAPMODELS einfache CA-Modelle implementieren, in dem ein MapModel wiederholt hintereinander ausgeführt wird und jeweils einen Teil seiner Input-Daten überschreibt. Zur Steuerung ist jedoch ein Anwender erforderlich, der den Makro-Aufruf für jeden simulierten Zeitschritt manuell vornimmt. Eine Automatisierung mittels Schleifen wird von MAPMODELS nicht unterstützt. Aus diesem Grund hat RIEDL für die Arbeit von JANA BORGWARDT eigens einen sog. *Iterator-Knoten* implementiert, mit dem eine automatische Wiederholung eines gesamten MapModels ermöglicht wird. Dieser ist jedoch auch nur eingeschränkt für die Iterationen eines *Gezwungenen Zellularen Automaten* geeignet, da es sich bei dem Iterator-Knoten vielmehr um eine FOR-Schleife handelt, für die die Anzahl an

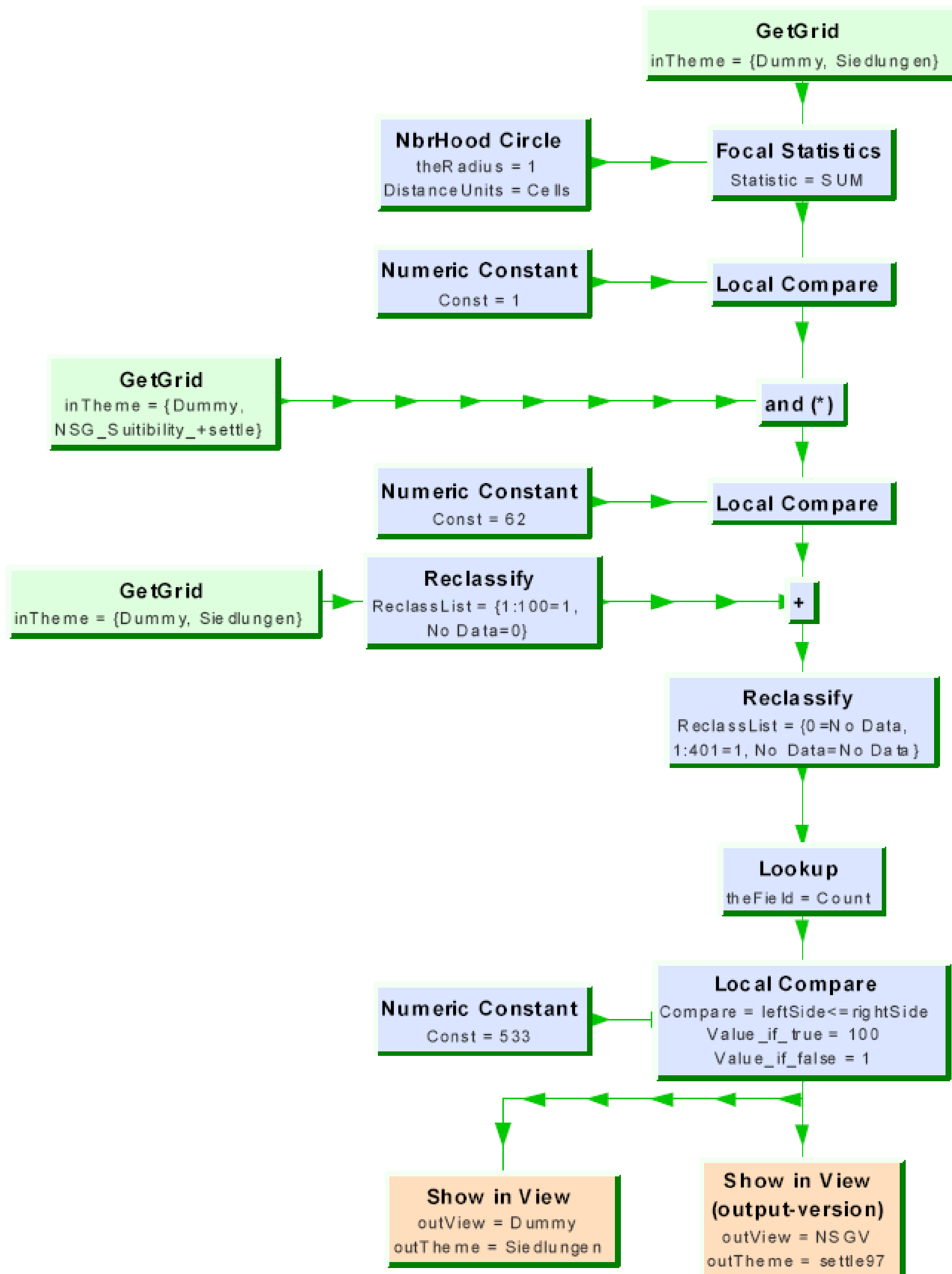


Abbildung 2.6: Das *MapModel* für eine Iteration der Siedlungsexpansion (Quelle: [3])

Durchläufen a priori festgelegt werden muss. Ein beliebiges Abbruchkriterium kann nicht formuliert werden. Deshalb behilft sich BORGWARDT bei der Siedlungsexpansion (und entsprechend bei der Feld-Expansion) mit einer Notlösung (siehe Abb. 2.6):
Die Iterationsschleife (FOR) wird hinreichend oft ausgeführt. Ist der geforderte Bedarf für die Landnutzung erreicht, wird allen aktuell für *Siedlung* genutzten Zellen ein spezieller Wert (`Reclassify` auf 100) zugewiesen. Entsprechend zeigt sich ein visueller Effekt, welcher dem Anwender als Zeichen dient, dass die Iterationsschleife abgebrochen und zum nächsten Modellschritt übergegangen werden kann.
Ein weiterer Schwachpunkt in diesem Zusammenhang ist, dass der Iterator-Knoten nicht mehr korrekt funktioniert, wenn das betreffende MapModel in einer MAPMODELS-Hierarchie (siehe [25]) eingebettet ist. Dies mag daran liegen, dass der Iterator-Knoten keinen ausgereiften Bestandteil von MAPMODELS darstellt [31].

Liegen Geo-Datensätze in Form numerischer Raster vor, so sind darin in der Regel immer auch Zellen vorhanden, denen *kein* Wert zugeordnet ist, sondern eine spezielle Kennzeichnung „NODATA". Dies ist z.B. der Fall, wenn für die Zelle kein Messwert vorliegt, oder die Zelle nicht zum Untersuchungsgebiet gehört. Alle numerischen MAPMODELS-Funktionen (z.B. die boolschen Operatoren oder mathematische Funktionen) können jedoch nicht mit NODATA-Werten umgehen, d.h. sie brechen mit einer Fehlermeldung ab. Deshalb ist vor der Anwendung eines solchen Funktionsknotens auf ein Raster fast immer eine `Reclassify`-Behandlung notwendig, um alle NODATA-Zellen des Rasters in einen passenden numerischen Wert (z.B. 0) umzuwandeln[11]. Dies ist nicht nur „störend" für den Anwender, sondern auch im hohen Maße ineffizient. Ein globales `Reclassify` für ein Raster mit 1000×1000 Zellen bedeutet, dass 1Mio. Zellen betrachtet werden müssen. Wünschenswert wäre es, wenn die numerischen MAPMODELS-Funktionen die NODATA-Zellen „on-the-fly" als einen bestimmten (einstellbaren) numerischen Wert behandeln würden. Hierdurch könnten viele unnötige Rasterdurchläufe vermieden werden.

Der vorangegangene Abschnitt zeigt, dass MAPMODELS nur sehr eingeschränkt für LUC- und LUCC-Modellierung geeignet ist. Einfache CAs können implementiert werden, sobald jedoch komplexere Abläufe für die Modellsemantik benötigt werden (z.B. individuelle Fallunterscheidungen oder Schleifen), stößt man schnell an die Grenzen vom MAPMODELS. Zudem sei abschließend angemerkt, dass MAPMODELS speziell auf das Programm ARCVIEW zugeschnitten ist und mit den neuen ESRI-Produkten ARCGIS 8 oder 9 nicht mehr funktioniert. Der Grund hierfür ist, dass die Programmiersprache AVENUE – und somit sämtliche MAPMODELS-Skripte – in ARCGIS nicht mehr unterstützt wird.

[11] Abschließend ein weiteres `Reclassify`, um diese Umbenennung rückgängig zu machen

2.3 CLUE

Das CLUE-Model[12] wurde an der WAGENINGEN UNIVERSITY (Niederlande) entwickelt und simuliert die räumliche Verteilung *gegebener* Landnutzungsanforderungen (Bedarfe) über einen bestimmten Zeitraum [6]. BRIASSOULIS ordnet CLUE in die Gruppe der *Simulation Integrated Models* ein, da es in der Lage ist, mehrere Landnutzungstypen und deren Konkurrieren simultan und dynamisch zu modellieren [5]. CLUE arbeitet komplett rasterbasiert und baut sehr stark auf der Projizierung statistisch ermittelter Regelmäßigkeiten auf. Von CLUE gibt es 2 Varianten (siehe Tabelle 2.1), welche durch die unterschiedliche Struktur und Auflösung der zugrunde liegenden Daten begründet sind.

CLUE	CLUE-S
• nationales und kontinentales Level	• lokales und regionales Level
• grobe Raster-Auflösung ($> 1km^2$)	• feinere Rasterauflösung ($< 1km^2$)
• Daten aus manueller Erfassung (z.B. Volkszählung)	• Daten aus Fernerkundung und Karten
→ aufgrund der groben Auflösung ist es nicht sinnvoll, jeder Zelle genau einen LUC-Typ zuzuordnen (Verzerrung)	→ feinere Auflösung erlaubt Zuordnung genau eines LUC-Typs zu jeder Zelle
→ die LUC jeder Zelle wird durch prozentuale Anteile repräsentiert, die für das Vorkommen jedes LUC-Typs in der Zelle stehen	→ uneindeutige Zuordnung lediglich in Übergangsbereichen; kann jedoch vernachlässigt werden, da auf Grund des Modellierungsverfahrens dort als erstes ein Wechsel stattfindet

Tabelle 2.1: Die zwei Varianten des CLUE-Modells

CLUE wurde bereits oftmals erfolgreich eingesetzt [37, 36]. Aus diesem Grund verwendet es MICHAEL JUDEX (RSRG) für das IMPETUS-Untersuchungsgebiet in Benin. Da dieses Einsatzgebiet dem regionalen Maßstab entspricht, wird in den folgenden Abschnitten ausschließlich auf die Arbeitsweise von CLUE-S eingegangen.
Wie bereits angedeutet, modelliert CLUE-S (wie auch CLUE) „lediglich“ die räumliche Ausbreitung (*Allocation*) vorgegebener Bedarfe, jedoch *nicht* die Bedarfsentwicklung selbst. Der Bedarf für jeden LUC-Typ muss für alle betrachteten Zeitpunkte von außen vorgegeben werden (siehe Abbildung 2.7). Für jeden modellierten Zeit-Schritt (in der Regel 1 Jahr) wird solange iteriert, bis der vorgegebene Bedarf für jeden LUC-Typ erreicht ist[13].

[12] CLUE – The **C**onversion of **L**and **U**se and its **E**ffects
[13] bzw. bis auf eine vorgegebene Fehlertoleranz

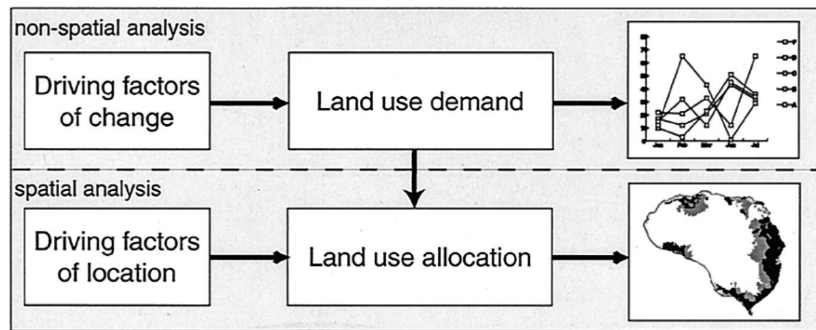


Abbildung 2.7: Überblick über die grobe CLUE-Struktur. Der obere Bereich läuft außerhalb der CLUE-Anwendung ab. (Quelle: [37])

2.3.1 Eingabedaten

CLUE-S ist ein offenes Modell, d.h. es ist nicht auf ein bestimmtes Szenario mit bestimmten LUC-Typen zugeschnitten, sondern es kann auf unterschiedlichste Untersuchungsgebiete angewendet werden. Dies macht eine recht aufwendige Konfiguration der CLUE-Anwendung erforderlich, welche komplett über Eingabe-Dateien stattfindet [39]. Neben einer Reihe von Konfigurationsparametern (Tabelle 2.2 zeigt einige wichtige), sind für den Modellablauf vor allem folgende Eingaben wichtig [39]:

a) ***Base Scenario*** (*numerisches* $x \times y$*-Raster*)
Beschreibt die LUC-Konfiguration, mit der die Modellierung startet. Ermittelt z.B. aus klassifizierten Fernerkundungsdaten.

b) ***LU Conversion Matrix*** (*numerische* $n \times n$*-Matrix*)
Beschreibt ob und wie ein direkter Wechsel zwischen den LUC-Typen erlaubt ist.

c) ***Conversion Elasticity*** (*numerischer* n*-Vektor*)
Beschreibt die Beständigkeit der LUC-Typen und stellt so eine weitere Einschränkung an den LUC-Wechsel dar.
$ELAS_U = 1$ bedeutet, dass wenn an einer Stelle eine U-Zelle „verschwindet“, an anderer Stelle im gleichen Zeitschritt keine Zelle vom Typ U entstehen darf.
$ELAS_U = 0$ bedeutet, dass uneingeschränkt an einer Stelle ein Wechsel von U nach A und gleichzeitig an anderer Stelle von B nach U möglich ist.
$0 < ELAS_U < 1$ drückt eine Tendenz zu einem der beiden Extremwerte 0 oder 1 aus.

d) ***Area Restrictions, Spatial policies*** (*numerisches* $x \times y$*-Raster*)
Beschreiben, an welchen Stellen des Untersuchungsgebiets ein LUC-Wechsel generell verboten ist (z.B. Schutzgebiete oder Bereiche, die nicht zum Untersuchungsgebiet gehören).

e) ***Demand Scenario*** (*numerische* $t \times n$*-Matrix*)
Beschreibt den LUC-Bedarf für jeden LUC-Typ zu jedem modellierten Zeitschritt.

n = Anzahl an LUC-Typen
m = Anzahl an *Driving Forces* (Einflussfaktoren)
t = Anzahl an simulierten Zeitschritten
x = Breite des Rasters (Untersuchungsgebiet)
y = Höhe des Rasters (Untersuchungsgebiet)

Tabelle 2.2: Einige wichtige Parameter des CLUE-Modells

f) ***Driving Forces*** (*m numerische* $x \times y$*-Raster*)
Beschreiben die Ausprägung der Einflussfaktoren an jeder Stelle des Untersuchungsgebiets. Diese bleiben über den gesamten Modellierungszeitraum konstant.

g) ***Regression Results*** (*numerische* $n \times (m+1)$*-Matrix*)
Beschreiben die Einfluss-Stärke der *Driving Forces* auf jeden einzelnen LUC-Typ (siehe unten).

Alle diese Angaben sind *vorab* vollständig durch den Anwender und/oder externe Programme bereitzustellen. Interessant sind dabei vor allem die *Regression Results*. Sie bestimmen im weiteren Modellablauf entscheidend den Prozess der LUC-Allokation. Wie bereits erwähnt, spiegeln die *Regression Results* den Einfluss der einzelnen *Driving Forces* auf die jeweiligen LUC-Typen wieder. Zusammen mit den Ausprägungen der *Driving Forces* ermittelt CLUE-S für jede Zelle i eine Wahrscheinlichkeit für das Vorkommen eines jeden LUC-Typs U. Diese berechnet es mit Hilfe des *Binary Logit Model* [37, 39]:

$$\ln\left(\frac{P_{i,U}}{1-P_{i,U}}\right) = \beta_{0,U} + \beta_{1,U} \cdot X_{1,i} + \ldots + \beta_{m,U} \cdot X_{m,i} \tag{2.4}$$

mit		
	U	einer von n LUC-Typen
	i	eine Zelle des Untersuchungsgebiets
	$P_{i,U}$	Wahrscheinlichkeit, dass Zelle i für den LUC-Typ U genutzt wird
	m	Anzahl an Einflussfaktoren (*Driving Forces*)
	$X_{k,i}$	Ausprägung des Einflussfaktors k in Zelle i
	$\beta_{0,U}$	Regressionskonstante für den LUC-Typ U
	$\beta_{k,U}$	Einfluss von Faktor k auf LUC-Typ U

Formel 2.4 stellt das Verhältnis von Wahrscheinlichkeit zur Gegenwahrscheinlichkeit dar (das sog. *odds ratio*). Nach ein paar Äquivalenz-Umformungen erhält man die absolute Wahrscheinlichkeit:

$$P_{i,U} = \frac{e^{\beta_{0,U} + \beta_{1,U} \cdot X_{1,i} + \ldots + \beta_{m,U} \cdot X_{m,i}}}{1 + e^{\beta_{0,U} + \beta_{1,U} \cdot X_{1,i} + \ldots + \beta_{m,U} \cdot X_{m,i}}} \tag{2.5}$$

Die *Betas* in dieser Gleichung stellen die *Regression Results* dar. Mit einer gegebenen (empirischen) LUC-Konfiguration und dazugehörenden Ausprägungen der *Driving Forces* (gegebene $P_{i,U}$[14] und $X_{k,i}$) können die *Betas* mit Hilfe eines Statistik-Programms (z.B. SPSS [29, 31]) berechnet werden [39].

[14] entweder 1 oder 0 je nachdem, ob Zelle i den Typ U enthält oder nicht

2.3.2 Modellablauf

Der erste Schritt der CLUE-Modellierung berechnet über die *Regression Results* und *Driving Forces* die LUC-Wahrscheinlichkeiten (Formel 2.5). Da die *Driving Forces* über den Modellablauf konstant bleiben, braucht diese Berechnung nur ein einziges Mal vor dem eigentlichen Modell-Start vorgenommen werden.
Für jedes zu simulierende Jahr t werden dann folgende Schritte durchgeführt:

1. Es werden alle Raster-Zellen identifiziert, für die ein LUC-Wechsel aufgrund der *Area Restrictions* oder *Conversion Elasticity* ausgeschlossen ist. Diese Zellen werden in den folgenden Schritten nicht mehr betrachtet.

2. Über die Formel
$$TPROB_{i,U} = P_{i,U} + ELAS_U + ITER_U \tag{2.6}$$
werden Gesamt-Wahrscheinlichkeiten für jede Zelle i des Untersuchungsgebiets errechnet. Die Iterationsvariablen $ITER_U$ sind initial für alle LUC-Typen U gleich und werden nach jedem Iterationsschritt entsprechend der Abweichung zum angestrebten Bedarf $DEMAND_{U,t+1}$ angepasst.

3. Jeder Zelle i wird der LUC-Typ U^* zugewiesen, der die größte Wahrscheinlichkeit
$$TPROB_{i,U^*} = \max_U(TPROB_{i,U}) \tag{2.7}$$
für die Zelle besitzt.

4. Wird der Bedarf des folgenden Jahres für einen LUC-Typ noch nicht erfüllt,
$$COVER_U < DEMAND_{U,t+1} \tag{2.8}$$
werden die Variablen $ITER_U$ angepasst und mit Schritt 2 fortgefahren (Iteration!). Ansonsten wird die LUC-Konfiguration für das betrachtete Jahr (in einer Datei) festgeschrieben und die Simulation des nächsten Jahres begonnen (Schritt 1).

Kern des CLUE-Modellablaufs bildet die Iteration, in der solange Wechsel in der LUC-Konfiguration vorgenommen werden, bis der Bedarf aller LUC-Typen für den nächsten Zeitschritt (bis auf eine Abweichnung) gedeckt ist. Der Wechsel der LUC-Typen hängt dabei nur von der jeweiligen Gesamt-Wahrscheinlichkeit $TPROB_{i,U}$ ab. Betrachtet man die Formel 2.6 genauer, so stellt man fest, dass $P_{i,U}$ und $ELAS_U$ Konstanten darstellen, die sich über den gesamten Modellablauf nicht ändern. Die einzige Variable, die die Iteration beeinflusst, ist also $ITER_U$.

$ITER_U$ wird in jeder Iteration um die (prozentuale) Abweichung der aktuellen Landnutzung $COVER_U$ vom Bedarfswert $DEMAND_{U,t+1}$ erhöht (bzw. verringert).

$$ITER_U = ITER_U + \frac{DEMAND_{U,t+1} - COVER_U}{COVER_U} + RAND \tag{2.9}$$

Den LUC-Typen, deren Bedarf noch nicht gedeckt ist, wird also sukzessive eine höhere Priorität (LUC-Wahrscheinlichkeit $TPROB_{i,U}$) gegenüber den anderen LUC-Typen zugeordnet.

2.3.3 Bewertung

Bei der Bewertung von CLUE, bzw. CLUE-S muss zwischen dem Modell selbst und der Modell-Anwendung unterschieden werden.
Der **Modellablauf** ist zwar sehr einfach gehalten (vgl. die 4 Schritte in 2.3.2), liefert aber trotzdem sehr passable Ergebnisse [31]. Durch die Kombination lokaler (statistischer) Regeln und globaler LUC-Bedarfe, stellt CLUE-S eine gute Verbindung zwischen Mikro- und Makro-Level her [5]. Außerdem ist das Modell trotz fest vorgegebenem Ablauf sehr flexibel in Bezug auf das LUC-Szenario (Untersuchungsgebiet, räumliche Auflösung und betrachtete LUC-Typen). Daneben hat der Anwender viele Möglichkeiten, die Eingabeparameter zu variieren. Er kann z.B. die LUC-Wechsel über die *LU-Conversion-Matrix* und die *Conversion Elasticity* steuern, sowie die Statistik-Methode zur Bestimmung der Regressionsparameter (*Betas*) frei wählen. Eine Einbeziehung frei definierbarer Regeln für die LUC-Wechsel – z.B. „Im Radius 1 um eine *Siedlung*-Zelle darf keine neue *Feld*-Zelle entstehen – ist hingegen nicht möglich. Zwar können in einer neueren CLUE-S-Version Nachbarschaftsbeziehungen berücksichtigt werden, jedoch nur eingeschränkt über statistische Regressionsparameter (analog zu den *Driving Forces*).
Ein weiterer Schwachpunkt im CLUE-S-Modell ist die (unrealistische) Annahme, dass die *Driving Forces* über den gesamten Modellierungszeitraum unverändert bleiben (konstante $X_{k,i}$). Die neuere CLUE-S-Version ermöglicht auch an dieser Stelle nur eine beschränkte Verbesserung: Es können *Dynamische Driving Forces* definiert werden, deren jeweilige Ausprägung für jeden Modellierungszeitpunkt angegeben wird, also $X_{k,i,t}$ statt $X_{k,i}$. Entsprechend werden auch die lokalen LUC-Wahrscheinlichkeiten $P_{i,U}$ nicht nur einmal, sondern für jedes modellierte Jahr neu berechnet. Durch die *Dynamischen Driving Forces* können Veränderungen des Untersuchungsgebiets einbezogen werden, die außerhalb des Modells begründet sind (z.B. Entwicklung von Infrastruktur durch Veränderung des Einflussfaktors „Distanz zur nächsten Straße“). Eine wirkliche Dynamik stellt dies jedoch nicht dar, da modellbedingte Veränderungen der *Driving Forces* – z.B. Verschlechterung der Bodenqualität bei bestimmter Landnutzung – hierdurch nicht simuliert werden können, obwohl diese einen wichtigen Einflussfaktor für die Veränderung der LUC darstellen.

Was auf Modell-Seite vorteilhaft erscheint, stellt sich jedoch auf der **Anwendungsseite** als nachteilig heraus. Die CLUE-S-Anwendung dient im weitesten Sinne lediglich der Modellsteuerung und bietet ansonsten keinerlei weitere Funktionalität (z.B. Visualisierung oder Statistik-Modul). Die Flexibilität in den Eingabeparametern bedeutet somit, dass der Anwender entsprechend viel Vorarbeit (außerhalb der CLUE-S-Anwendung) zu leisten hat, um zu diesen Parametern zu gelangen. Sehr aufwendig ist vor allem, den LUC-Bedarf für jeden LUC-Typ und jeden modellierten Zeitschritt zu schätzen. Dies kann durch externe, ökonomische Modelle geschehen oder einfach nur durch Expertenwissen. Ebenso sind die Regressionsparameter extern zu ermitteln und manuell über Konfigurtionsdateien zu pflegen. Die meisten Statistik-Programme (z.B. SPSS) sind jedoch nicht in der Lage, die Betas direkt aus GIS-Daten zu ermitteln. In der Regel ist eine vorherige Konvertierung notwendig. Wünschenswert wäre an die-

ser Stelle eine entsprechende Funktionalität direkt in der CLUE-S-Anwendung, die die *Binary Logit Regression* direkt aus GIS-Daten errechnet[15].
Darüberhinaus hat die CLUE-S-Anwendung einige kleine Schwachpunkte, die auf den ersten Blick zwar irrelevant erscheinen, die tägliche Praxisarbeit mit CLUE-S jedoch erschweren [31]. Zum Beispiel verwendet CLUE-S statische, unveränderbare Dateinamen und -pfade für die Ein/Ausgabe. Der Anwender muss zwischen 2 Modell-Läufen immer sämtliche Dateien in andere Verzeichnisse umkopieren, wenn diese nicht überschrieben werden sollen. Anwenderfreundlicher wäre, zumindest ein Arbeitsverzeichnis definieren zu können, welches für sämtliche Ein- und Ausgaben genutzt wird. Desweiteren bestehen die Konfigurationsdateien zum großen Teil aus sehr unübersichtlichen Listen von Zahlen, zwischen denen keinerlei Benutzer-Kommentare erlaubt sind.

2.4 Technische Probleme derzeitiger LUC-Modellierung

Der überwiegende Teil von LUC-Modellen ist als eigenständige und in sich geschlossene Applikation (Programm) implementiert. Die Entwicklung neuer, komplexerer Modellierungsideen gestaltet sich daher sehr problematisch, da die softwaretechnischen Grundlagen bisher immer eng mit der Modellimplementierung verknüpft sind. Jede Applikation implementiert ihre *Anwendungsfunktionen* neu:

- Datenbeschaffung[16]
- Datenverwaltung (häufig auch eigene Datenformate)
- Modellsteuerung
- Modellausgabe (z.B. direkte Visualisierung und/oder Dateiausgabe)

Das Austauschen einzelner Sub-Modelle durch andere Modellierungskomponenten ist nur selten möglich. Dies erschwert erheblich die Kombination von verschiedenen bestehenden Modell-Ansätzen in einem neuen Konzept. In den vorliegenden Beispielen wäre es z.B. wünschenswert, den Ansatz der *Zellularen Automaten* für die Siedlungs- und Feldexpansion mit den statistischen Methoden von CLUE-S und einem neuen Modul für die Entstehung komplett neuer Siedlungen zu verknüpfen.
Darüber hinaus ist es für Anwender auch wichtig, experimentelle Modellierungsideen einmal „schnell ausprobieren“ zu können. Dies scheitert aber daran, dass das entsprechende Modellierungsprogramm (samt Datenverwaltung, Visualisierung, usw.) immer von Grund auf neu implementiert werden muss [31].

[15] gleichzeitig aber auch die Verwendung externer Tools zulassen würde
[16] im Sinne von Daten-Import und -Export

2.5 Zusammenfassung

Nach einer thematischen Einführung in das Anwendungsgebiet, mit dem sich die RSRG im Rahmen von IMPETUS beschäftigt, hat das vorangegangene Kapitel zwei Modelle/Modellapplikationen vorgestellt, die die RSRG für das Untersuchungsgebiet in Benin/Westafrika einsetzt.

Auf der einen Seite steht CLUE-S mit einer mächtigen, aber festgelegten Modellalgorithmik und einer speziell darauf abgestimmten Anwendung, die außer der Modellkonfiguration und -steuerung keine weiteren Funktionalitäten bietet. CLUE-S enthält keine Möglichkeiten der Visualisierung oder Daten-Manipulation, ist in Bezug auf die modellierten LUC-Typen und das Untersuchungsgebiet jedoch sehr flexibel. Für den Anwender hat dies allerdings gleichzeitig einen hohen Aufwand an (Vor-) Konfiguration zur Folge. Auf der anderen Seite steht MAPMODELS, das durch die Trennung von Modellsemantik und Anwendungsfunktionen dem Anwender die Vorzüge und Mächtigkeit der kommerziellen Software ARCVIEW zur Verfügung stellt. Der Anwender kann sehr einfach eigene Modelle gestalten, die jedoch nicht sehr komplex sein können. Es können z.B. nur sehr einfache Zellulare Automaten integriert werden.

Obwohl beide dargestellten Modelle auch semantische Schwachstellen haben, zeigen die Arbeiten von BORGWARDT und JUDEX, dass sich mit beiden Modellierungsapplikationen passable Ergebnisse erzielen lassen [31]. Insbesondere gilt dies für CLUE-S. Eine direkte Kombination der beiden Modellansätze ist jedoch aufgrund der großen (technischen) Unterschiede nicht möglich. Dies ist ein generelles Problem, da die überwiegende Anzahl von Modellen durch komplett eigenständige Programme realisiert ist. Das Austauschen von Sub-Modellen ist nicht möglich, und zum Erstellen eines neuen Modells ist in der Regel die Implementierung eines komplett neuen Programms (samt Daten-Import und -verwaltung, Visualisierung, GUI, usw.) erforderlich.

Kapitel 3

Anforderungen an die flexible Modellierungsplattform XULU

Das vorangegangene Kapitel hat am Beispiel der zwei exemplarischen Modellierungsapplikationen MAPMODELS und CLUE die Probleme der derzeitigen Bestrebungen aufgezeigt, Landnutzung zu simulieren.

Die neue *eXtendable **U**nified **L**and **U**se Modelling Platform* (kurz: XULU)[1] sollte auf den Vorzügen beider Ansätze aufbauen und ein Framework bereitstellen, das zum einen den derzeitigen modellierungstechnischen Bedürfnissen genügt, zum anderen aber auch so flexibel gestaltet ist, dass es für zukünftige Anwendungsbereiche leicht erweitert werden kann. Die Anforderungen an die Modellierungsplattform XULU lassen sich dabei in zwei Klassen unterteilen:

1. Sicht des **Anwenders**, der *bestehende* Modelle (Modell-Implementierungen) verwendet, um daraus Ergebnisse zu generieren
2. Sicht des **Modell-Entwicklers**, der (neben dem Anwenden) auch neue Modelle *implementiert*

Darüberhinaus gibt es natürlich auch noch die Sicht des Anwendungsprogrammierers, der die „fertige" Modellierungsplattform warten und zukünftige Anwender-Anforderungen integrieren muss. Hierauf möchte ich an dieser Stelle jedoch nicht eingehen, da sich die Interessen des Programmierers – wie leichte Wartbarkeit und Erweiterbarkeit – in den Designentscheidungen widerspiegeln, welche Thema eines folgenden Abschnitts sind (Kapitel 4).

[1] **Bemerkung:**
Der Names-Teil „Land Use" ist lediglich durch das exemplarische Anwendungsgebiet begründet. XULU wird auch für andere Bereiche eingesetzt werden können (z.B. Wirtschaftssimulationen).

3.1 Anforderungen des Anwenders

Eine Grundanforderung des *Anwenders* besteht darin, dass die Modellierungsplattform eine einfach zu handhabende und vor allem grafische Oberfläche bereitstellt (nach Möglichkeit in verschiedenen Sprachen). Darüberhinaus sollen mehrere Modelle gleichzeitig verwaltet werden können, so dass die Ausgaben eines Modells unmittelbar für ein anderes Modell als Eingabe dienen können, ohne dass zunächst ein aufwändiger Datenaustausch vorgenommen werden muss (Export/Import). Dies ist (aus Anwendersicht) ein unangenehmer (weil zeitraubender) Schwachpunkt von CLUE. Zunächst muss mit einem Modell die Bevölkerungsentwicklung simuliert werden. Auf dieser Basis berechnet ein weiteres unabhängiges Modell die Bedarfsanforderungen für die einzelnen LUC-Typen, welche schließlich CLUE als Eingabe dienen. Alle Modelle sind unabhängig von einander, was zwischen den Modellen eine Konvertierung der Daten erfordert.

Diese grundlegenden Anwender-Wünsche führen automatisch zu einer zentralen Anforderung an die Modellierungsplattform XULU: Sie soll eine gemeinsame und einheitliche Datenverwaltung für alle Modelle bereitstellen, wie es z.B. bei MAPMODELS möglich ist (durch den Aufsatz auf ARCVIEW). Mit „einheitlich“ ist hierbei gemeint, dass alle Modelle auf den gleichen Datenformaten (Datentypen) operieren, so dass eine Konvertierung der Daten vermieden wird. Vor allem sollen die Daten variabel in die Plattform geladen und den Modellen (als Eingabe) zugeordnet werden können (im Gegensatz zu CLUE, welches statische Dateinamen in einem festen Verzeichnis voraussetzt[2]).

Mehrere Modelle gleichzeitig in der Plattform zu verwalten bedeutet zudem, dass entsprechend viele Daten zu verwalten sind. Hierbei ist es notwendig, dass diese gegenüber dem Anwender aussagekräftig repräsentiert werden. Bezeichnungen wie „Raster01“, „Raster02“, „Raster03“, usw. sind nichts-sagend und führen in der Praxis zu Verwirrung. Die Daten-Objekte sollen durch den Anwender auf Basis ihrer Semantik (Verwendungszweck) frei benannt werden können.

Auch wenn eine gemeinsame und einheitliche Datenverwaltung innerhalb der Plattform im Vordergrund steht, so ist dennoch der Datenaustausch mit externen Anwendungen nicht zu vernachlässigen. Um dem Anwender eine Vorbereitung und Nachbearbeitung der Modell-Daten zu ermöglichen (z.B. durch ARCVIEW), muss XULU den Daten-Import und -Export in gängigen Formaten – wie z.B. ARCVIEW-Grid, GEOTIFF oder Shape-Files – unterstützen. Darüberhinaus gewinnt die Anbindung an Datenbanken oder das Internet (WMS[3] oder WFS[4]) immer mehr an Bedeutung.

Externe Programme sollen jedoch nicht die einzige Möglichkeit der Datenmanipulation und -viualisierung darstellen. Modellierung stellt z.B. ein gängiges Werkzeug im Bereich *Decision Support* dar. Hierbei ist es wichtig, Veränderungen an den Modell-Daten vorzunehmen, um deren Auswirkungen auf das Modellergebnis zu analysieren.

[2] Soll in CLUE ein alternativer Datensatz verwendet werden, ist Umkopieren erforderlich!

[3] Web Map Server

[4] Web Feature Service

Im Bereich des IMPETUS-Projekts stellt sich zum Beispiel die Frage, wie sich das Vorhandensein und die Lage einer Straße auf die Ausbreitung von Dörfern, Siedlungen und Feldflächen auswirkt. Die Anpassung des entsprechenden Vektor-Datensatzes innerhalb von XULU ist für den Anwender wesentlich komfortabler (weil zeitsparender), als hierfür eine externe Anwendung heranziehen und die Daten nach jeder Änderung neu in XULU importieren zu müssen[5]. Gerade im Bereich des *Decision Support* müssen solche Daten-Anpassungen nämlich nicht nur einmal, sondern häufig vorgenommen werden. Ähnliches gilt für die Visualisierung von Daten. Für den Anwender ist es wichtig, diese direkt nach der Modellierung betrachten zu können, um zu evaluieren, ob das Modell plausible Ergebnisse geliefert hat. Beide Punkte – sowohl Visualisierung als auch Datenmanipulation – sind z.B. Schwachstellen von CLUE.

In diesem Zusammenhang ist es für den Anwender wünschenswert, die Modell-Daten auch während des Modellablaufs visuell beobachten zu können (Zwischenergebnisse), um die Arbeitsweise, das Verhalten und somit die Modellergebnisse besser nachvollziehen zu können. Da Zwischenergebnisse vom Modell (aus Speicherplatzgründen) in der Regel wieder überschrieben werden, kann eine solche Funktionalität besser durch eine in XULU integrierte Visualisierung bereitgestellt werden, als durch externe Anwendungen. Die Zeitpunkte, zu denen die Zwischenergebnisse betrachtet werden, sollten vom Anwender frei wählbar sein. Die Visualisierung sollte sich in bestimmten Fällen automatisch aktualisieren, wenn sich die zugrunde liegenden Daten ändern.

Abschließend ergibt sich noch eine weitere Aufgabe, die von der Modellierungsplattform XULU übernommen werden muss. Mitunter reicht es nicht aus, mehrere Modelle gleichzeitig in der Plattform zu verwalten, damit ein Modell auf den Ausgabe-Daten eines anderen Modells aufbaut. Statt dessen ist erforderlich, dass mehrere Modelle auch *parallel ablaufen* und gleichzeitig auf denselben Daten operieren. Die Modellierungsplattform muss in der Lage sein, diese Abläufe zu koordinieren, so dass permanente Datenkonsistenz gewährleistet ist.

A1)	Grafische Benutzeroberfläche
A2)	Mehrere (verschiedene) Modelle gleichzeitig und koordiniert
A3)	Gemeinsame Datenverwaltung
A4)	Gleiche Datentypen
A5)	Aussagekräfitige Datenrepräsentation
A6)	Daten-Import und -Export in gängigen Formaten
A7)	Neben Datei-Ein/Ausgabe auch Datenbank- und Internet-Anbindung
A8)	Integrierte Datenmanipulation
A9)	Integrierte Visualisierung
A10)	Automatisches Update der Visualisierung (steuerbar!)

Tabelle 3.1: Anforderungen des *Anwenders* an die Modellierungsplattform XULU

[5] Unter Umständen ist zusätzlich auch noch eine Konvertierung notwendig!

3.2 Anforderungen des Modell-Entwicklers

Neben den im vorangegangenen Abschnitt dargelegten Aspekten, ergeben sich aus Sicht des *Modell-Entwicklers* noch eine Reihe weiterer Anforderungen an die XULU-Modelling-Platform.

Zunächst einmal ist es für den Modell-Entwickler wichtig, dass in der XULU-Plattform eine Trennung zwischen Modell-Semantik und generellen Anwendungsfunktionen vorgenommen wird. XULU soll ein in sich geschlossenes (*stand alone*) und lauffähiges Programm (inkl. GUI) darstellen, welches zum einen die *Anwendungungsfunktionen* ...

- Datendefinition (allgemein gültige Datenstrukturen)
- Datenverwaltung und Datenbeschaffung (Import/Export)
- Datenmanipulation
- Visualisierung von Daten
- Controlling des Modellablaufs (Start, Stop, ...)

... vollständig zur Verfügung stellt, diese darüberhinaus aber auch verknüpft, steuert und kontrolliert. In diesen Plattformrahmen sollen dann verschiedenste Modelle dynamisch – also ohne Programmanpassungen an der Plattform – eingebettet werden können (Abb. 3.1). Dabei soll es möglich sein, unterschiedlichste Anwendungsgebiete abzu-

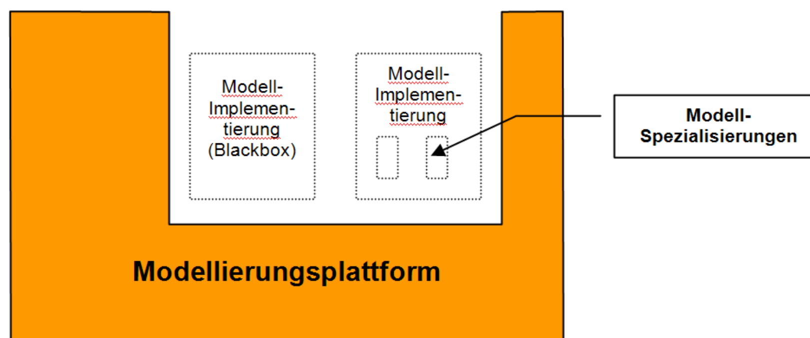


Abbildung 3.1: Die Modellierungsplattform trennt die modellspezifische Semantik von allgemeinen modellunabhängigen Funktionalitäten

decken, also nicht nur Landnutzungsmodellierung, sondern beispielsweise auch Wirtschaftssimulationen. Die zentrale Anforderung hierbei ist jedoch, dass sich die Integration eines neuen Modells im Wesentlichen auf die Implementierung der Modell-Algorithmik beschränkt. Deren Realisierung sollte auch für programmier-unerfahrene Modell-Entwickler möglich sein, also – ähnlich wie bei MAPMODELS – mittels Flowcharts und Drag&Drop. Die Programmierung aufwändiger grafischer Oberflächen (z.B.

für Steuerungsfunktionen Start/Stop/... oder Statusausgaben) soll in keinem Fall in den Aufgabenbereich des Modell-Designers fallen, sondern muss von der Plattform (generisch) realisiert werden. Hierdurch ist der Modell-Entwickler in der Lage, neue Ideen „auch einmal schnell auszuprobieren" [31]. In diesem Zusammenhang ist es auch wichtig, dass XULU dem Modell-Entwickler eine Reihe vielseitig einsetzbarer Datenstrukturen (Datentypen) zur Verfügung stellt und diese nicht für jedes Modell neu implementiert werden müssen (Tabelle 3.2). Um die Problematik der Speicherverwaltung soll sich der Modell-Entwickler nicht kümmern müssen. Dies ist von der Modellierungsplattform zu übernehmen.

Datentyp	**Einsatzgebiet**
Integer Double String Matrix Liste ...	vielseitig einsetzbar
2D-Raster Mehrdim. Raster Vektor-Daten Feature / FeatureCollection	Modelle mit Geo-Bezug

Tabelle 3.2: Beispiele für *allgemeine* XULU-Datentypen

Andererseits darf die Plattform nicht auf die Verwaltung dieser generischen Datentypen beschränkt sein. Es muss dem Modell-Entwickler offen stehen, für komplexere Modelle eigene Datentypen zu entwerfen (Tabelle 3.3). Ein Beispiel hierfür ist ein strukturierter Datentyp (Record) „CLUE-Parameter", der ganz bestimmte Steuerungsparameter für das CLUE-Modell enthält. Ein anderes Beispiel ist ein Agenten-Datentyp „Impetus-Altsiedeldorf", der einen Teil der IMPETUS-Modellsemantik enthält und in seinem Verhalten (z.B. Siedlungs- oder Feldexpansion) explizit auf das IMPETUS-Untersuchungsgebiet und die darin modellierten Landnutzungtypen ausgerichtet ist. Die in XULU integrierte Datenverwaltung und Visualisierung muss in der Lage sein, mit solchen Datentypen umzugehen.

Datentyp	**Einsatzgebiet**
Altsiedeldorf Neusiedeldorf ...	nur IMPETUS-Modell, da Verhalten speziell darauf ausgerichtet
Struktur mit Modell-Parametern	Speziell auf ein konkretes Modell zugeschnitten

Tabelle 3.3: Beispiele für *modellspezifische* XULU-Datentypen

Eine weitere Anforderung, die bei der Modell-Entwicklung eine Rolle spielt, ist die variable Kombination verschiedener, voneinander unabhängiger Modelle. So wie der Modell-Anwender Interesse daran hat, diese dynamisch und nach eigenem Wunsch in die Modellierungsplattform „zu laden" und (u.U. gleichzeitig) ablaufen zu lassen, bezweckt der Modell-Entwickler mitunter genau das Gegenteil. Ein Modell B soll auch als statische Sub-Routine in ein anderes Modell A eingebettet werden können. Dabei übernimmt nicht der Anwender die Kontolle über B, sondern das übergeordnete Modell A. Ein Beispiel hierfür wäre das von BORGWARDT entwickelte Modell zur Siedlungs- und Feld-Expansion, welches einerseits eigenständig ablaufen kann, andererseits aber auch innerhalb eines neuen IMPETUS-Modells einsetzbar sein soll (ohne es neu zu implementieren!).

E1)	Eigenständige und lauffähige Applikation (unabhängig vom Modell)
E2)	Modelle allein durch ihren Algorithmus integrierbar
E3)	GUI kein Bestandteil der Modell-Entwicklung
E4)	Modell-Ideen können „schnell" ausprobiert werden
E5)	Vielseitig einsetzbare Datentypen
E6)	Modelle auch intern verwendbar
E7)	Verschiedenste Modellarten (Anwendungsgebiete)
E8)	Interaktive Modell-Erstellung
E9)	Offene Datenverwaltung (auch modellspezifische Datentypen)

Tabelle 3.4: Anforderungen des *Modell-Entwicklers* an die Modellierungsplattform XULU

3.3 Zusammenfassung

Anwender und Modell-Entwickler haben unterschiedliche Anforderungen an die generische Modellierungsplattform XULU. Aus Entwickler-Sicht soll XULU vor allem eine eigenständige, lauffähige Applikation darstellen, die alle allgemeinen Anwendungsfunktionen – wie Datenverwaltung, Visualisierung und Modellsteuerung – samt GUI zur Verfügung stellt. In diese müssen unterschiedlichste Modelle *dynamisch* integrierbar sein. Die Implementierung eines neuen Modells soll sich auf die Modell-Algorithmik beschränken, nach Möglichkeit über eine interaktive Schnittstelle (Vorbild MAPMODELS). Technische Probleme (wie z.B. die Speicherverwaltung der Daten) sind durch XULU generisch zu lösen. Der Anwender hat zudem Interesse daran, mehrere Modelle gleichzeitig in XULU zu verwenden und für alle Modelle eine gemeinsame Datenbasis zu nutzen. Dabei möchte er auch Zwischenergebnisse der Modellierung (grafisch) betrachten. Vor allem im Rahmen des *Decision Support* sind Funktionen zur Datenmanipulation erforderlich. In jedem Fall muss jedoch auch ein Datenaustausch mit externen Programmen möglich sein (gängige Import/Export-Formate).

Kapitel 4

Entwurf von XULU

Abbildung 4.1 zeigt grob die Kernkonzepte, die beim Design der XULU-Modelling-Platform zum Einsatz kommen.

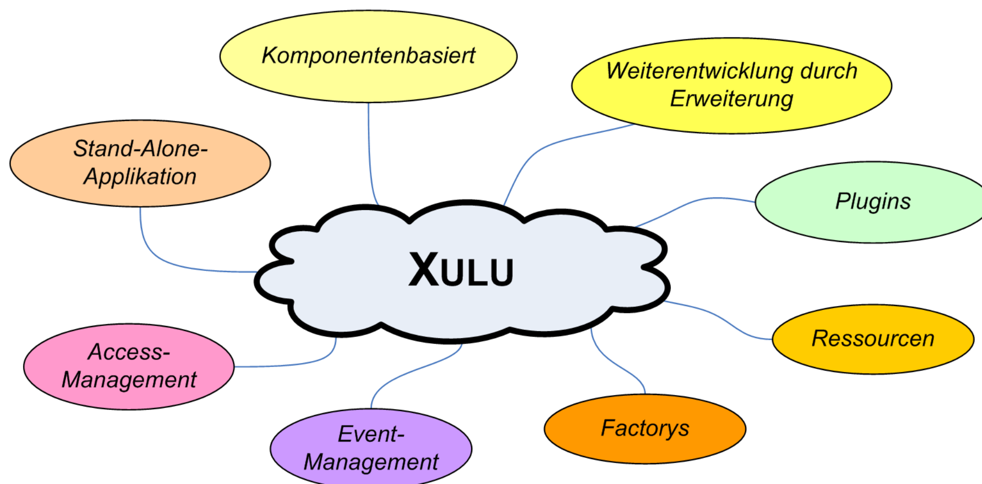


Abbildung 4.1: Die Kernkonzepte der Modellierungsplattform XULU

Hierbei spielen die *Prinzipien der Objektorientierung* (OO) eine grosse Rolle [7]:

- Kapselung und Modularität
- Dynamisches Binden
- Vererbung

Diese kommen an verschiedenen Stellen des Plattform-Entwurfs zum Einsatz und tauchen deshalb in den folgenden Abschnitten immer wieder auf. Sie sollen deshalb nicht an dieser Stelle, sondern anhand der konkreten Fall-Beispiele erläutert werden.

Eine erste zentrale Design-Entscheidung für XULU wird bereits durch die zentrale Modell-Entwickler-Anforderung (E1 in Tabelle 3.4) getroffen: XULU wird als *Stand-Alone-Applikation* realisiert, in die verschiedene Modelle dynamisch integriert werden können. Die Modellierungsplattform muss also lauffähig sein, ohne dass ihr

konkrete Informationen darüber vorliegen, welche Semantik die Modelle besitzen, die gerade in ihr ausgeführt werden.
Alternativ zu diesem *Stand-Alone*-Konzept wäre auch eine Bibliothek von Einzel-Komponenten denkbar, die jeweils eine einzelne der unter 3.2 aufgeführten Anwendungsfunktionen (mit entsprechenden GUI-Komponenten, A1) realisieren. Aus diesen könnte der Modell-Entwickler dann eine auf „sein Modell" zugeschnittene Applikation zusammensetzen (Baukasten-System). Dies steht jedoch im Gegensatz zu den Anforderungen A2, E2 und E3 gleichzeitig *mehrere*, *verschiedene* und *unabhängige* Modelle in der Plattform verwalten zu können und im Rahmen der Modell-Entwicklung nur die Algorithmik (und keinerlei GUI) berücksichtigen zu müssen.

Darüberhinaus möchte ich an dieser Stelle eine weitere wesentliche Design-Entscheidung anführen, die hinsichtlich einer Implementierung von XULU getroffen werden muss. Sie besteht darin, dass XULU-Modelle *nicht* durch eine interaktive Anwenderschnittstelle erstellt werden (E8), sondern aus fest (in JAVA) programmierten Modulen bestehen. Diese Einschränkung muss vorgenommen werden, da die Implementierung eines MAPMODELS-ähnlichen Modell-Designers den Rahmen dieser Diplomarbeit sprengen würde. Aspekte der Implementierung sollten zwar beim Entwurf eine untergeordnete Rolle spielen, dieser Punkt ist jedoch zu zentral, um beim Design der Plattform vernachlässigt zu werden. Der Entwurf und die Implementierung eines benutzerfreundlichen Modell-Designers kann Aufgabe weiterführender Projekte sein. Aufgrund des komponentenbasierten Aufbaus von XULU (vgl. 4.1) gestaltet sich die Integration eines solchen Moduls in die bestehende Modellierungsplattform als unproblematisch. Darüberhinaus wird die Evaluation (Kapitel 6) zeigen, dass XULU auch ohne eine solche Funktionalität bereits dazu genutzt werden kann, neue Modellideen ohne großen Aufwand umzusetzen (E4).

4.1 Übersicht über den Aufbau von XULU

Auch wenn das Design von XULU – aus Sicht des Anwenders und Modell-Entwicklers – einer *Stand-Alone*-Applikation entspricht (*Blackbox*), so ist es aus dem Blickwinkel des Programm-Entwicklers dennoch sinnvoll, die Modellierungsplattform komponentenbasiert aufzubauen und die Aufgabenbereiche der einzelnen Komponenten (z.B. Datenverwaltung, Visualisierung, usw.) klar abzugrenzen. Dies verbessert entscheidend die Wartbarkeit der Plattform. Voneinander unabhängige Komponenten, die nur über fest definierte Schnittstellen miteinander agieren, können durch *zentrale* Anpassungen (und somit mit begrenztem Aufwand) intern verändert oder sogar komplett ausgetauscht werden, ohne dass andere Komponenten in ihren Abläufen davon betroffen sind (⇒ OO: Kapselung und Modularität).

Die schematische Abbildung 4.2 zeigt die Erweiterung der unter Abschnitt 3.2 geforderten Abgrenzung (zwischen Modell und Plattform) auf die weiteren XULU-Module[1]. Die folgende Auflistung soll eine kurze Übersicht über die (bis auf die Daten-

[1] Der Übersicht halber zeigt Abbildung 4.2 nicht alle, sondern nur einzelne Plattform-Komponenten!

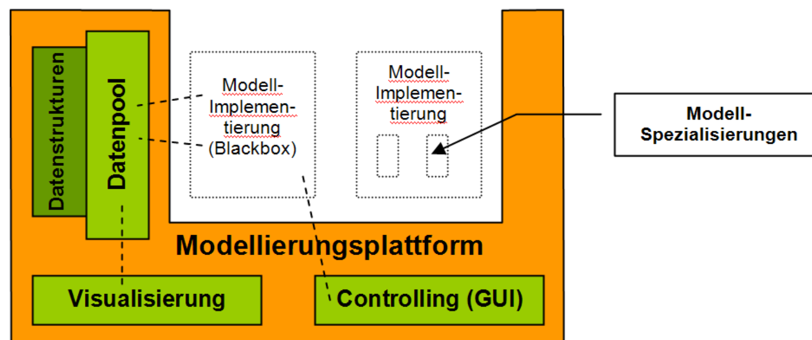


Abbildung 4.2: Eine (auch intern) komponentenbasiert aufgebaute Modellierungsplattform ermöglicht zentrale Erweiterung und den unabhängigen Austausch einzelner Komponenten.

typen internen) Plattform-Komponenten geben. Die Details werden in den kommenden Abschnitten näher erläutert:

- **Datentypen**
  Gemeinsame Datenstrukturen für alle Modelle
- **Datenpool**
  Übernimmt die *gemeinsame* Datenverwaltung für alle Modelle und den Datenaustausch (Import/Export)
- **Visualisierungs-Manager**
  Verwaltet unterschiedliche Visualisierungstools
- **Modell-Manager**
  Verwaltet und steuert die dynamisch in XULU geladenen Modelle und koordiniert deren Abläufe untereinander (Synchronisation)
- **Event-Manager**
  Nimmt Ereignisse der einzelnen Komponenten entgegen und reagiert darauf
- **Registry**
  Verwaltet zentral sämtliche in die Plattform integrierten Plugins

Das flexible Zusammenspiel verschiedener Modelle, eine einfache und unkomplizierte Modell-Implementierung, sowie der Datenaustausch mit externen Programmen[2] muss immer in Kombinationen mit Daten-Objekten betrachtet werden, und hängt somit sehr stark vom Design der Datentypen ab. Neben der Schnittstelle zu den Modellen (Modell-Manager), gehören somit die Datentypen und der Datenpool zu den wichtigsten Bestandteilen der Modellierungsplattform.
In den folgenden Abschnitten wird jedoch bewusst zunächst auf den Entwurf der restlichen Komponenten eingegangen, da sich hierbei diverse Aspekte heraus kristallisieren, die das Design der Datentyp-Schnittstelle entscheidend beeinflussen.

[2] Wichtige Anforderungen an die Modellierungsplattform (A3, A4, A6, E4, E5)!

4.2 XULU-Registry: Flexibilität durch dynamische Plugins

Eine zentrale Anforderung an die Modellierungsplattform ist, sie flexibel für viele Anwendungsgebiete einsetzen zu können (A2, E7). Hierdurch ist sie einem stetigen Weiterentwicklungsprozess ausgesetzt. Dabei ist es wichtig, dass bestehende Einsatzgebiete nicht negativ beeinflusst werden. Um dies zu gewährleisten, sollte nach dem Prinzip ***Weiterentwicklung durch Erweiterung*** und nicht *Weiterentwicklung durch Abändern* vorgegangen werden. Ein gutes Beispiel für diese Thematik bildet die Visualisierung. Unterschiedliche Anwendungsgebiete erfordern mitunter verschiedene Arten der Visualisierung (z.B. Karten als Geo-Visualisierung oder Diagramme für statistische Daten). Wird eine alternative Darstellunsart benötigt, so sollte nicht die bestehende Visualisierung *abgeändert*, sondern die Plattform um einen zusätzliches Visualisierungstool *erweitert* werden. Somit bleiben andere Anwendungsgebiete unbeeinflusst. Gleiches gilt z.B. auch für Datentypen, Import- und Export-Routinen oder Strategien der Modell-Synchronisation.

Wie das obige Beispiel zeigt, beschränkt sich die Anwendungssemantik (z.B. Landnutzungsmodellierung oder Wirtschaftsimulation) nicht nur auf die Modelle, sondern beeinflusst auch weite Teile des Plattform-Rahmens. Da dieser also einer stetigen Entwicklung ausgesetzt ist, erweist es sich als unpraktikabel, die Modellierungsplattform mit jedem Evolutionsschritt (z.B. neue Visualisierungsform) programmtechnisch anzupassen.

Eine weitere zentrale Design-Entscheidung besteht deshalb darin, die Modellierungsplattform XULU in zwei Bereiche zu gliedern:

Applikation und *Plugin*

Abbildung 4.3 zeigt, welche XULU-Komponenten als unabhängige Plugins realisiert werden. Die Applikationsseite ist statisch implementiert und stellt einen funktionalen Rahmen dar, der vor allem verwaltende Aufgaben übernimmt. In diesem Rahmen können verschiedene Arten von Plugins dynamisch – also ohne programmtechnische Anpassungen der Applikationsseite – integriert werden. Der Anwender kann zur Laufzeit selbst entscheiden, welche Plugins (z.B. welches Visualisierungstool oder welchen Datentyp) er verwenden möchte.

Ein wichtiges Hilfsmittel bei der Realisierung von Plugins ist das OO-Prinzip des *dynamischen Bindens*. Objekt-Eigenschaften und Objekt-Verhalten (in diesem Fall des Plugins) werden durch ein festes *Interface* spezifiziert und im Programmcode nur auf Basis dieses Interfaces gearbeitet. Welches konkrete Verhalten sich hinter dem Methodenaufruf einer Objekt-Variablen verbirgt, steht zum Zeitpunkt der Kompilierung noch nicht fest, sondern hängt davon ab, welche Implementierung des Interfaces zur Laufzeit instanziiert und der Objekt-Variablen zugewiesen wird. Die Methode wird zur Laufzeit dynamisch an das Objekt *gebunden* (Abb. 4.4).

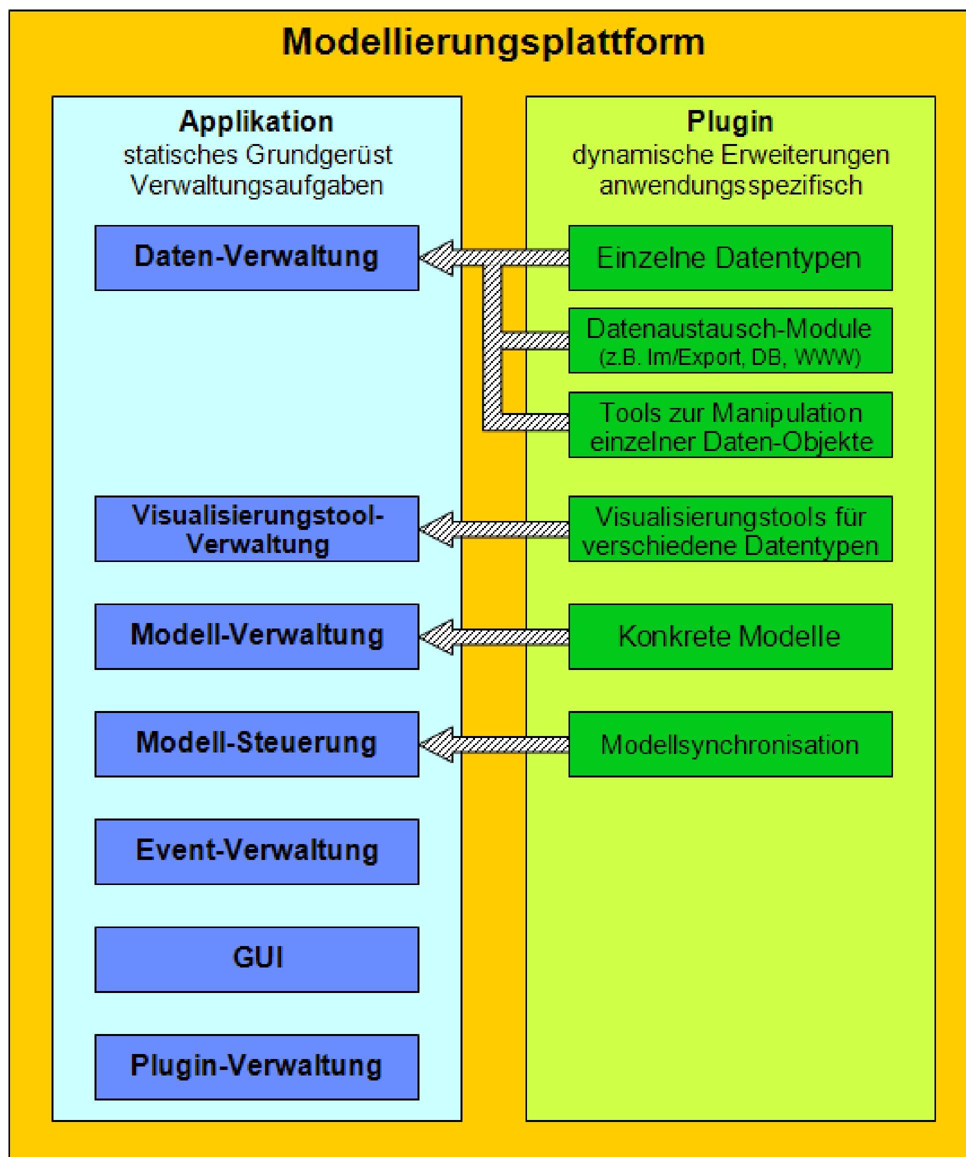


Abbildung 4.3: Die zwei Teile der Modellierungsplattform: *Anwendung* und *Plugins*.

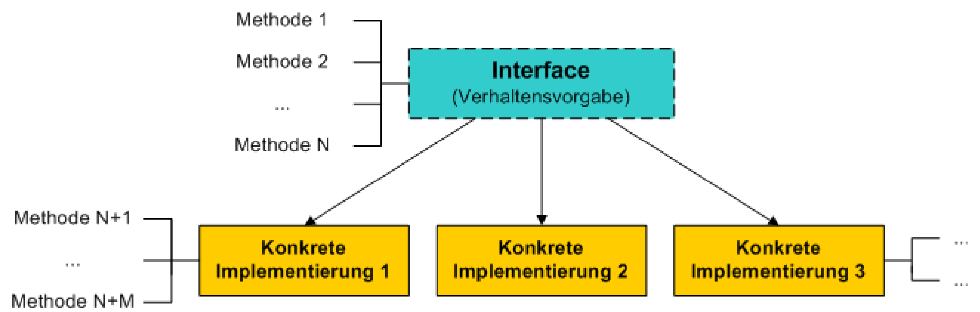


Abbildung 4.4: *Dynamisches Binden*: Im Programmcode wird nur mit den durch das Interface spezifizierten Methoden (1 bis N) gearbeitet, ohne Kenntnis von der konkreten Implementierung.

Die Applikationsseite alleine stellt bereits ein eigenständiges und lauffähiges Programm dar. Jedoch wird dieses erst durch die Entwicklung und Integration diverser Plugins (insbesondere Datentypen und Modelle) zu einer funktionstüchtigen Modellierungsumgebung für verschiedenste Anwendungsbereiche.

Aus Gründen der Übersichtlichkeit, Wartbarkeit und Erweiterbarkeit ist es *nicht* sinnvoll, dass jede XULU-Komponente selbst für die Integration ihrer Plugins verantwortlich ist. Besser ist die Verwaltung aller Plugins in einem zentralen Modul. Hierzu dient die XULU-*Registry*. Sie wird zum Programmstart über eine Konfigurationsdatei initialisiert und prüft somit *vorab*, ob alle angegebenen Plugins den benötigten Schnittstellen gerecht werden.
Wird im weiteren Verlauf eine konkrete Plugin-Instanz (z.B. ein Visualisierungstool) von einer der XULU-Komponenten benötigt, fordert sie diese bei der Registry an. Die Registry übernimmt dessen Erzeugung, wobei zu diesem Zeitpunkt gewährleistet ist, dass das Plugin (zumindest von seiner Struktur her) funktionstüchtig ist.
Darüberhinaus gibt die XULU-Registry Auskunft darüber, welche Plugins dem Anwender zur Verfügung stehen. Beispielsweise gibt sie dem Datenpool bekannt, welche Datentypen, Import- und Export-Routinen und Visualisierungstools der Anwender verwenden kann oder dem Modell-Manager, welche Modelle geladen werden können.

4.3 XULU-Event-Manager: Flexibilität durch den Anwender

Das *Event-Konzept* der XULU-Modelling-Platform entspringt der Anwender-Anforderung, zu „beliebigen“ Zeitpunkten, Zwischenergebnisse des Modellablaufs (visuell) beobachten zu können (A10).

Der direkte Weg, dieser Anforderung nachzukommen, besteht darin, die Visualisierung eines Objekts automatisch dann zu aktualisieren, wenn das Objekt seinen Zustand ändert. Dies kann jedoch schnell „zum Stillstand“ der kompletten Modellierungsplattform führen. Man braucht sich nur ein Szenario vorstellen, in dem der Anwender viele grosse Raster gleichzeitig visualisiert hat und ein Modell verwendet, das diese Raster sukzessive und häufig verändert. Würde die Veränderung einer einzigen Rasterzelle automatisch ein Visualisierungsupdate nach sich ziehen, wäre die Rechnerkapazität schnell erschöpft.

Für viele Anwender würde es wahrscheinlich ausreichen, wenn die Visualisierung am Ende eines jeden Modellschritts aktualisiert würde. Die Steuerung eines solchen Updates über die Modell-Algorithmik, ist jedoch nicht sinnvoll. Denn ein anderer Anwender wünscht sich das Update vielleicht bereits an bestimmten Stellen *innerhalb* eines Zeitschritts (z.B. am Ende einer Iteration), und für einen dritten ist das Zwischenergebnis vollkommen irrelevant, da er nur an einem *schnellen* Modell-Ablauf und dem Endergebnis interessiert ist. In beiden Fällen müsste der *Anwender* zunächst das Modell „umprogrammieren“ (und somit zum *Entwickler* werden!), bevor er es nach seinen jeweiligen Bedürfnissen verwenden kann.
Die XULU-Modelling-Platform sieht deshalb vor, dass die einzelnen Plattform-Komponenten (z.B. Daten-Objekte oder Modelle) Ereignisse initiieren und die Kontrolle über diese Ereignisse in der Hand des XULU-*Event-Managers* liegt. Der Event-Manager koppelt sich hierzu (als *Listener*) an die Komponenten und reagiert auf die Ereignisse (z.B. „Modellschritt beendet“) mit einem automatischen *Handler*-Aufruf. Neben dem Update der Visualisierung können so auch ganz andere Vorgänge über den Event-Manager automatisiert werden[3]. Eine Reihe von wichtigen *Event-Handlern* sind fest im Event-Manager integriert, z.B.:

- Wird ein Objekt aus dem Datenpool entfernt, wird es aus allen Visualisierungen entfernt. Anschließend werden alle mit dem Objekt verknüpften Ereignisse aus dem Event-Manager entfernt.
- Wird ein Visualisierungstool geschlossen, werden alle damit verknüpften Ereignisse aus dem Event-Manager entfernt.

Hierdurch wird ein großer Teil der Inter-Komponenten-Kommunikation über eine zentrale Stelle gesteuert. Statt einem verzweigten Netz von Zusammenhängen entsteht im Idealfall eine sternförmige Anordnung (Abbildung 4.5). Dies erhöht die Übersichtlichkeit und die Wartbarkeit der Plattform.

[3] In diesem Bereich ist sehr gut eine schrittweise Erweiterung von XULU möglich.

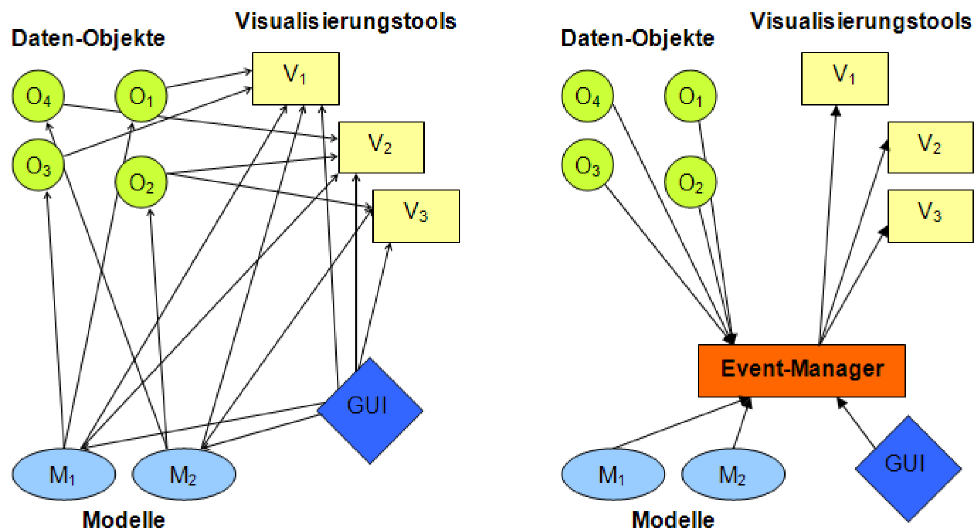


Abbildung 4.5: Eine zentrale Ereignis-Steuerung erhöht wesentlich die Übersichtlichkeit der internen Abläufe.

Neben den fest definierten Ereignis-Routinen besteht für den Anwender die Möglichkeit, über ein User Interface (UI) des Event-Managers, eigene Event-Handler zu spezifizieren. Hierzu bietet der Event-Manager dem Anwender eine Reihe vordefinierter Ereignisse an, denen der Anwender (ebenfalls vordefinierte) Aktionen zuordnen kann. Eine Reihe von Möglichkeiten ist in Tabelle 4.1 aufgeführt. Die benutzerdefinierten Event-Handler sind insbesondere für Modell-Ereignisse geeignet. Mit Hilfe dieses Werkzeuges kann der Anwender nach seinen individuellen Präferenzen steuern, welche Modell-Ereignisse für welche Objekte als relevant anzusehen sind.

Ereignis	Auslöser	Handler	Parameter
Objekt hat sich geändert	Objekt O	Visualisierung aktualisieren	Objekt O, Visualisierungs-Tool V
Modellschritt beendet	Modell M	Objekt exportieren	Objekt O, Export-Factory E, Speicherort D
Modellschritt beendet	Modell M	Visualisierung aktualisieren	Objekt O, Visualisierungs-Tool V
...	...	...	...

Tabelle 4.1: Mögliche benutzerdefinierte Event-Handler bestehen aus einem Ereignis (links) und einer Aktion (rechts).

Zu beachten ist, dass die Möglichkeit benutzerdefinierter Event-Handler den Verantwortungsbereich des Anwenders erheblich ausweitet, da sich die Event-Handler – wie oben angesprochen – entscheidend auf die Performanz des Modellablaufs auswirken können.

4.4 Modell-Einbettung in XULU

Da XULU eine Stand-Alone-Anwendung darstellt und trotzdem verschiedene und unterschiedliche Modelle aufnehmen soll (A2, E1, E7), werden die Modelle als Plugins in XULU integriert (vgl. 4.2)[4]. Von jedem Modell können (durch den Anwender) beliebig viele Instanzen in den *Modell-Manager* der Plattform geladen werden. Dieser übernimmt dabei die Aufgabe

- die einzelnen Modell-Instanzen zu verwalten (laden/entfernen)
- für jedes Modell eine Steuerungskomponente zur Verfügung zu stellen
- die Modelle zu initialisieren
- ggf. die Modelle zu synchronisieren

4.4.1 Ressourcen-Philosophie

Damit den Modell-Implementierungen nur die Aufgabe der Modell-Algorithmik zukommt (E2), stellen sie eigenständige und von der Plattform weitestgehend isolierte Komponenten dar, die *keinen* direkten Zugriff auf die einzelnen Plattform-Komponenten haben (z.B. Visualisierung und insbesondere Datenpool, siehe hierzu Abb. 4.6 auf Seite 47). Statt dessen teilt das Modell der Plattform (Modell-Manager) in einer Vorlaufphase alle für den Ablauf benötigten Daten als sog. *Ressourcen* mit. Dabei besteht eine Ressource aus 4 Teilen:

- **Beschreibung**: Aussagekräftige Beschreibung (gegenüber dem Anwender), zu welchem Zweck die Ressource benötigt wird (z.B. auch erwartete Raster- oder Listengröße).
- **Datentyp**: Art des angeforderten Objekts (z.B. Raster, Liste, ...).
- **Nullable-Flag**: Signalisiert der Plattform (bzw. dem Anwender), ob es sich um eine optionale Ressource handelt.
- **Datenobjekt**: Nimmt das Objekt auf, das der Anwender der Ressource zuordnet.

Anschließend ist es Aufgabe des Anwenders, diese Ressourcen mit Referenzen auf „passende“ Objekte des Datenpools zu füllen (vgl. 4.4.2). Dabei werden alle Ressourcen gleich behandelt. Welche als Eingabe, Ausgabe oder Zwischenspeicher dienen, wird

[4] An dieser Stelle sei daran erinnert, dass die XULU-Modelle als fest-programmierte Module vorliegen und *nicht* interaktiv (via Drag&Drop) erstellt werden (vgl. Beginn dieses Kapitels)

dem Anwender lediglich als (visuelle) Information vermittelt. Hierdurch wird das Zusammenspiel zwischen Modell und Datenbasis auf einem sehr einfachen und erweiterbaren Level gehalten. Theoretisch wäre es denkbar, neben statischen Daten, auch semantische Objekte – wie Sub-Modelle – als Ressourcen an ein Modell zu übergeben.

Ein weiterer entscheidender Punkt der Ressourcen-Philosophie ist, dass ein Modell keine Objekte selbst erzeugen und verwalten soll. Auch temporäre (nur modell-intern genutzte) Objekte sollten als Ressourcen aus dem Datenpool angefordert werden[5]. Somit bleibt die Daten- und Speicherverwaltung vollständig in der Verantwortung der Plattform und braucht von den Modell-Implementierungen nicht berücksichtigt werden (siehe 4.8.1).

4.4.2 Modell-Steuerung

Auf Basis der durch das Modell mitgeteilten Ressourcen kann der Modell-Manager ein allgemeines – für *alle* Modellarten verwendbares – User Interface erstellen, in dem der Anwender interaktiv jeder Ressource ein passendes Objekt aus dem Datenpool zuordnen kann (Abbildung 4.6).

Um zu gewährleisten, dass ein Modell während seines Ablaufs die restliche Plattform – also GUI, andere Modelle, Visualisierung – nicht blockiert, sorgt der Modell-Manager dafür, dass jede Modell-Instanz in einem eigenen Prozess ausgeführt wird. Dies geschieht jedoch transparent für den Anwender und muss bei der Modell-Entwicklung nicht berücksichtigt werden. Das oben angesprochene UI dient dem Anwender ebenfalls dazu, diesen Modell-Prozess zu steuern. Da der Modellalgorithmik von der Plattform-Seite nur wenige strukturelle Vorschriften aufgelegt werden (E7), beschränken sich die bereitgestellten Möglichkeiten der Modellsteuerung jedoch auf sehr grundlegende Funktionen:

- **Init**: Führt die Initialisierung durch (setzt das Modell zurück)
- **Start**: Startet den Modellablauf
- **Stop**: Beendet den Modellablauf komplett
- **Pause**: Hält den Modellablauf (zwischenzeitlich) an

Um dem Modell-Entwickler die Aufgabe einer GUI-Programmierung abzunehmen (E3), enthält das allgemeine UI zudem eine Komponente für einfache (textuelle) Statusausgaben, auf die der Modell-Entwickler innerhalb des Modell-Algorithmus zurückgreifen kann. Für manche Modelle kann es jedoch erforderlich sein, dem Anwender

[5] Dies bezieht sich insbesondere auf komplexe, speicherplatzintensive Daten (z.B. Raster). „Kleine“ Datenstrukturen – wie einzelne Zahlen, Zeichenketten, kleine Arrays oder Listen – können natürlich auch modell-intern verwaltet werden.

während des Modellablaufs auch stark modellspezifische Informationen oder zusätzliche Steuerungselemente zur Verfügung zu stellen (z.B. Fortschritts- oder Abweichungsbalken für bestimmte LUC-Typen). Deshalb sieht die Plattform – als Ergänzung zu dem allgemeinen Steuerungs-UI – auch die Integration einer modell-spezifischen Komponente vor. Ob und wie diese Option genutzt wird, hängt von den Fähigkeiten und der Intention des Modell-Entwicklers ab.

Verzichtet der Modell-Entwickler auf eine solche GUI (und die Verwendung neuer, modellspezifische Datentypen), so ist es für ihn möglich, ein Modell ausschließlich durch seine Algorithmik, zu implementieren. Einfache Modellierungsideen können also auch *ad hoc* umgesetzt werden („schnell einmal ausprobieren“, E4).

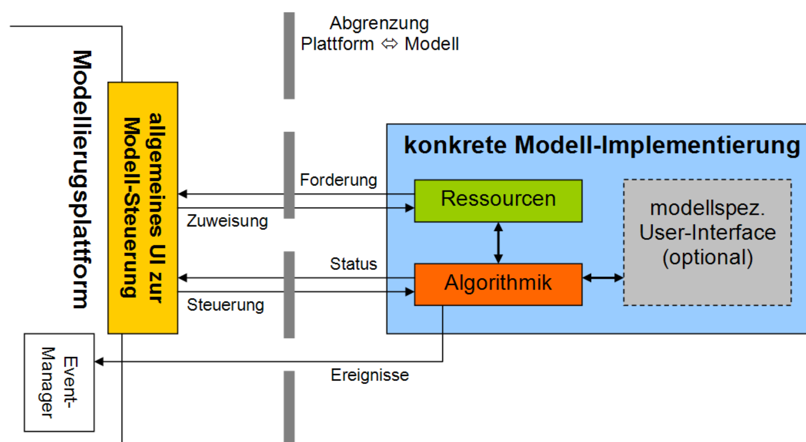


Abbildung 4.6: Die Modell-Semantik (rechts) wird strikt von den Anwendungsfunktionen der Plattform (links) getrennt und beschränkt sich im Wesentlichen auf die Algorithmik.

4.4.3 Der Weg zum Modell-Ergebnis

Abbildung 4.7 zeigt die unterschiedlichen Phasen, die zwischen dem Starten der XULU-Applikation und dem Modell-Ergebnis liegen[6]. Der Ablauf wird durch den Modell-Manager vorgegeben. Er sichert somit z.B. zu, dass vor dem Start die vom Anwender vorgenommene Ressourcen-Zuteilung auf Korrektheit überprüft wird (z.B. alle angegebenen Ein- und Ausgabe-Raster die gleiche Größe haben). Da diese Überprüfung jedoch stark von der jeweiligen Modell-Semantik abhängt, fällt sie in den Aufgabenbereich der Modell-Implementierung und wird nicht vom Modell-Manager durchgeführt. Der Modell-Manager initiiert lediglich den Check.

Welche Abläufe (Verzweigungen, Schleifen, ...) innerhalb des Algorithmus-Blocks vorgenommen werden, ist vollständig modell-spezifisch. Wichtig ist lediglich, dass die Modell-Algorithmik an „aussagekräftigen" Stellen Events initiiert, damit der Anwender die Möglichkeit hat, über Event-Handler darauf zu reagieren (z.B. Visualisierungsupdate, vgl. 4.3). Dies kann jedoch nur eine Richtlinie an den Modell-Entwickler sein, die er nicht zwangsläufig einhalten muss.

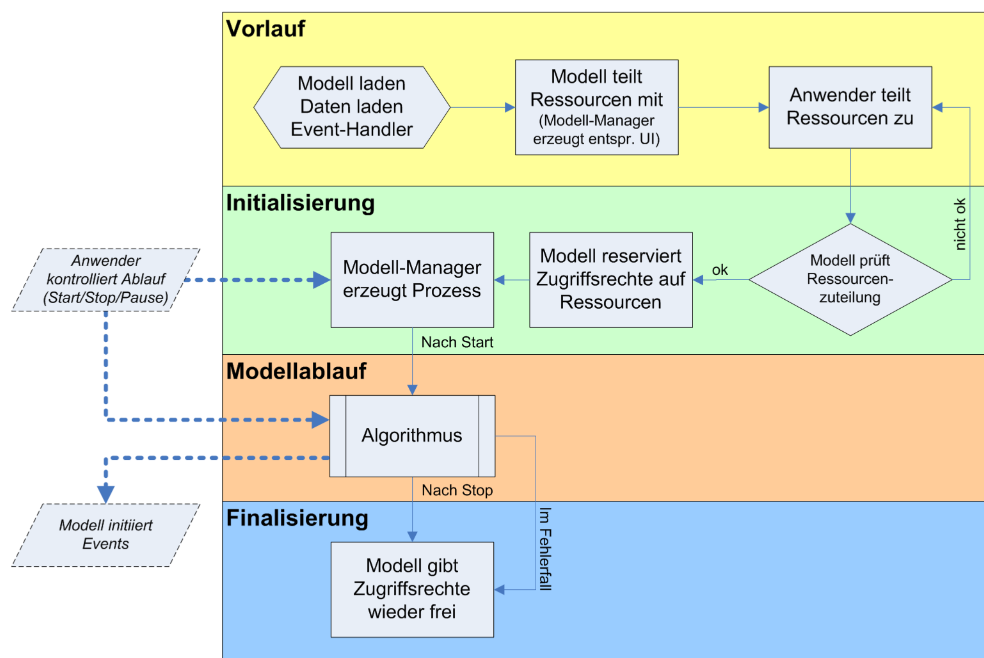


Abbildung 4.7: Der Verlauf von XULU-Programmstart bis hin zum Modell-Ergebnis erfolgt in verschiedenen Phasen.

[6] Die Modell-Implementierung ist hierbei außen vor gelassen.

4.4.4 Modell-Schnittstelle

Die vorangegangenen Abschnitte haben das angestrebte Zusammenspiel zwischen der Modellierungsplattform XULU und einem Modell-Plugin dargestellt. Um dieses zu realisieren, müssen eine Reihe von Anforderungen (in Form von Eigenschaften/Methoden) an eine Modell-Implementierung gestellt werden. Diese sind in Tabelle 4.2 nochmals zusammenfassend dargestellt. Dabei ist festzuhalten, dass die Schnittstelle ...

... die Modell-Implementierung im Wesentlichen auf den Modell-Algorithmus reduziert (E2)

... keine Einschränkungen an den Modell-Algorithmus vorgibt (E7)

... es zulässt, dass ein Modell auch intern (als Subroutine andere Modelle) verwendet werden kann (E6)

... und somit den wichtigsten Anforderungen des Modell-Entwicklers genügt. E6 wird dadurch erfüllt, dass das Modell lediglich seine Ressourcen (auf Anfrage) bekannt gibt und kein direkter Zugriff auf die Plattform erforderlich ist.

Durch die Trennung der Funktionalitäten *Init* und *Start*, sowie *Stop* und *Dispose*[7] wird der Modellierungsplattform zudem die Möglichkeit gegeben, gewisse Abläufe vorzuschreiben (vgl. 4.4.3). Zum Beispiel gewährleistet der Modell-Manager, dass nach dem Beenden eines Modells oder in einer Fehlersituation immer ein *Dispose* vorgenommen wird. Dies liegt somit nicht in der Verantwortung der Modell-Implementierung[8]

4.4.5 Multi-Modell-Szenarien

Da XULU es gestattet, dass mehrere Modelle gleichzeitig in die Plattform geladen und prinzipiell auch parallel ablaufen können, stellt sich automatisch die Frage, wie diese Abläufe koordiniert werden. Dies ist insbesondere dann von Bedeutung, wenn die Modelle auf gemeinsame Ressourcen (des Datenpools) zugreifen. Die Verantwortung hierüber liegt beim Modell-Managers, da er diejenige XULU-Komponente ist, die die Modelle verwaltet, also Kenntnis darüber hat, welche Modelle gerade aktiv sind, auf welche Daten diese Modelle zugreifen und ob sich daraus Konflikte ergeben.

Hierfür sieht XULU einfache Zugriffs-Mechanismen vor, die es verhindern, dass z.B. mehrere Modelle (oder andere Komponenten) gleichzeitig schreibend auf ein Objekt des Datenpools zugreifen (vgl. 4.8.2). Dies reicht für die Synchronisation eines komplexeren Zusammenspiels zwischen zwei oder mehr Modellen jedoch nicht aus, da

[7] Die Initialisierungsaufgaben könnten ja auch der *Start*-Methode zugeschrieben werden, genauso die *Dispose*-Funktionalität der *Stop*-Methode.

[8] **Ausnahme:**
Wird ein Modell als Sub-Routine eines anderen Modells aufgerufen (E6), ist die Implementierung des aufrufende Modell für solche Abläufe verantwortlich!

Modell-Funktion	**Bedeutung für die Plattform**
Ressourcen mitteilen	Ermöglicht die generische Generierung einer GUI, über die der Anwender die Ressourcen zuordnen kann.
Ressourcen checken	Überprüft die zugeteilten Ressourcen auf Konsistenz.
Geplante Events mitteilen	Ermöglicht dem Anwender ein automatisches Event-Handling.
Event-Listener verwalten und informierrn	Ermöglicht dem Event-Manger (und anderen Komponenten) auf Modell-Ereignisse zu lauschen.
Init	Überprüfung der bereitgestellten Ressourcen und Initialisierung aller internen Variablen.
Start	Startet den Modellablauf, definiert die Modell-Algorithmik.
Stop	Bricht den Modell-Algorithmus ab.
Dispose	Modell gibt Zugriffsrechte und interne Ressourcen wieder frei.
Error-Handling	Behandlung von Fehlern, falls das Modell unkontrolliert abbricht.
Modellspezifisches UI mitteilen	Ermöglicht Integration modellspezifischer Steuerungselemente in die Plattform.

Tabelle 4.2: Übersicht über die Anforderungen an ein Modell.

auch zeitliche Aspekte beachtet werden müssen. Zur Verdeutlichung möchte ich folgendes (durchaus realistisches) Szenario anführen:

> Eine Implementierung des CLUE-Modells (vgl. 2.3) greift zu Beginn jedes Zeitschritts auf ein unabhängiges DEMAND-Modell zurück. Dieses erstellt auf Basis der zuletzt durch CLUE generierten Siedlungsflächen neue Bedarfsanforderungen (siehe Abb. 4.8).
> Für einen korrekten Ablauf des Gesamt-Modells muss abwechselnd erst ein Schritt des DEMAND-Modells ausgeführt werden und anschließend ein Schritt des CLUE-Modells. Während ein Modellschritt ausgeführt wird, muss das jeweils andere Modell im Wartezustand verweilen.

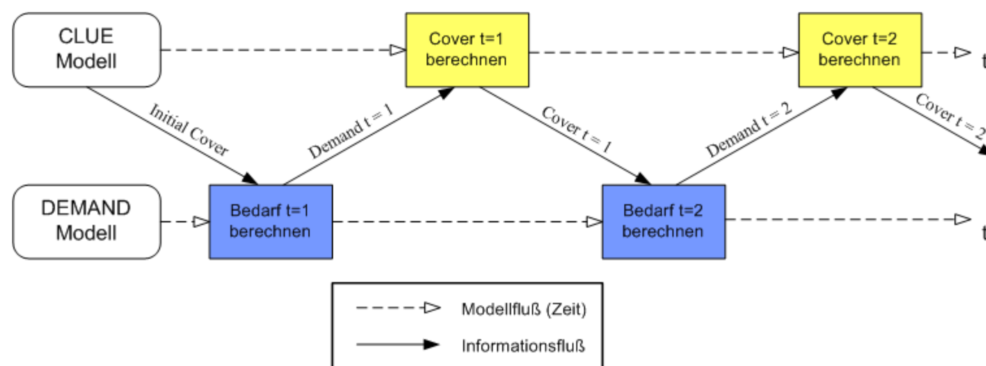


Abbildung 4.8: Zwei unabhängige Modelle, die auf gegenseitigen Modellausgaben operieren.

Für dieses Beispiel wäre die beschriebene Synchronisierung noch recht einfach zu automatisieren. Eine vollautomatische und allgemeine Synchronisation ohne Kenntnis der jeweiligen Modellsemantik ist jedoch überaus komplex. Selbst wenn alle Modelle a priori bekannt geben, zu welchem Modellzeitpunkt sie eine bestimmte Ressource (für lesenden und/oder schreibenden Zugriff) benötigen[9], gestaltet sich bereits ein serielles Scheduling als schwierig. Man kann dies mit der Serialisierung von Datenbank-Transaktionen vergleichen [16, 13]. Im Gegensatz zu einzelnen Datenbank-Transaktionen, müssen für Modellabläufe jedoch noch zusätzliche zeitliche Aspekte berücksichtigt werden, da bestimmte Schrittabfolgen von wesentlicher Bedeutung sind! Noch komplizierter wird es, wenn Zugriffszyklen auftreten, die Modelle nicht gleich getaktet sind (d.h. wenn z.B. ein Modell in einem Schritt 5 Monate modelliert, ein anderes Modell aber ein ganzes Jahr) oder aus zeitlichen Effizienzgründen eine parallele Abarbeitung angestrebt wird. Letzteres erfordert zusätzliche Buffering-Strategien, damit ein Modell bereits einen neuen Zustand schreiben kann, während ein anderes, zeitlich „zurückhängendes“ Modell noch auf den alten Zustand (lesend) zugreift. Aus Speicherplatzgründen gestaltet sich dies wiederum für große Datentypen (z.B. Raster bei Landnutzungsmodellierung) als sehr problematisch.

[9] In der Praxis stellt diese Annahme bereits eine sehr große Einschränkung dar, da sich „Zugriffswünsche“ häufig erst aus dem Modellablauf heraus ergeben!

Der vorangegangene Abschnitt zeigt, dass die Thematik der Modell-Synchronisation zu komplex ist, um im Rahmen dieser Diplomarbeit näher behandelt zu werden. An dieser Stelle möchte ich deshalb von den eigentlichen Schedulingstrategien abstrahieren und mich darauf beschränken, zu betrachten, *wie* solche Strategien und -mechanismen in den XULU-Modell-Manager integriert werden können:

Zunächst ist durch den Anwender festzulegen, welche Modelle synchronisiert werden sollen. Um eine kontrollierte Synchronisation zu gewährleisten, können diese Modelle während des Modell-Ablaufs nicht mehr individuell (einzeln) gesteuert werden, sondern nur noch kollektiv, d.h. wird ein Modell initialisiert, so werden alle Modelle initialisiert; wird ein Modell angehalten, werden alle Modelle angehalten; usw.
Der Synchronisationsmechanismus wird durch einen *Modell-Scheduler* implementiert, der vom Modell-Manager Zugriff auf alle zu synchronisierenden Modell-Prozesse erhält[10]. Die Implementierung des Modell-Schedulers kann somit vollkommen individuell festlegen, welches Modell zu welchem Zeitpunkt laufen darf und welches (aus welchen Gründen auch immer) warten muss.

Da in unterschiedlichen Anwendungsgebieten und Szenarien unterschiedliche Synchronisationsmechanismen vorteilhaft sind, ist es sinnvoll, die Modell-Scheduler in Form von Plugins (dynamisch) in den Modell-Manager zu integrieren. Je nach Situation kann der Anwender eine geeignete Strategie auswählen oder selbst einen speziellen Algorithmus implementieren und einbinden.

4.5 XULU-Visualisierung

Um der Anwenderanforderung A9 nachzukommen, sieht XULU vor, dass Daten-Objekte in Visualisierungstools grafisch dargestellt werden können. Auf den ersten Blick könnte es sinnvoll erscheinen, die Visualisierungsart eines Daten-Objekts direkt an den jeweiligen Datentyp zu koppeln, denn dann wären diese beiden Aspekte zentral und „nahe beieinander“ implementiert. Desweiteren könnte der interne Aufbau des Datentyps für eine effiziente Visualisierung genutzt werden.

Bei näherer Betrachtung kommt man jedoch schnell zu dem Schluss, dass aus Anwendersicht eine Kopplung der grafischen Darstellung direkt an den Datentyp *nicht* sinnvoll ist. Dies zeigt sich bereits, wenn ein Objekt gleichzeitig mehrfach angezeigt werden soll, in dem einen Fenster blau, in einem anderen rot. Dies ist nicht möglich, wenn das Datenobjekt gleichzeitig das Visualisierungsobjekt ist.
Viel wichtiger ist jedoch, dass eine geeignete Darstellungsart häufig erst durch die *Daten-Semantik* ersichtlich wird. So kann eine grosse 2D-Matrix als Raster aufgefasst werden, für das eine Kartendarstellung wünschenswert ist. Andererseits kann mit der Matrix auch eine Menge unabhängiger Zahlenreihen gemeint sein, für die eine Diagrammform sinnvoll ist (Abbildung 4.9). Eine dritte Möglichkeit wäre, das Raster als DHM aufzufassen, welches in 3D zu visualisieren wäre.

[10] Und somit auch Zugriff auf die Modelle selbst, sowie die verwendeten Ressourcen.

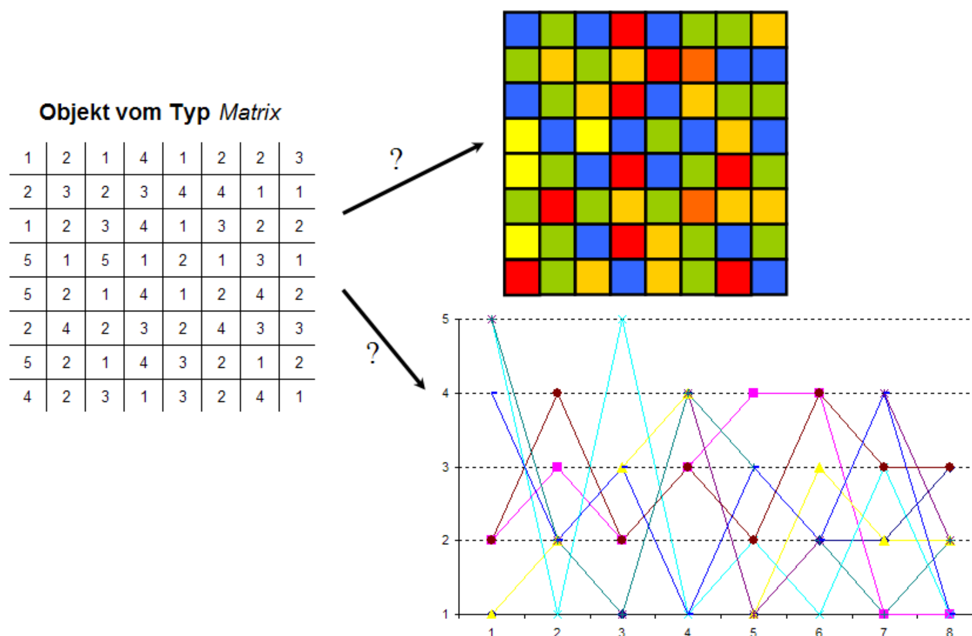


Abbildung 4.9: Die Art der Visualisierung kann nicht an einen Objekttyp gebunden werden. Der Anwender trifft diese Entscheidung manuell anhand der Objekt-Semantik.

Für die Modellierungsplattform XULU wird deshalb die Implementierung der grafischen Darstellung von den Datentypen getrennt und in einzelne Visualisierungstools verlagert. Über Plugins können beliebig viele Visualisierungstools in den *Visualisierungs-Manager* der Plattform integriert werden (vgl. 4.2), wobei jedes Tool für *eine* Darstellungsart steht (z.B. Geo-Visualisierung oder Diagramm). Die Kontrolle, welches Objekt wie dargestellt wird, liegt vollständig beim Anwender, in dem er (zur Laufzeit) ein für sein Anwendungsgebiet geeignetes Tool auswählt.

Die Integration als Plugin erleichtert überdies auch die Handhabung modellspezifischer Datentypen (z.B. IMPETUS-Dorf), die durch generische Visualisierungstools u.U. nicht dargestellt werden können. Statt diese (aufwändig) abzuändern, implementiert der Modell-Entwickler (neben dem modellspezifischen Datentyp) eine entsprechende Visualisierung und bindet sie parallel zu den bestehenden Tools in XULU ein. Alle bestehenden Anwendungen bleiben von dieser Weiterentwicklung unbeeinflusst.

Neben den Darstellungssarten ist es auch Aufgabe des Visualisierungs-Managers, einzelne Instanzen der Visualisierungstools zu verwalten. Von jedem Visualisierungstool sollten prinzipiell beliebig viele Instanzen (Fenster) geöffnet werden können. Hierdurch wird dem Anwender insbesondere für grafische Objekte – wie Karten, Raster oder 3D-Objekte – ermöglicht, gleichzeitig verschiedene Perspektiven zu betrachten (z.B. eine grobe Übersicht und ein Detail-Fenster, in dem er an ein konkretes Objekt heranzoomt).

4.6 XULU-Datenpool und Datenbeschaffung

Die Aufgabe des XULU-Datenpools besteht darin, sämtliche Daten zu verwalten, mit denen innerhalb der Plattform gearbeitet wird. Mit *Datenverwaltung* ist hierbei das zentrale – durch den Anwender gesteuerte – Sammeln und Bereitstellen von Objekten gemeint (z.B. in Form einer Liste oder Hash-Tabelle). Davon zu unterscheiden ist die Aufgabe der *Datenorganisation*, also des (internen) Objekt-Aufbaus. Diese fällt *nicht* der Datenpool-Komponente zu, sondern der jeweiligen Datentyp-Implementierung (siehe hierzu 4.8.1). Diese bestimmt beispielsweise auch, ob ein Objekt komplett im Hauptspeicher organisiert wird, oder seine einzelnen Bestandteile „bei Bedarf" aus dem Hintergrundspeicher oder einer Datenbank nachgeladen werden.

Der Datenpool stellt – neben der Modell-Steuerung – die wichtigste Plattform-Komponente dar, in der Anwender-Interaktion eine Rolle spielt. Ausgehend vom User Interface des Datenpools initiiert der Anwender sämtliche Objekt-Operationen (Importieren, Exportieren, Löschen, Visualisieren, usw.). Deshalb ist es wichtig, dass ein Objekt gegenüber dem Anwender aussagekräftig repräsentiert wird (A5). Hierfür wird eine (textuelle) Objekt-Bezeichnung vorgesehen, die der Anwender (ebenfalls über das Datenpool-UI) frei festlegen kann. Da die aussagekräftige Repräsentation eines Objekts jedoch nicht nur im Datenpool benötigt wird, sondern z.B. auch in einem Visualisierungtool hilfreich ist, fällt die Verwaltung einer solchen Bezeichnung nicht in den Bereich des Datenpools, sondern in den der Datentypen.

In erster Linie erfolgt die Datenverwaltung für die von den Modellen benötigten Ressourcen (sowohl Input, als auch Output)[11]. Dabei erhält jedoch nicht jedes Modell „seinen eigenen" (isolierten) Bereich, sondern der Datenpool stellt eine *gemeinsame* Datenbasis für *alle* Modelle zur Verfügung. Der Datenaustausch *zwischen* verschiedenen Modellen reduziert sich hierdurch auf den Zugriff auf dieselbe Ressource (A2, A3). Welche Probleme damit verbunden sein können, wurde bereits in Abschnitt 4.4.5 „MULTI-MODELL-SZENARIEN" erläutert.

Damit Daten für ein Modell als Ressource zur Verfügung stehen, müssen sie zunächst komplett in den Datenpool „geladen" werden (*Datenbeschaffung*). Hierzu bietet das Datenpool-UI verschiedene Schnittstellen an:

- Datei-Import und -Export
- Erzeugen komplett neuer (leerer) Objekte
- Datenbank-Anbindung
- Herunterladen aus dem Internet (z.B. einzelne Dateien, WMS, WFS)

[11] Es ist natürlich auch möglich, Daten nur zur Visualisierung oder Bearbeitung in die Plattform zu laden.

Da es in der Regel sehr viele verschiedene Formate gibt, in denen ein Objekt(-typ) *materialisiert* werden kann, wird die Problematik der Datenbeschaffung von den Datentypen getrennt[12]. Beispielsweise sind ARCINFOASCIIGRID, GEOTIFF und JPEG nur drei von vielen Datei-Formaten, in denen Raster-Daten vorliegen können. Daneben bestehen Austausch-Möglichkeiten mit Datenbanken, die gänzlich anders aufgebaut sind, als Datei-Zugriffe. Da die Objekt-Erzeugung einem stetigen Entwicklungsprozess unterliegt, müsste für jedes neue Datenformat der entsprechende Datentyp umprogrammiert werden. Dies ist nicht sinnvoll, da ein Datentyp ein beständiges Element der Modellierungsplattform darstellen soll (vgl. 4.8).

Da man dasselbe Argument auch für den Datenpool anführen kann, wird die Datenbeschaffung ebenfalls nicht fest in den Datenpool integriert. Statt dessen sieht XULU den Daten-Import und -Export (im Rahmen des *Factory*-Konzepts, 4.8) über dynamische Plugins vor. Jedes Plugin implementiert den Import oder Export für *ein* Datei- oder Datenbank-Format. Der Datenpool[13] veranlasst lediglich den Factory-Aufruf. Die Datenbeschaffung bleibt somit sehr flexibel und kann sukzessive um neue Austausch-Möglichkeiten erweitert werden, ohne den Datenpool (oder die Datentypen) selbst anpassen zu müssen (A6, A7).

4.7 Datenmanipulation in XULU

Eine Anforderung des Modell-Anwenders besteht darin, Datensätze bereits innerhalb der XULU-Modellierungsplattform manuell verändern zu können, um im Rahmen des *Decision Support* alternative Szenarien zu modellieren (A8).
Aus Zeitgründen spielt jedoch die Entwicklung von interaktiven Datenmanipulations-Funktionen im Rahmen der Diplomarbeit keine Rolle. An dieser Stelle möchte ich nur auf die grundlegende Schnittstelle eingehen, um zu verdeutlichen, wie die Integration von Datenmanipulation in XULU ermöglicht werden kann.

Die Steuerung interaktiver Datenmanipulation – insbesondere komplexer Datentypen – ist extrem datentyp-abhängig. Für das allgemeine Datenpool-UI sind deshalb nur Änderungsfunktionen auf einfachen Basis-Datentypen (wie z.B. Zahlen, Strings oder entsprechende Listen) vorgesehen, da diese sehr einfach textuell manipuliert werden können. Für alle anderen Datentypen wird in XULU analog zur Visualisierung verfahren (vgl. 4.5) und die Datenmanipulation in Plugins ausgelagert, welche der Anwender zur Laufzeit frei auswählen kann. Ein *Manipulations-Tool* erhält (vom XULU-Datenpool) ein Objekt mit „dem Auftrag“, dem Anwender ein adäquates User Interface zur Verfügung zu stellen, mit dem er das Daten-Objekt verändern kann. Wie diese realisiert ist – z.B. im Fall von Rastern per Drag&Drop oder durch manuelles Eingeben von Koordinaten – hängt von der jeweiligen Implementierung des Plugins ab[14].

[12] Zumindest, wenn eine Datenorganisation komplett im Hauptspeicher möglich ist. Geht es darum, Objekt-Teile – wie oben angedeutet – bei Bedarf nachzuladen (z.B. aus einer Datenbank), sollte dies direkt in einem speziellen Datentyp implementiert werden.

[13] Bzw. der Anwender über das User Interface des Datenpools

[14] Alternativ kann eine Datenmanipulation auch in ein Visualisierungstools integriert werden. Denkbar ist dies zum Beispiel für Vektor-Datensätze.

4.8 XULU-Datentypen

Um die Modellierungsplattform für viele Anwendungsgebiete einsetzen zu können, ist es wichtig, dass die Datentypen als eine von den Modellen (und auch von der Plattform) unabhängige Bibliothek von Plugins in XULU realisiert werden. Zum einen können so verschiedene Modelle auf den gleichen Datenstrukturen aufbauen (A4, E5), zum anderen bleibt XULU flexibel erweiterbar, ohne dass interne Anpassungen vorgenommen werden müssen (E9).

Die Schnittstellen-Anforderungen, die für die Datentyp-Plugins notwendig sind, werden in Abschnitt 4.8.3 zusammengefasst. Welche weiteren Konzepte beim Entwurf neuer (u.U. modellspezifischer) Datentypen beachtet werden sollten, beleuchtet 4.8.4. Zuvor möchte ich jedoch auf einige grundlegende Aspekte eingehen, wie die Datentypen in XULU behandelt werden.

4.8.1 Objekt-Aufbau und Erzeugung durch Factorys

In den vorangegangenen Kapiteln wurde immer sehr allgemein mit dem Begriff *Datentyp* umgegangen. Dabei wurden zwei Themenbereiche zusammengefasst, die eigentlich zu unterscheiden sind:

- **Datendefinition:** Bestimmt die Schnittstelle eines Objekts, also *welche* Methoden und Eigenschaften es (nach außen) zur Verfügung stellt.
- **Datenorganisation:** Bestimmt, *wie* ein Objekt (intern) aufgebaut ist und wie die (durch die Datendefinition vorgegebenen) Methoden und Eigenschaften realisiert sind.

XULU trennt diese Bereiche. Wenn im folgenden Abschnitt von einem *Datentyp* oder *Interface* die Rede ist, so ist damit lediglich die Datendefinition gemeint. Geht es um eine konkrete Art der Datenorganisation, wird der Begriff *Datentyp-Implementierung* (oder kurz: *Implementierung*) verwendet.

Das *Factory-Konzept* basiert auf dem *Abstract Factory Pattern* [9, 4], das voraussetzt, dass allgemeine Objekt-Typen (z.B. Raster, LineString, MultiLineString, Agent) zunächst durch *Interfaces* spezifiziert werden. Die konkrete Implementierung (Datenorganisation) soll für die *Objekt-Nutzer* (insbesondere die Modelle) transparent sein. Sie arbeiten ausschließlich auf Basis der durch das Interface vorgegebenen Methoden (⇒ OO: Dynamisches Binden, vgl. 4.2).
Der Grund hierfür liegt darin, dass es sinnvoll sein kann, je nach Anwendungssemantik oder tatsächlicher Daten-Ausprägung unterschiedliche Implementierungen eines Datentyps einzusetzen. Dies lässt sich sehr gut am Beispiel von Raster-Daten verdeutlichen:

Für Raster-Daten sind verschiedene Implementierungen möglich, die je nach Anwendungsgebiet mehr oder weniger gut geeignet sind. Eine Implementierung als *Array* (Matrix) hat den Vorteil konstanter Zugriffszeit auf jede Zelle, unabhängig davon, wie die Werte im Raster verteilt sind. Jedoch ist eine Array-Implementierung immer mit linearem Speicherplatzbedarf verbunden[15], was sich bei sehr großen (oder feinen) Datensätzen als kritisch erweist. Eine alternative Implementierungsvariante bildet der *Region Quad-Tree*, bei dem das Raster baumartig aufgebaut wird und benachbarte Zellen, die mit dem gleichen Wert belegt sind, zusammengefasst werden[16]. Für Raster, in denen diskrete Werte gespeichert werden und die darüberhinaus in zusammenhängenden Flächen verteilt sind (z.B. Ausbreitung von Feuer), bietet der Quad-Tree (auch für grosse Datensätze) eine speicherplatz-effiziente Darstellungsform (siehe Abb. 4.10). Er hat

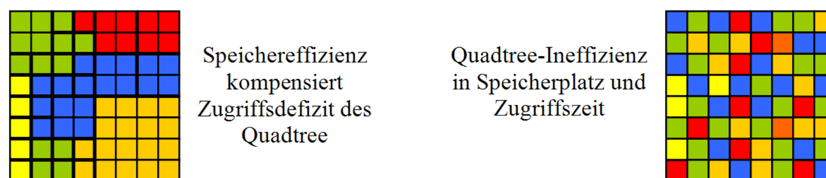


Abbildung 4.10: Vergleich eines flächendeckend verteilten mit einem stark gestreuten Raster.

gegenüber dem Array jedoch einen Nachteil in der Zugriffszeit, da kein wahlfreier Zugriff auf jede einzelne Zelle möglich ist. Statt dessen ist ein Durchlaufen der Baumstruktur notwendig, um einen Zellwert zu ermitteln. Liegt eine gestreute Verteilung der Rasterwerte vor, verliert der Quad-Tree auch den Vorteil der Speicherplatzeffizienz, da er im worst-case zu einem vollständigen Baum degeneriert[17].

Nach dem *Abstract Factory Pattern* werden die für alle (2D-)Raster-Implementierungen benötigten Zugriffsmethoden (z.B. `getValue(x,y)` und `setValue(x,y,v)`), durch ein übergeordnetes Interface *Raster2D* spezifiziert (vorgeschrieben). Wie diese Methoden arbeiten (auf Array oder Quad-Tree), bestimmt erst die jeweilige Implementierung *Raster2DArray* oder *Raster2DQuad*).

Über das *Abstract Factory Pattern* lassen sich Datentyp-Implementierungen sehr flexibel austauschen:
Die verschiedenen Objekt-Nutzer instanziieren dabei ein Objekt nicht selbst (via Konstruktor), sondern geben die Instanziierung bei einer *Factory* „in Auftrag". Welche konkrete Implementierung „geliefert" wird, hängt einzig und allein von der Factory ab. Soll

[15] Aus Komplexitätstheoretischer Sicht, bezogen auf die Anzahl der darzustellenden Raster-Zellen.
[16] Der Aufbau ist vergleichbar mit der gleichnamigen Index-Struktur für Geo-Datenbanken [40]. Eine binäre Variante ist in [27] beschrieben. Verschiedene QuadTree-Demos sind unter `http://www.cs.umd.edu/~brabec/quadtree/` zu finden.
[17] Aufgrund der Zwischenknoten benötigt dieser dann sogar wesentlich mehr Speicherplatz, als eine Array-Darstellung!

eine alternative Objekt-Implementierung eingesetzt werden (z.B. *Raster2DQuad* statt *Raster2DArray*), ist nur die entsprechende Factory anzupassen, und alle Objekt-Nutzer verwenden fortan die neue Datentyp-Implementierung. Eine explizite Anpassung aller Objekt-Nutzer ist nicht erforderlich!

Im *Abstract Factory Pattern* ist eine Factory für die Erzeugung ganzer Objekt-Gruppen verantwortlich. Im XULU-Factory-Konzept ist dies nicht so. Da es in XULU um individuelle Objekte geht, ist eine Factory immer genau für einen Objekttyp zuständig.
Statt eine bestehende Factory programmtechnisch anzupassen, gestaltet es sich im Rahmen einer Plattform flexibler, sämtliche Factorys als Plugins zu realisieren und (in der XULU-Registry) dynamische Tupel aus ...

- Datentyp (Interface)
- zum Datentyp gehörende Instanziierungs-Factory

... zu verwalten. Um zwischen alternativen Objekt-Implementierungen zu wechseln, reicht es, im entsprechenden Tupel, die Factory zu ändern. Dies kann auch zur Laufzeit – also ohne jegliche Programmanpassung – geschehen.

Darüberhinaus löst das Factory-Konzept ein weiteres Problem, das im Rahmen eines allgemeinen Datenpools auftritt. Häufig werden zur Objekt-Erzeugung zusätzliche Informationen benötigt (z.B. Raster-Größe oder -Auflösung). Es gestaltet sich äußerst schwierig, diese über einen generisch erstellten Anwender-Dialog des Datenpools abzufragen, da der Datenpool keine Informationen über die Objekt-Semantik besitzt. Statt dessen kann diese Aufgabe in die Factory verlagert werden. Da diese „weiß“, welches Objekt sie „produziert“, kann sie auch entsprechende Anwender-Dialoge kreieren, um die zusätzlichen Informationen einzuholen. Somit können alle Factorys über eine gleichlautende Routine angesprochen werden, unabhängig davon, welches Datenobjekt sie erzeugen! Dies vereinfacht wesentlich die internen Abläufe des Datenpools.
Im Rahmen von XULU wird das *Abstract Factory Pattern* auf den Daten-Import und -Export erweitert. Hierzu wird das oben beschriebene 2-Tupel um ...

- eine Liste von Import-Factorys
- eine Liste von Export-Factorys

... zu einem 4-Tupel erweitert. Wie die Instanziierungs-Factory erzeugt die Import-Factory ein Objekt und liefert es (an den Datenpool) zurück. Demhingegen erhält die Export-Factory ein Objekt (vom Datenpool) mit der Aufgabe dieses zu materialisieren (z.B. in einer Datei)[18]. Dabei ist eine Factory immer für genau ein Import- oder

[18] Da sie kein Objekt „produziert“, mag der Leser die Bezeichnung „Factory“ für die Export-Factory irreführend finden. Sie wurde primär wegen des Zusammenhangs mit dem Factory-Konzept (und aus Gründen der Vereinheitlichung) gewählt. Jedoch kann man den Begriff „Factory“ auch mit „Verarbeitung“ assoziieren, was sowohl auf Import, als auch auf Export zutrifft.

Export-Format zuständig. Neben der klassischen Ein- und Ausgabe in Dateien, können auch andere Austausch-Routinen (z.B. an eine Datenbank) über das Factory-Konzept realisiert werden (A7). Entsprechende Anwender-Dialoge – z.B. für den Aufbau einer Datenbank-Verbindung oder die Eingabe eines SQL-Querys – müssen jedoch innerhalb der jeweiligen Factory realisiert werden und werden nicht von der Modellierungsplattform zur Verfügung gestellt.

4.8.2 Zugriffskontrolle

Im Abschnitt 4.4.5 „MULTI-MODELL-SZENARIEN“ wurde angesprochen, dass es zu Problemen kommen kann, sobald in der Modellierungsplattform mehrere Modelle gleichzeitig ablaufen. Während auf die Lösung des Problems der zeitlichen Koordination (Synchronisation) im Rahmen dieser Diplomarbeit nicht näher eingegangen wird, möchte ich das Problem der *Zugriffskonkurrenz* an dieser Stelle genauer betrachten. Dieses tritt nämlich nicht nur dann auf, wenn mehrere Modelle gleichzeitig (schreibend) auf dem selben Objekt operieren, sondern bereits wenn der Anwender versucht ein Daten-Objekt während eines Modell-Ablaufs (oder eines Visualisierungsvorgangs) zu manipulieren oder aus dem Datenpool zu löschen.
Das Zugriffs-Konzept von XULU sieht deshalb vor, dass die Daten-Objekte den Zugriff auf „sich selbst“ überwachen und kontrollieren müssen. Dies geschieht, in dem das Objekt *Zugriffrechte* verteilt:

- **Leserecht:** Erlaubt nur den Zugriff auf lesende Objekt-Methoden, nicht aber auf Operationen, die das Objekt verändern.
- **Schreibrecht:** Erlaubt sowohl den Zugriff auf Methoden, die das Objekt lesen, als auch auf verändernde Methoden.

Ohne ein entsprechendes Zugriffsrecht ist keine Operation auf einem Daten-Objekt möglich. Wird ein Recht von einem Objekt-Nutzer nicht mehr in Anspruch genommen, muss es „zurückgegeben“ (bzw. ungültig gemacht) werden. Erst dies signalisiert dem Objekt, dass ein entsprechendes Recht wieder an einen anderen Objekt-Nutzer verteilt werden kann. Wie viele (gleichzeitige) Lese- und Schreibrechte zugelassen werden, bestimmt ein Objekt selbst, bzw. wird durch den Anwender bei der Objekt-Erzeugung (mittels Factory) festgelegt. Zugriffskonflikte – wie oben geschildert – können so automatisch vermieden werden:

- Indem immer nur ein Schreibrecht vergeben wird, können keine zwei Objekt-Nutzer gleichzeitig ein Objekt verändern.
- Indem keine Leserechte verteilt werden, solange ein Schreibrecht vergeben ist, kann verhindert werden, dass (z.B. während des Modellablaufs) ein inkonsistenter Objekt-Zustand gelesen wird.
- Indem ein Objekt Auskunft über vergebene Zuriffsrechte erteilt, kann der Datenpool unkontrollierte Lösch-Operationen abfangen.

4.8.3 Schnittstelle zu den Datentypen

Wie die vorangegangenen Abschnitte gezeigt haben, gehören Visualisierung und Materialisierung (verschiedene Datei-Formate) *nicht* in den Aufgabenbereich eines Datentyps. Die grafische Darstellung wird durch verschiedene Visualisierungstools geregelt (4.5) und die Materialisierung/Objekterzeugung durch *Factorys* (4.8.1).

Trotzdem haben sich aus dem Design der im bisherigen Kapitel beschriebenen XULU-Komponenten eine Reihe von Anforderungen an die Datentyp-Implementierungen ergeben, welche im Folgenden nochmals zusammenfassend dargestellt werden.

Zunächst erfordert es die Anwender-Interaktion, dass ein Objekt gegenüber dem Benutzer durch eine **Bezeichnung** (Name) aussagekräftig repräsentiert wird (vgl. 4.6). Da eine textuelle Bezeichnung für interne Referenzbildungen nicht gut geeignet ist, verwaltet jedes Objekt zusätzlich eine **numerische ID**, welche das Objekt innerhalb der Plattform (insbesondere im Datenpool) *eindeutig* identifiziert.

In Abschnitt 4.3 wurde erläutert, dass die Objekte Zustandsänderungen „mitteilen“ sollen, damit andere Komponenten (z.B. Visualisierung) darauf reagieren können. Wie die Modelle, muss ein Datenobjekt deshalb Methoden bereitstellen, über die sich **Listener** registrieren können, die mittels entsprechender Events informiert werden (z.B. „Objekt geändert“ oder „Objekt zerstört“)[19].

Um einen geordneten und konsistenten Objekt-Zugriff für alle Plattform-Komponenten zu ermöglichen, muss ein Objekt kontrollieren, „wer“ **aktuell Zugriff** auf das Objekt besitzt (vgl. 4.8.2) und darüber Auskunft erteilen. Anhand der Information, *ob* aktuell ein Zugriff besteht, kann z.B. der Datenpool verhindern, dass ein noch verwendetes Objekt durch den Anwender gelöscht wird.

Richtet man den Blick auf komplexere Datentypen, so kristallisiert sich eine weitere benötigte Objekt-Eigenschaft heraus. Insbesondere speicherplatz-intensive Objekte (z.B. grosse Raster) müssen dafür Sorge tragen, dass sie sämtliche in Anspruch genommenen **Ressourcen wieder freigeben** (z.B. Speicher), sobald sie geschlossen (aus dem Datenpool entfernt) werden. Eine entsprechende Methode muss für den Datenpool zugänglich sein.

Tabelle 4.3 zeigt eine Übersicht über die von XULU geforderte Schnittstelle zu den Datentypen. Diese wird über ein Interface spezifiziert, das alle Datentypen implementieren müssen.

[19] Folgt man dem in Abschnitt 4.3 angesprochenen Konzept, so sollte (im Idealfall) der Event-Manger der einzige Listener sein. Die Kopplung mehrerer Listener lässt jedoch weiterführende Möglichkeiten der Ereignis-Behandlung offen.

Objekt-Funktion	**Bedeutung für die Plattform**
Namen/Beschreibung verwalten	aussagekräftige Darstellung gegenüber dem Anwender
ID verwalten	eindeutige interne Identifikation
Zugriffsrecht verteilen	Objekt-Nutzern Zugriff auf Objekt-spezifische Eigenschaften erteilen (lesend/schreibend)
Auskunft über bestehende Zugriffe	Unterbinden von Objekt-Zugriffen (z.B. Objekt-Löschung)
Listener registrieren	Propagierung von Objekt-Änderungen
Objekt zerstören	Objekt gibt Ressourcen wieder frei

Tabelle 4.3: Übersicht über die Anforderungen an ein XULU-Daten-Objekt.

4.8.4 Entwurf neuer Datentypen

In Abschnitt 4.8.1 wurde bereits erläutert, dass es sinnvoll ist, die Datendefinition von der Datenorganisation zu trennen. An dieser Stelle möchte ich auf zwei weitere Strategien eingehen, nach denen bei der Erweiterung der Datentyp-Bibliothek vorgegangen werden sollte.

Aufbau von Objekt-Hierarchien

Das OO-Prinzip der *Vererbung* dient vor allem der Wiederverwendbarkeit. Bestehen semantische und strukturelle Zusammenhänge zwischen Objekt-Typen, ist es *nicht* sinnvoll, diese unabhängig voneinander zu implementieren[20]. Die Folge wäre identischer Programm-Code an mehreren Stellen, was den Aufwand für nachträgliche Änderungen und die Anzahl möglicher Fehlerquellen erheblich erhöht.

Betrachtet man zum Beispiel verschiedene Dörfer für ein IMPETUS-Modell, so haben alle *Dorf*-Typen (Altsiedel-Dorf, Neusiedel-Dorf, ...) gemeinsame Eigenschaften, z.B.

- Name
- Einwohnerzahl

[20] Selbst, wenn die Objekt-Struktur durch ein übergeordnetes Interface vereinheitlicht ist (Datendefinition)

- Geometrische Fläche
- Geographische Lage
- Algorithmus zur Siedlungsexpansion
- Algorithmus zur Feldflächen-Expansion
- ...

Würden all diese Eigenschaften nur durch ein gemeinsames Interface vorgeschrieben, wären alle Methoden zur Verwaltung dieser Eigenschaften in sämtlichen Dorf-Typen einzeln (und somit doppelt) zu implementieren. Insbesondere trifft dies auch auf die Expansions-Algorithmen zu, welche sich von Dorf-Typ zu Dorf-Typ nur durch unterschiedliche Präferenzfunktionen unterscheiden.

Statt dessen sollte eine Objekt-Hierarchie erstellt werden, in der die Unter-Typen sämtliche Eigenschaften und Methoden von den jeweiligen Ober-Typen *erben*. Eine mehrfache Implementierung beispielsweise der Verwaltung einer geometrischen Fläche wird vermieden:

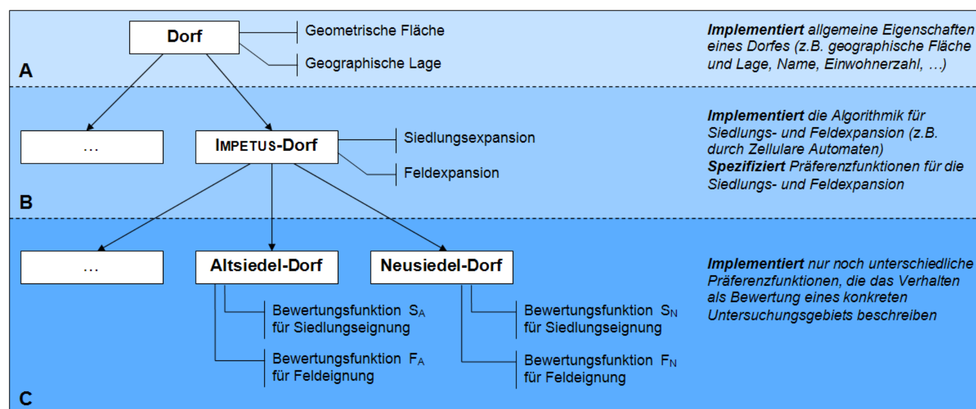


Abbildung 4.11: Ein möglicher struktureller Aufbau von *Dorf*-Datentypen.

Weiterentwicklung durch Erweitern

Programmanpassungen werden hauptsächlich aus zwei Gründen vorgenommen:

1. Fehlerbehebung
2. Weiterentwicklung

Bei der Fehlerbehebung ist es sinnvoll, bestehenden Programmcode direkt anzupassen, damit die Änderungen automatisch an *allen* Stellen wirksam werden, an denen der Programmcode (z.B. eine Methode oder Funktion) zum Einsatz kommt.

Bei der Weiterentwicklung[21] verhält es sich anders. Es ist vorzuziehen, existierende Funktionalitäten unverändert bestehen zu lassen und dem Konzept *Weiterentwicklung durch Erweitern* zu folgen. Dies wurde bereits in Abschnitt 4.2 angesprochen, in Verbindung mit den Datentypen werden jedoch sehr gut die Zusammenhänge mit den OO-Prinzipien *Vererbung* und *Dynamisches Binden* verdeutlicht.

Durch Vererbung können Anpassungen am Objektverhalten vorgenommen werden, ohne eine konkrete Objekt-Klasse A abzuändern. Die Anpassung besteht vielmehr darin, eine neue Objekt-Unterklasse B zu erstellen (von A *abzuleiten*), in der jedoch nur die änderungsrelevanten Methoden neu implementiert (*überschrieben*) werden. Alle übrigen Methoden und Eigenschaften werden automatisch von der Oberklasse geerbt. Wird bei der Objekt-Instanziierung die neue, abgeleitete Klasse B eingesetzt, führt das dynamische Binden dazu, dass automatisch die überschriebenen Methoden verwendet werden und nicht die Methoden der Oberklassen A.

Der Vorteil dieser Vorgehensweise (gegenüber dem Abändern) kann sehr gut anhand des Dorf-Beispiels aus dem vorangegangenen Abschnitt veranschaulicht werden:
Angenommen, anhand einer Fallstudie sei festgestellt worden, dass das Siedlungs-Verhalten eines Neusiedel-Dorfes besser durch eine neue Präferenzfunktion $S_{N'}$ beschrieben wird, als durch die bestehende Funktion S_N (vgl. Abb. 4.11). Würde im Typ *Neusiedel-Dorf* die Präferenzfunktion abgeändert, hätte dies Auswirkungen auf alle im modellierten Untersuchungsgebiet befindlichen Neusiedel-Dörfer. Dies ist jedoch nicht immer sinnvoll, denn für ein anderes Neusiedel-Dorf könnte die neue Präferenzfunktion (widererwartend) versagen. Besser ist es, die Objekt-Hierarchie um einen neuen Untertyp von *Neusiedel-Dorf* zu erweitern. In diesem wird nur die Präferenzfunktion S_N durch $S_{N'}$ überschrieben (und F_N geerbt). Der ursprüngliche *Neusiedel-Dorf*-Typ steht somit weiterhin für das Untersuchungsgebiet zur Verfügung. Der neue Untertyp, wird nur dort eingesetzt, wo er auch geeignet ist.

[21] im Sinne von Fortschritt oder Evolution

4.9 GUI-Komponenten

An einigen Stellen der vorangegangenen Abschnitte wurde bereits die Verwendung von *User Interfaces* angeführt. Im Prinzip benötigt jede XULU-Komponente eine GUI, über die der Anwender die Komponente steuern kann:

- **Datenpool:** Objekte hinzufügen, löschen, zum Visualisieren auswählen
- **Modell-Manager:** Modelle laden, löschen, steuern
- **Event-Manager:** Benutzerdefinierte Event-Handler erstellen
- **Visualisierungs-Manager**: Visualisierungstools anzeigen, schließen
- **Registry:** Plugins anzeigen und zur Laufzeit hinzufügen/entfernen

In den Rahmen einer komponenten-basierten Plattform (⇒ OO: Kapselung und Modularität) fällt jedoch nicht nur, dass die Komponenten untereinander gekapselt sind, sondern insbesondere auch die Trennung von Funktionalität und GUI. In einer Realisierung muss jede Komponente der Plattform[22] also aus 2 Teilen bestehen. Abbildung 4.12 zeigt dies exemplarisch für den Datenpool. Die Funktionalität der Datenverwaltung – z.B. als Hash-Tabelle – mit entsprechenden Einfüge- und Lösch-Operationen wird getrennt von der Datenpool-GUI implementiert, die für die Steuerung dieser Funktionen (über Buttons o.Ä.) zuständig ist. Somit ist es möglich, ohne die Datenverwaltung direkt anpassen zu müssen, das User Interface auszutauschen (und umgekehrt). Gleiches gilt für alle anderen XULU-Komponenten. Es besteht sogar die Möglichkeit, auch die GUI-Komponenten als dynamisch auswechselbare Plugins zu realisieren.

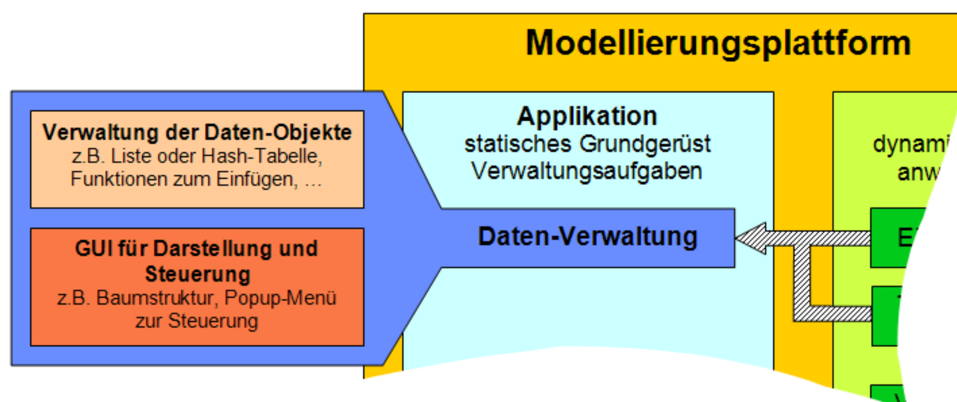


Abbildung 4.12: Alle Plattform-Komponenten bestehen aus 2 Teilen: Funktion und GUI.

[22] Mit Ausnahme der Datentypen.

4.10 Zusammenfassung

Die unterschiedlichen Anforderungen von Modell-Entwickler und -Anwender (Kap. 3) haben zu verschiedenen Konzepten für den Aufbau der XULU-Plattform geführt. Die wichtigsten bestehen darin, dass XULU eine (lauffähige) *Stand-Alone*-Applikation darstellt, welche über die dynamische Integration von *Plugins* zu einer Modellierungsplattform für unterschiedlichste Anwendungsgebiete erweitert werden kann (4.2). Hierbei spielen die Konzepte der Objekt-Orientierung (Modularität, Vererbung, Dynamisches Binden) eine zentrale Rolle. Im Hinblick auf eine Implementierung im Rahmen dieser Diplomarbeit war die Einschränkung in Kauf zu nehmen, dass Modelle durch fest programmierte JAVA-Klassen (Plugins) dargestellt werden und der Anwender diese nicht interaktiv erstellen kann (E8).
XULU bildet einen Rahmen, der die wichtigsten Anwendungsfunktionen (Datenverwaltung, Import/Export, Visualisierung, Datenmanipulation, Modellsteuerung) zur Verfügung stellt. Bei der direkten Modell-Implementierung müssen diese Aspekte somit nicht mehr berücksichtigt werden. Ein gemeinsamer Datenpool und eine erweiterbare Bibliothek von Datentypen reduziert den Datenaustausch zwischen Modellen auf den Zugriff auf dasselbe Daten-Objekt.
Der interne Aufbau von XULU erfolgt komponentenbasiert mit festen Schnittstellen zwischen den Modulen. Insbesondere erfolgt eine Trennung zwischen *Funktion* und *GUI*. Die Plattform bleibt somit nicht nur im Bereich der Plugins erweiterbar (4.1).

Am Beispiel des Datentyps Raster wird deutlich, dass es sinnvoll ist, Datendefinition und Datenorganisation voneinander zu trennen. Über das *Factory-Konzept* (4.8.1) können konkrete Typ-Implementierungen flexibel und vor allem transparent für die Modelle ausgetauscht werden. Ebenso gibt es verschiedenste Formen, wie ein und dasselbe Datenobjekt visualisiert und materialisiert (im- und exportiert) werden kann (4.5, 4.6). Entsprechende Routinen werden deshalb nicht fest an einen Datentyp gekoppelt, sondern ebenfalls über Plugins in XULU integriert. Im Rahmen des Entwurfs der Modellschnitte wurde zudem festgestellt, dass Daten-Objekte den *Zugriff* auf sich selbst verwalten und kontrollieren müssen (Lese/Schreibrechte), sowie Veränderungen an *Listener* melden sollen (4.8.2, 4.3). Für komplexe Multi-Modell-Szenarien reichen diese Mechanismen jedoch nicht aus (4.4.5).

Der Modell-Manager schreibt einen generellen Modellierungsablauf vor, der die mögliche Modellalgorithmik jedoch nicht einschränkt. Eine Modell-Implementierung reduziert sich somit auf bestimmte Phasen (Initialisierung, Modell-Algorithmus, Finalisierung). Dabei sieht das *Ressourcen-Konzept* vor, dass ein Modell nur über den Modell-Manager mit der Plattform kommuniziert und von den restlichen Komponenten (insbesondere Datenpool und Visualisierung) isoliert wird. Die für die Modellierung benötigten Daten spezifiziert ein Modell über *Ressourcen*. Diese werden durch den Anwender individuell zugeteilt (4.4). Speicherverwaltung und Datenorganisation bleiben im Verantwortungsbereich der Plattform. Um trotz dieser Isoliertheit, Plattform-Reaktionen auf den spezifischen Modellablauf zu ermöglichen (z.B. Visualisierungs-Update am Ende eines Zeitschritts), sieht XULU vor, dass ein Modell *Events* initiiert. Die Reak-

tion auf solche Ereignisse kann der Anwender individuell gestalten. Das *Event-Konzept* (4.3) verallgemeinert diese Vorgehensweise auf die anderen Plattform-Komponenten, insbesondere auf Daten-Objekte (*Änderungspropagierung*).

Die Entwicklung der einzelnen Plattform-Module führt zu Anforderungen, die sämtliche Daten-Objekte erfüllen müssen. Diese werden in einer allgemeinen Schnittstelle für Datentyp-Plugins zusammengefasst (4.8.3).

Kapitel 5

Implementierung von XULU

Die *eXtendable Unified Land Use Modelling Platform* ist eine in JAVA implementierte, komponentenbasierte Stand-Alone-Applikation, in die unterschiedlichste Modelle integriert werden können. Sie realisiert die wesentlichen in Kapitel 3 und 4 beschriebenen Anforderungen und Konzepte für eine generische Modellierungsplattform. Wie bereits aus dem Entwurf ersichtlich wird, ist XULU sehr offen gestaltet. Die wesentlichen Komponenten (Datentypen, Import/Export-Routinen, Modelle, Visualisierung) werden über dynamische Plugins in die Plattform eingebettet. Im Rahmen der Diplomarbeit wurden exemplarische Plugins realisiert, die sich auf das Anwendungsgebiet der Landnutzungsmodellierung[1] beziehen (also z.B. Raster- und Vektor-Datentypen, Geo-Visualisierung).

In diesem Kapitel kann jedoch nicht auf alle Einzelheiten der Implementierung eingegangen werden. Insbesondere verzichte ich darauf, die JAVA-Klassendefinitionen im Detail zu beschreiben. Hierzu ist eine *JavaDoc* besser geeignet, da sie genauen Aufschluss darüber gibt, welche Variablen und Methoden die einzelnen Klassen zur Verfügung stellen und welche Bedeutung diese haben [41]. Dieses Kapitel beschränkt sich auf die Beschreibung der Implementierungs-Konzepte, die bei der Realisierung der wichtigsten XULU-Komponenten verfolgt werden:

- Aufbau der XULU-Applikation (5.2)
- Integration der Plugins (5.3)
- Flexibilität des Event-Management (5.4)
- Realisierung der XULU-Datenverwaltung (5.5)
- Zusammenspiel zwischen Plattform und Modell (5.6)

Auf die genaue Implementierung der GUI-Komponenten, der Daten-Visualisierung, sowie der Verwaltung von Zugriffsrechten (auf Daten-Objekte) möchte ich bewusst verzichten, da hierfür zu tief auf die Details der JAVA-Programmierung eingegangen werden müsste.

[1] Wie sie zur Zeit von der RSRG betrieben wird.

Im folgenden Abschnitt 5.1 möchte ich jedoch zunächst die in XULU umgesetzten *Skript-Interpreter* darstellen, denen eine wesentliche Bedeutung in Bezug auf Anwenderfreundlichkeit und *Usability* zufällt. Diese Aspekte werden im Rahmen des Plattform-Entwurfs (Kapitel 4) nicht berücksichtigt, da hierbei die generelle Funktionalität der Plattform im Vordergrund steht. Aspekte der Anwenderfreundlichkeit kristallisieren sich zudem häufig erst im Zuge der Umsetzung heraus.

5.1 XULU-Skripte

Wie bereits zu Beginn des Entwurfs (Kapitel 4) dargestellt wird, werden XULU-Modelle durch fest programmierte JAVA-Klassen repräsentiert (vgl. 5.6). In der Phase der Modell-Entwicklung (insbesondere bei *Ad-hoc*-Realisierung) werden häufig Veränderungen an den Modell-Klassen vorgenommen (*trial-and-error*), was ein Neu-Kompilieren der jeweiligen Klasse erfordert. Im Rahmen der Realisierung der Plugin-Verwaltung musste diesbezüglich festgestellt werden, dass das dynamische Nach- und Neuladen von JAVA-Klassen Probleme bereitet (vgl. 5.3). Aus diesem Grund ist die gesamte XULU-Applikation neu zu starten, damit eine Modell-Änderung wirksam wird. Somit müssen Abläufe, wie Objekte in den Datenpool zu laden oder die Zuteilung von Modell-Ressourcen, während der Modell-Entwicklung vom Anwender sehr häufig wiederholt werden[2].

Um dem Anwender diese Abläufe zu vereinfachen, integriert XULU eine zusätzliche Plugin-Schnittstelle: Die XULU-*Skript-Interpreter*.
Mit ihnen können „Makros“ aus einer Datei eingelesen werden und innerhalb der XULU-Plattform ausgeführt werden. Jedes Skript-Interpreter-Plugin implementiert dabei einen bestimmten Funktionsumfang.

Im Rahmen der Diplomarbeit wurde ein Interpreter für Datenpool-Aktionen implementiert (siehe Anhang B):

- XULU-Objekt neu erzeugen
- XULU-Objekt aus Datei importieren
- Struktur eines bestehenden XULU-Objekts kopieren
- XULU-Objekt umbenennen

Ein ähnlicher Skript-Interpreter sollte für die angesprochene Zuteilung zwischen Modell-Ressourcen und Datenpool-Objekten implementiert werden, um diesen Ablauf insbesondere in der Phase der Modellentwicklung zu automatisieren. Idealerweise sollte im Zuge dieser Entwicklung auch die Modell-Steuerung (vgl. 5.6.2) erweitert werden, so dass ein solches Skript automatisch aus einer bestehenden Ressourcen-Zuteilung generiert werden kann (Konfiguration abspeichern).

[2] An dieser Stelle zeigt sich einer der wenigen Vorteile statischer Konfigurationsdateien, wie sie z.B. in CLUE-S zum Einsatz kommen.

5.2 Aufbau der XULU-Applikation

Abbildung 5.1 auf Seite 70 zeigt einen Screen-Shot des XULU-Hauptfensters. Jede Komponente der Modellierungsplattform wird darin durch ein eigenes Unterfenster repräsentiert. Hierdurch entsteht der Vorteil, dass einzelne Komponenten (je nach Bedarf des Anwenders) ein- und ausgeblendet werden können. Zudem gestaltet sich die Erweiterung der Plattform sehr einfach, da sich die Integration einer neuen Komponente auf das Hinzufügen eines neuen Unter-Fensters reduziert. Beide Aspekte wären mit einer fest aufgeteilten Oberfläche (z.B. Datenpool oben, Status-Ausgaben unten, sowie Visualisieruns- und Modell-Manager dazwischen) nicht so einfach realisierbar.

Für die Implementierung der GUI werden ausschließlich Klassen des JAVA-Standard eingesetzt. Auf die Details möchte ich an dieser Stelle verzichten, sondern lediglich den strukturellen Aufbau und das Zusammenspiel der Komponenten untereinander darlegen. Abbildung 5.2 auf Seite 71 stellt dies schematisch dar.

Basis für die XULU-Modelling-Platform bildet die Klasse `schmitzm.dipl.xulu.XuluModellingPlattform`[3]. Sie enthält je eine Instanz der Xulu-Komponenten, sowie des Hauptfensters der GUI (linker Teil von Abb. 5.2). Wie in Abschnitt 4.9 beschrieben wurde, teilt sich jede XULU-Komponente in 2 Klassen: Funktionalität und GUI (z.B. Datenpool und Datenpool-Fenster). Die einzelnen Komponenten-Fenster werden durch das Haupt-Fenster verwaltet (rechter Teil von Abb. 5.2).

Die Schnittstellen zwischen dem XULU-Hauptfenster und den einzelnen Komponenten-Fenstern werden durch Interfaces und abstrakte Oberklassen definiert. Die Komponenten-Fenster stellen somit einfach auswechselbare Bestandteile – also in gewisser Weise Plugins – der Plattform-Oberfläche dar. Zur Zeit können sie jedoch nicht dynamisch ausgetauscht werden, sondern werden innerhalb des Hauptfenster-Klasse statisch instanziiert. Da dies an einer zentralen Stelle geschieht, ist trotzdem ein problemloser Austausch der GUI-Komponenten möglich und somit eine einfache Erweiterung der Plattform hinsichtlich Usability.

Zum Ende dieses Abschnitts möchte ich noch darauf hinweisen, dass die XULU-GUI in der Lage ist, verschiedene Sprachen zu unterstützen. Über das *Internationalization*-Konzept des JAVA-Standard kann dies sehr flexibel gestaltet werden [18, 19]. Zur Zeit wird die deutsche und englische Sprache unterstützt. Da das *Java-Internationalization*-Konzept auf einer Hierarchisierung von Klassennamen basiert, ist eine Erweiterung von XULU auf weitere Sprachen *ohne* jegliche Anpassungen an der XULU-GUI selbst möglich.

[3] Wenn im Folgenden Klassenbezeichnungen angeführt werden, verzichte ich aus Gründen der Übersicht auf die exakte Bezeichnung `package.class` und beschränke mich auf die Angabe des Klassennamens. Die genaue Strukturierung der Klassen in Packages kann der *JavaDoc* entnommen werden [41].

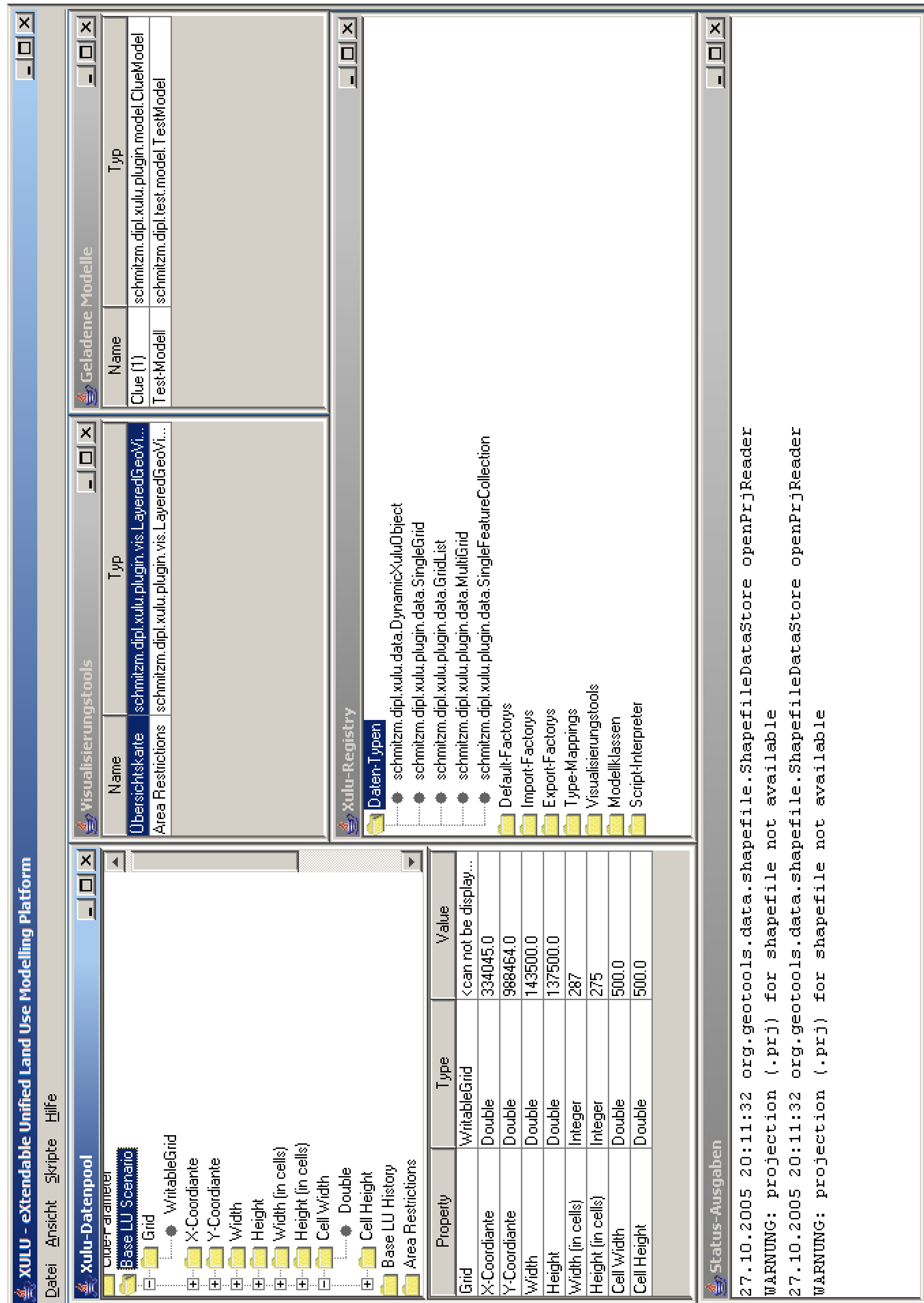


Abbildung 5.1: Das Hauptfenster der XULU-Modelling-Plattform enthält für jede Komponente ein eigenes Unter-Fenster.

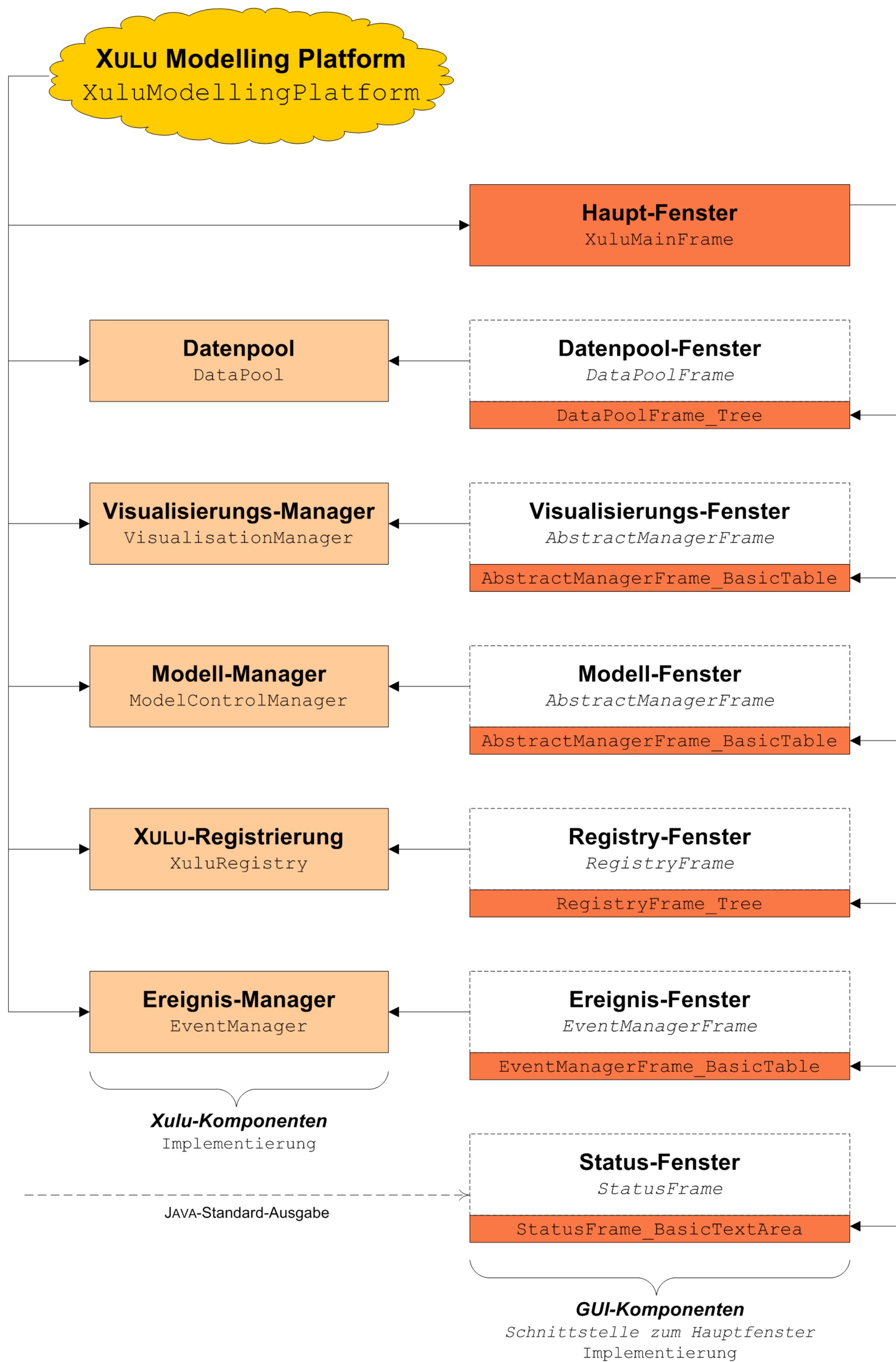


Abbildung 5.2: Der Aufbau der Xulu-Modelling-Platform. Die farbig hinterlegten Klassen stellen instanziierbare Komponenten dar; die gestrichelt umrandeten die Schnittstellen (Interfaces, abstrakte Klassen).

5.3 Plugin-Verwaltung durch die XULU-Registrierung

Die XULU-Registry verwaltet zentral sämtliche dynamisch in die Plattform eingebetteten Komponenten. Dabei unterscheidet XULU zwischen zwei Arten von Plugins:

A) Plugins, von denen in der gesamten XULU-Applikation nur *eine einzige* Instanz benötigt wird

B) Plugins, von denen während des Programmablaufs u.U. *mehrere* Instanzen erzeugt werden müssen

Plugins vom Typ A werden von der Registry *unmittelbar* erzeugt (einmal). Während des Programmablaufs wird diese eine Instanz immer wiederverwendet. Zu diesem Plugin-Typ gehören zum Beispiel die Factorys, die ein Objekt für den Datenpool erzeugen (oder exportieren). Es wäre unnötig, für jede Objekt-Erzeugung eine neue Factory-Instanz zu kreieren.

Für Typ-B-Plugins verwaltet die Registry hingegen nur die entsprechende Klasse und stellt deren *Primär-Funktionalität*[4] sicher. Die eigentliche Plugin-Instanziierung erfolgt erst zu dem Zeitpunkt, an dem das Plugin benötigt wird. Ein Beispiel hierfür sind die Visualisierungstools. Hiervon können mehrere Instanzen benötigt werden, je eine für jedes vom Anwender geöffnete Darstellungsfenster. Die Erzeugung eines solchen Fensters nimmt die Registry erst vor, wenn der Visualisierungs-Manager (bzw. der Anwender) ein neues Fenster anfordert.

Tabelle 5.1 zeigt eine Übersicht der von der XULU-Registry verwalteten Plugins. Wie bereits in den Abschnitten 4.4.5 und 4.7 angedeutet wird, sind die erläuterten Konzepte zur Modell-Synchronisation und Daten-Manipulation noch nicht in XULU umgesetzt. Deshalb verwaltet die XULU-Registry auch noch keine entsprechenden Plugins. Die Implementierung der Registry ist jedoch auf die Integration neuer Plugin-Typen

Plugin	**Typ**	**Struktur in XULU-Registry**
Datentypen	B	Klassen von `XuluObject`
Factorys zur Objekterzeugung	A	Instanzen von `InstantiationFactory`
Import-Factorys	A	Instanzen von `ImportFactory`
Export-Factorys	A	Instanzen von `ExportFactory`
Visualisierungstools	B	Klassen von `VisualisationTool`
Modelle	B	Klassen von `XuluModel`
Skript-Interpreter	A	Instanzen von `ScriptInterpreter`

Tabelle 5.1: Plugin-Komponenten, die von der XULU-Registry dynamisch (in Listen) verwaltet werden

[4] Existenz der Klasse und Eignung für die Plugin-Funktionalität (z.B. Implementierung bestimmter Interfaces).

ausgerichtet, so dass die Umsetzung dieser Konzepte keinen wesentlichen Anpassungsaufwand an der Registry verursacht.

Der Inhalt der Registry ist in einer statischen Konfigurationsdatei `registry.xif` hinterlegt, die sich im XULU-Programmverzeichnis befindet. Sie wird automatisch beim Programmstart eingelesen. Dabei wurde Wert darauf gelegt, dass die Registry-Datei für den Anwender leicht – und ohne Vorkenntnisse (z.B. von XML) – modifiziert werden kann. Die aktuelle XULU-Implementierung verwendet deshalb ein einfaches und intuitives ASCII-Format, so dass die Datei durch einen beliebigen Text-Editor veränderbar ist. Eine Beispiel-Datei ist in Anhang A zu finden.

In der Registry-Datei wird jedes Plugin durch einen vollständigen JAVA-Klassennamen identifiziert. Die Instanziierung eines Plugins auf Basis dieses Names wird über die *Reflection API* des JAVA-Standard realisiert [11]. Ein JAVA-Vorteil wird an dieser Stelle allerdings zum Nachteil für die XULU-Plugin-Verwaltung: Um Objekte während des Programmablaufs schneller erzeugen zu können, cached die *Java Virtual Machine* (JVM) – in der *jedes* JAVA-Programm ausgeführt wird – beim Programmstart sämtliche benötigten Klassen[5]. Wie Abschnitt 5.1 bereits andeutet, ist deshalb ein kompletter Neustart der XULU-Applikation notwendig, damit Änderungen an einem Plugin (insbesondere an einem Modell) wirksam werden. Mittlerweile konnte ich zwar eine prinzipielle Vorgehensweise ermitteln, Klassen dynamisch von außerhalb der aktiven JVM neu- bzw. nachzuladen, diese ist jedoch noch nicht in XULU implementiert [14, Kap. 2.17.8].

Neben den in Tabelle 5.1 aufgeführten Plugins, verwaltet die XULU-Registry noch sog. *Type-Mappings*. Dabei handelt es sich um die in Abschnitt 4.8.1 als *Tupel* bezeichnete Zuordnung zwischen registrierten Datentypen und Factorys. Jedem Datentyp muss eine Instanziierungs-Factory zugeordnet sein, über die eine (leere) Standard-Instanz des Datentyps erstellt wird. Darüberhinaus kann ein Type-Mapping beliebig viele Import- und Export-Factorys enthalten. Die Type-Mappings ermöglichen es beispielsweise, in einer GUI Datentyp-spezifische Menüs oder Dialoge zur erstellen, in denen nur die Import/Export-Routinen erscheinen, die für ein angewähltes Datenobjekt zuständig sind.

[5] Details zur JVM können [14] entnommen werden.

5.4 Event-Management

Die Implementierung des XULU-Event-Managements beruht auf dem in Abbildung 5.3 dargestellten *Observer Pattern* [9, 4]. Ich möchte dies jedoch über das in JAVA übliche *Event-Listener*-Konzept erläutern:

- **Listener:**
  Ein Listener ist ein Objekt, das auf bestimmte Ereignisse eines anderen Objekts „lauscht“, um darauf zu reagieren. Hierzu *koppelt* er sich an das zu überwachende Objekt an.

- **Event (Ereignis):**
  Ein Objekt, das Ereignisse auslöst, stellt Methoden bereit, damit sich Listener ankoppeln können. Die angekoppelten Listener verwaltet das Objekt z.B. in einer Liste. Ein Ereignis wird ausgelöst, indem alle angekoppelten Listener (über eine fest spezifizierte Methode des Listeners) das Event mitgeteilt bekommen. Die Listener entscheiden daraufhin, ob und ggf. wie sie auf dieses Ereignis reagieren.

Um im XULU-Event-Manager die in 4.3 beschriebene Flexibilität zwischen Ereignis und Reaktion zu erreichen, wird der oben beschriebene Listener in zwei Komponenten aufgespalten: *Handler* und *Event-Handler*. Die resultierenden 3 Komponenten sind in Abbildung 5.4 dargestellt.

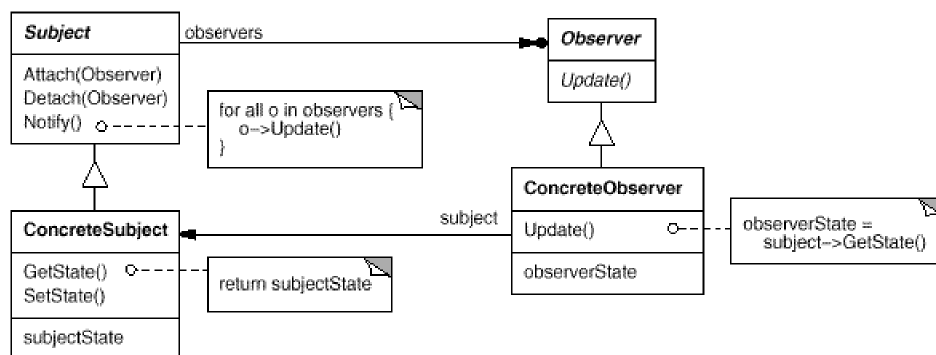


Abbildung 5.3: Das *Observer Pattern* (Quelle: [9])

5.4.1 Event

In XULU besteht ein Event-Objekt aus zwei Teilen:

a) Ein Ereignis-Typ (z.B. „Objekt geschlossen“ oder „Modellschritt beendet“)

b) Ein Auslöser-Objekt

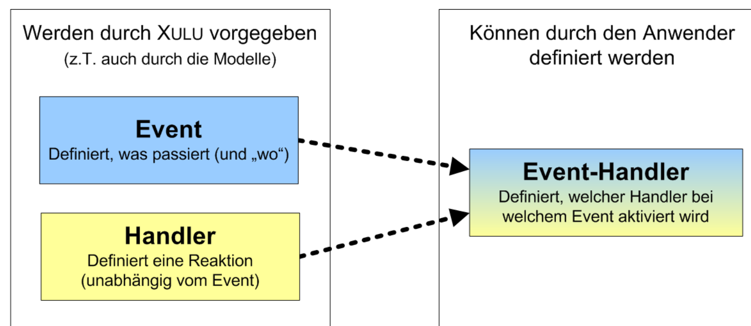


Abbildung 5.4: Ein Event-Handler verbindet ein Ereignis mit einer davon unabhängigen Funktion zu einer Reaktion.

Bestimmte (Spezial)-Ereignisse können noch weitere Eigenschaften besitzen. Zum Beispiel enthält ein *ModelStepFinished*-Event die Information, *welcher* Zeitschritt beendet wurde. Für die Ereignis-Behandlung im Event-Manager sind jedoch nur die oben genannten Informationen relevant.

5.4.2 Handler

Ein Handler beschreibt eine konkrete Reaktion auf ein Ereignis. Diese ist jedoch *unabhängig* von einem bestimmten Ereignis oder Ereignis-Typ definiert. Deshalb kann ein Handler in XULU prinzipiell „alles“ sein, z.B. auch die Löschung eines Objekts aus dem Datenpool oder das Beenden der XULU-Applikation. Aus diesem Grund gehört zu einer Handler-Implementierung immer eine Handler-Factory (mit entsprechender GUI), über die der Anwender das Neu-Erstellen und Abändern eines Handlers vornehmen kann. Zur Zeit werden die Handler (mit entsprechenden Factorys) statisch in den XULU-Event-Manager integriert. An dieser Stelle kann XULU jedoch sinnvoll erweitert werden, indem die Handler in die Plugin-Verwaltung (vgl. 5.3) aufgenommen werden.

Exemplarisch wurde in XULU ein Handler zur Objekt-Visualisierung implementiert. Dieser setzt sich zusammen aus ...

- Einem Objekt aus dem Datenpool
- Einer Visualisierungstool-Instanz (Fenster)

Bei Ausführung des Handlers fordert dieser das Visualisierungstool auf, das angegebene Objekt zu aktualisieren. Wird das Objekt im Visualisierungstool noch nicht dargestellt, fügt es der Handler hinzu.

Für die Modellierung wäre darüberhinaus ein Handler wünschenswert, der ein Objekt des Datenpools über eine bestimmte Factory (in eine Datei) exportiert. Durch einen entsprechenden Event-Handler (vgl. 5.4.3) wäre der Anwender in der Lage, zu bestimmten Zeitpunkten (z.B. am Ende jedes Zeitschritts) Zwischen-Ergebnisse für spätere Analysen zu sichern, obwohl sie im weiteren Modell-Verlauf modell-intern überschrieben werden.

5.4.3 Event-Handler

Wie Abbildung 5.4 zeigt, definiert ein Event-Handler, welche Reaktion auf ein bestimmtes Ereignis zu erfolgen hat. Diese Fragestellung wird durch die folgenden drei Eigenschaften eines Event-Handler-Objekts vollständig spezifiziert:

a) Ein Auslöser-Objekt, das beobachtet wird.

b) Ein Ereignis-Typ, auf den reagiert wird.

c) Ein Handler, mit dem reagiert wird.

Über das User Interface des Event-Managers kann der Anwender eigene Event-Handler spezifizieren. Hierzu muss er zunächst ein Objekt des Datenpools oder ein (geladenes) Modell auswählen, das „beobachtet“ werden soll. Anschließend erhält er vom Event-Manager eine Auswahl, welche Events hierfür zur Verfügung stehen (Tabelle 5.2) und mit welchen Handlern auf das Ereignis reagiert werden kann.

Objekt-Ereignisse	Modell-Ereignisse
• „Objekt verändert“ • „Objekt geschlossen“	• „Modell initialisiert“ • „Modell gestartet“ • „Modell beendet“ • „Modell zerstört“ + *modellspezifische Ereignisse, wie z.B. „Zeitschritt begonnen“*

Tabelle 5.2: Für benutzerdefinierte Event-Handler zur Verfügung stehende Ereignisse

5.4.4 Ablauf der Event-Managers

Zunächst registriert sich der Event-Manager als Listener an den Datenpool, sowie an den Modell-Manager, um darüber informiert zu werden, sobald ein neues Objekt im Datenpool erzeugt wird und sobald der Anwender ein Modell lädt. Tritt ein solches Ereignis ein, koppelt sich der Event-Manager unmittelbar an das neue Objekt (bzw. Modell) an[6]. Somit wird der Event-Manager in der Folge über sämtliche Ereignisse *aller* Objekte, bzw. Modelle informiert, unabhängig davon, ob der Anwender bereits einen entsprechenden Event-Handler definiert hat. Der Grund dafür liegt darin, dass der Event-Manager auch interne Event-Handler verwaltet (nicht durch den Anwender erzeugt), die *für alle* Datenpool-Objekte oder Modelle gelten. Ein Beispiel hierfür wurde bereits in Abschnitt 4.3 angeführt:

[6] Prinzipiell könnte diese Vorgehensweise auch über die Definition eines speziellen Event-Handlers geschehen. Aus Gründen der Übersichtlichkeit wurde dieser Weg jedoch nicht verfolgt.

Wird ein Objekt aus dem Datenpool gelöscht, werden automatisch alle Visualisierungen dieses Objekts entfernt.

Um solche Event-Handler zu verarbeiten, muss sich der Event-Manager ohnehin an alle Objekte ankoppeln.

Für jedes eintreffende Ereignis durchsucht der Event-Manager sämtliche gespeicherten Event-Handler (anwender-definierte und interne) und führt für alle, zum Ereignis passenden, den zugehörigen Handler aus (Abb. 5.5). Ein Ereignis passt zu einem Event-Handler, wenn Event-Typ und -Auslöser übereinstimmen. Das Auffinden der Event-Handler wird dabei durch die Verwendung von *Hash-Tabellen* beschleunigt.

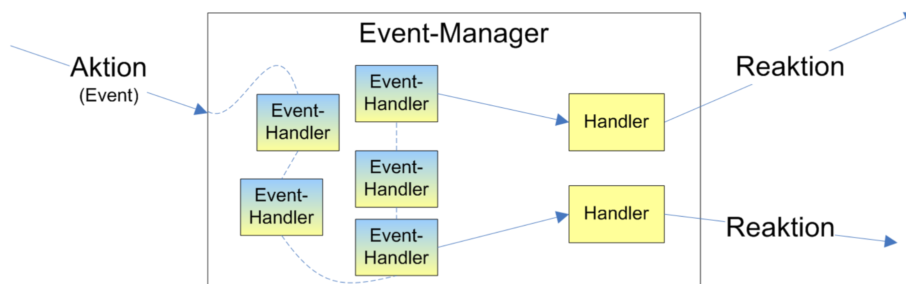


Abbildung 5.5: Das Eintreffen eines Ereignisses im Event-Manager.

Abbildung 5.6 zeigt die wesentlichen internen Abläufe des Event-Managers als SDL-Diagramm[7]. Zu beachten ist, dass Handler, die auf ein Modell-Ereignis hin ausgeführt werden, als Bestandteil des Modell-Prozesses ausgeführt werden. Somit blockiert ein Modell während der Ausführung eines Handlers. Dies ist notwendig, um Daten-Konsistenz für den Handler zu gewährleisten. Während der Handler zum Beispiel ein Objekt visualisiert oder exportiert, darf es vom Modell nicht verändert werden.

[7] Anhang G zeigt eine Übersicht, über die verwendeten Symbole. Weitere Informationen zu SDL können [17, Kap. 2] und [28] entnommen werden.

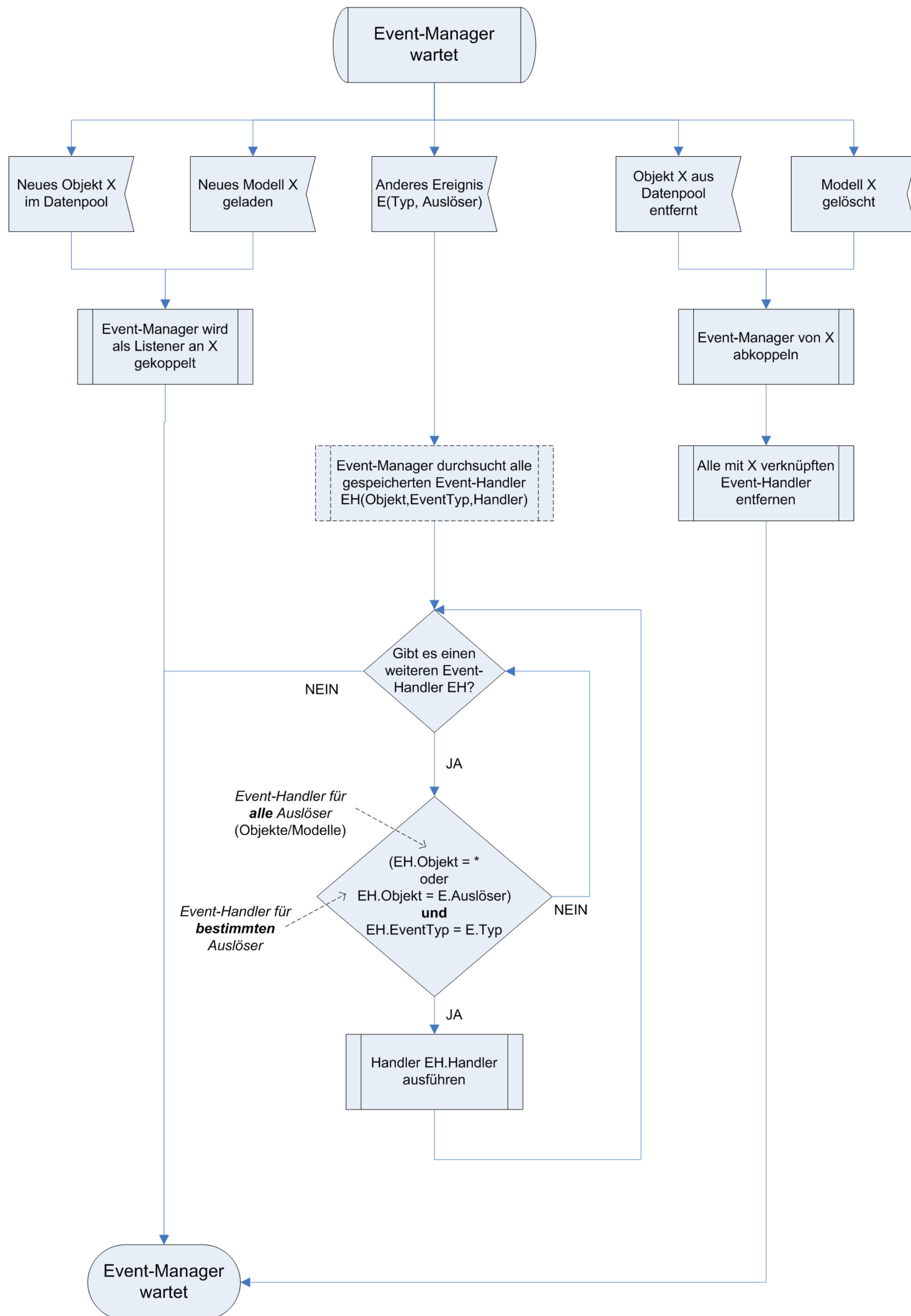


Abbildung 5.6: Der interne Ablauf des Event-Managers (ohne die Definition neuer Event-Handler).

5.5 Datenverwaltung

Auf die Implementierung der Datenpool-Komponente möchte ich an dieser Stelle nicht näher eingehen, da sie im Wesentlichen realisiert wurde, wie in Abschnitt 4.6 entworfen. Über das User Interface des Datenpools hat der Anwender die Möglichkeit, XULU-Objekte ...

... zu importieren oder zu exportieren (hierzu werden dem Anwender alle registrierten Factorys angeboten).

... zu löschen

... zu visualisieren (hierzu werden dem Anwender alle registrierten Visualisierungstools angeboten, sowie alle bereits erzeugten Visualisierungstool-Fenster)

... zu modifizieren (nur für Basis-Datentypen möglich, da Tools zur interaktiven Manipulation komplexer Objekte noch nicht in XULU integriert sind)

... komplett neu zu erzeugen (hierzu werden dem Anwender alle registrierten Datentypen angeboten)

... strukturell zu kopieren (also das aktuell angewählte Objekt als Vorlage für ein neues zu verwenden, vgl. 5.5.2)

5.5.1 Datentypen

Im Prinzip können im XULU-Datenpool beliebige Objekte verwaltet werden. Als Datentypen können jedoch trotzdem nicht beliebige JAVA-Klassen verwendet werden. Wie Abschnitt 4.8.3 zusammenfassend darstellt, muss jedes XULU-Objekt gewisse Anforderungen erfüllen, um innerhalb der Plattform verwaltet werden zu können. Zur Spezifikation dieser Anforderungen dient das JAVA-Interface `XuluObject`. Durch dessen Implementierung kann jede existierende Objekt-Klasse zu einem XULU-Objekt erweitert und somit in XULU verwendet werden. Damit beim Neu-Entwurf von Datentypen jedoch nicht sämtliche Funktionen neu implementiert werden müssen (z.B. die Verwaltung eines Namens und einer ID), stellt XULU abstrakte Klassen bereit, von denen die wesentlichen Methoden des XULU-Objekts geerbt werden können.

Da JAVA jedoch nur Einfach-Vererbung erlaubt[8], kann eine Mehrfach-Implementierung der `XuluObject`-Schnittstelle jedoch nicht gänzlich vermieden werden. Hierzu kommt es, wenn eine bestehende Objekt-Klasse (z.B. ein Raster aus einer existierenden Klassen-Bibliothek) zu einem XULU-Objekt werden soll (siehe Abb. 5.7).

[8] Eine JAVA-Klasse kann zwar mehrere Interfaces implementieren, jedoch nur eine Oberklasse besitzen, von der sie Methoden/Eigenschaften erbt.

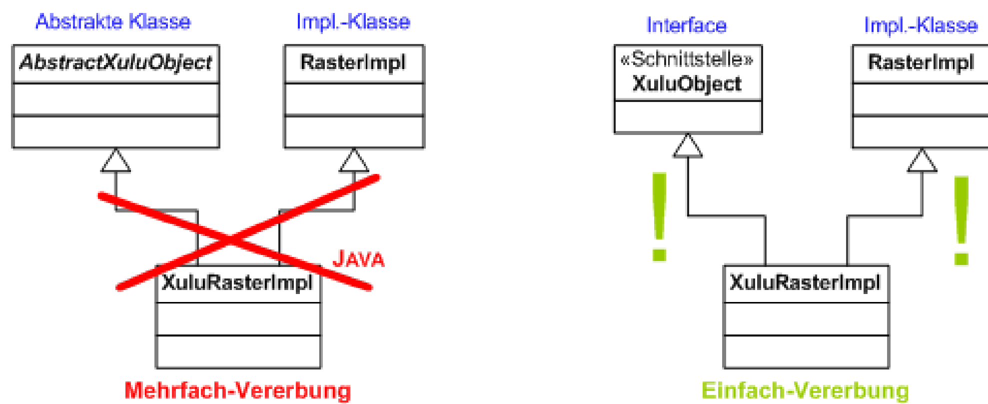


Abbildung 5.7: Ableiten von *mehreren* Klassen-Implementierungen ist in JAVA nicht möglich (links); von *einer* Klasse-Implementierung und mehreren Interfaces schon (rechts).

Das Property-Konzept

Im Rahmen der Implementierung stellen sich zwei wesentliche Fragen:

a) Wie kann die Plattform dem Anwender (in einer generischen GUI) Auskunft über die Struktur eines Objekts geben, ohne konkrete Informationen über dessen internen Aufbau zu haben?

b) Wie lassen sich Änderungspropagierung und Zugriffskontrolle zentral realisieren?

Beide Punkte werden durch eine weitere Anforderung an die XULU-Objekte realisiert, die der Entwurf der XULU-Datentypen (vgl. 4.8.3) noch nicht vorsieht:

Jedes XULU-*Objekt setzt sich aus Eigenschaften (Propertys) zusammen.*

Dabei besteht eine allgemeine Eigenschaft (Property) aus:

Einem Namen (`String`)	Identifiziert die Property *eindeutig* innerhalb einer Menge von Eigenschaften.
Einem Typ (`Class`)	Die Eigenschaft kann nur Instanzen dieser Klasse als Wert(e) aufnehmen.
Wert(en)	Repräsentiert die aktuelle Ausprägung der Eigenschaft.

Punkt a) lässt sich somit sehr einfach dadurch realisieren, dass jedes XULU-Objekt Methoden bereitstellen muss, die Auskunft über seine Eigenschaften geben.
Die Umsetzung von Punkt b) basiert darauf, dass XULU eine Reihe allgemeiner Property-Klassen vordefiniert:

- **ScalarProperty**
  Kann *einen* Wert (Basis-Datentyp oder Objekt) aufnehmen.
- **ListProperty**
  Kann *mehrere* gleichartige Werte (Basis-Datentypen oder Objekte) aufnehmen und ist dynamisch erweiterbar (hinzufügen/entfernen).
- **MatrixProperty**
  Kann *mehrere* gleichartige Werte (Basis-Datentypen oder Objekte) aufnehmen. Dimension und Größe der Matrix werden bei der Instanziierung festgelegt und sind in der Folge fix.

Die dargestellen Eigenschaften stellen keine speziellen Bedingungen an die in ihnen gespeicherten Werte. Sie können somit flexibel in vielen Datentypen eingesetzt werden. Da der Zugriff auf die Property-Werte nur über *fest definierte* Methoden erlaubt wird, kann die Zugriffskontrolle (Zugriff auf die Werte nur über Lese- und Schreibrechte), sowie die Änderungspropagierung (Ereignis bei jedem Schreibzugriff) zentral implementiert werden und ist nicht für jeden Datentyp neu zu realisieren. Ein Datentyp muss lediglich Instanzen der oben genannten Propertys erzeugen.

Die Abbildungen 5.8 und 5.9 verdeutlichen nochmals die Auslagerung der Zugriffskontrolle und Änderungspropagierung in die Propertys, aus denen sich jedes XULU-Objekt zusammen setzt.

Dynamische XULU-Objekte

Durch das Property-Konzept ergibt sich ein weiterer Vorteil für die XULU-Datenverwaltung. Neben *statischen* (fest programmierten) XULU-Objekttypen, die aus unveränderbaren[9] Eigenschaften bestehen, können auch *dynamisch* neue Objekttypen erstellt werden, ohne explizit neue Klassen zu programmieren. Hierzu wurde ein spezieller XULU-Objekttyp `DynamicXuluObject` implementiert (vgl. Abb. 5.8), dessen Instanzen zur Laufzeit dynamisch um Propertys erweitert und reduziert werden können.

Angedacht wurde dieser Datentyp, um dem Anwender die Möglichkeit zu geben, über einen „Baukasten“ in der XULU-Applikation – also ohne die Programmierung neuer Klassen – eigene Datentypen zusammenzusetzen. Die Realisierung eines entsprechenden interaktiven User Interface gestaltet sich jedoch als zu umfangreich, um im Rahmen dieser Diplomarbeit umgesetzt zu werden. Die Klasse `DynamicXuluObject`

[9] „Unveränderbar“ bezieht sich auf Namen und Typ der Property. Deren Werte können natürlich verändert werden!

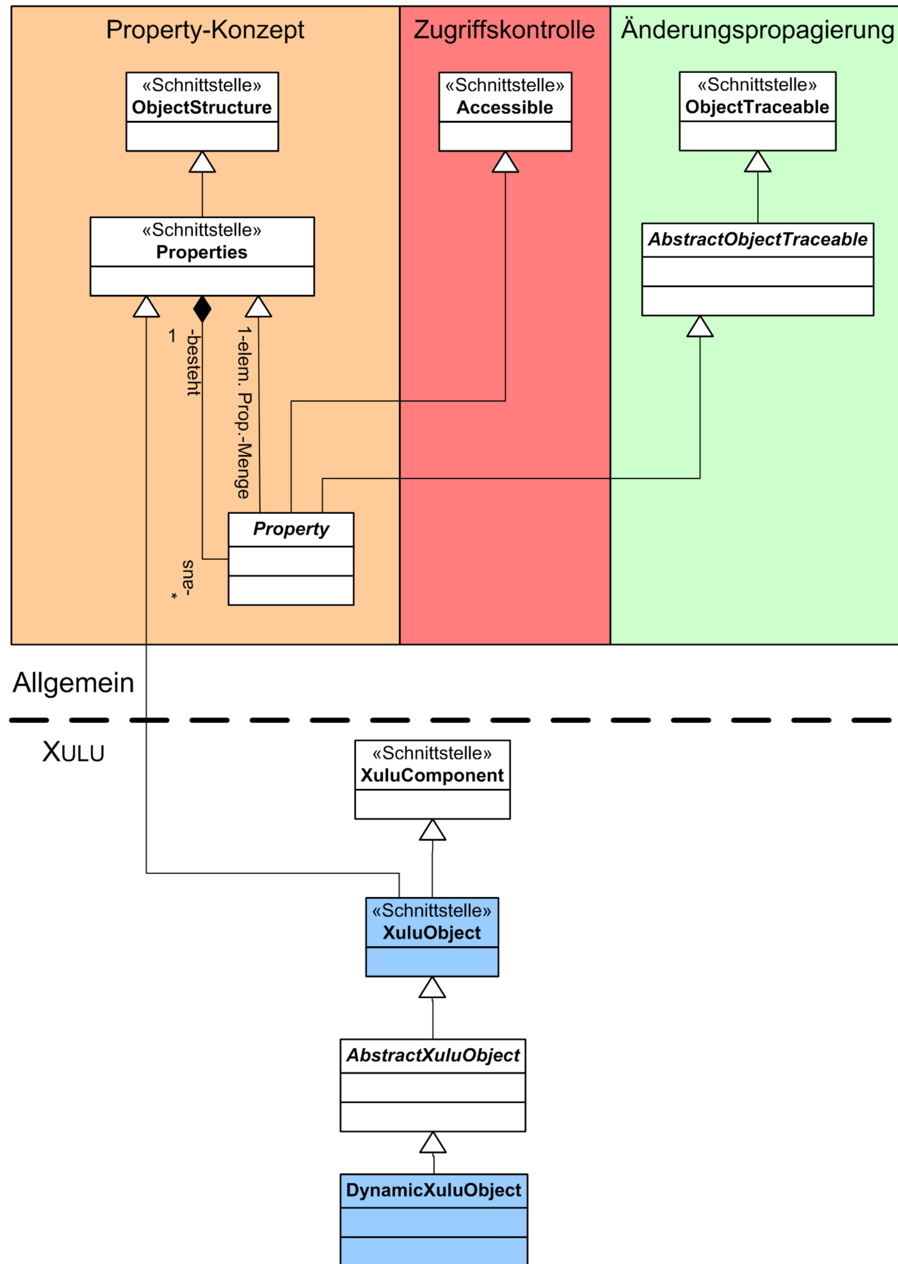


Abbildung 5.8: Der Aufbau eines XULU-Objekts.

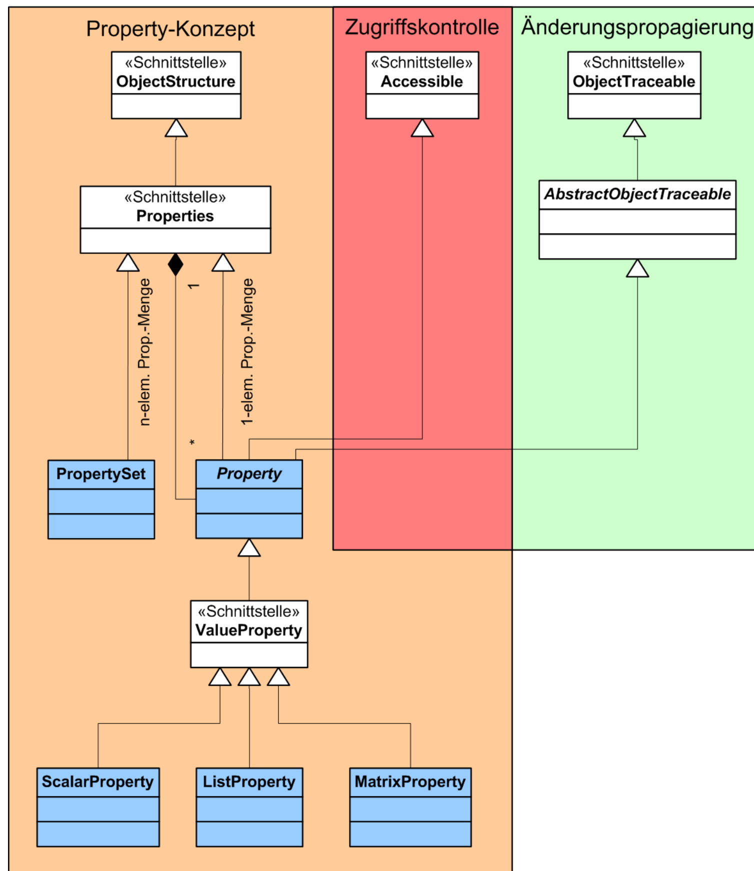


Abbildung 5.9: Der Aufbau einer *Property* (Eigenschaft) eines XULU-Objekts.

stellt aber bereits die Basis dar, um in weiterführenden Projekten eine solche Baukasten-Komponente in XULU zu integrieren. Zur Zeit ist lediglich eine Import-Factory implementiert, die ein *dynamisches* XULU-*Objekt* aus einer einfachen ASCII-Datei erstellt (siehe Anhang D). Dabei können Skalare, Listen und Matrizen von Basisdatentypen (z.B. String, Integer, Double) als Objekt-Propertys definiert werden. Im Rahmen der CLUE-Implemenierung (vgl. Kapitel 6) wird dieses Dateiformat eingesetzt, um alle Modell-Parameter in einer zentralen Datei zu hinterlegen und im Datenpool als *ein* Objekt zu behandeln.

5.5.2 Objekt-Erzeugung durch Factorys

Wie in Abschnitt 4.8.1 beschrieben, erfolgt die Instanziierung von Datenpool-Objekten ausschließlich über sog. *Factorys*, die beim Programm-Start in der XULU-Registry eingetragen werden. Durch die Auswahl der Factory entscheidet der Anwender, welches Objekt er wie (z.B. welches Dateiformat) erzeugen/verarbeiten möchte. XULU unterscheidet drei Arten von Factorys:

a) **`InstantiationFactory`**
Erzeugt eine Standard-Instanz eines Objekts.

b) **`ImportFactory`**
Liest eine Objekt-Instanz aus einer Eingabe-Quelle.

c) **`ExportFactory`**
Schreibt eine Objekt-Instanz in eine Ausgabe.

Das in Abschnitt 4.8.1 beschriebene Konzept wurde im Rahmen der XULU-Implementierung in einigen Punkten erweitert.

Die **Instanziierungs-Factorys** bietet neben der Erzeugung einer „leeren“ Standard-Instanz auch die Möglichkeit, ein XULU-Objekt auf Basis *einer Vorlage* zu instanziieren. Hierbei soll die Factory jedoch keine 1:1-Kopie der Vorlage erstellen, sondern lediglich die Objekt-Struktur übernehmen, z.B. für ein Raster die Größe und die Geo-Position (nicht aber die einzelnen Zell-Werte!). Die Raster-Factory bietet dabei dem Anwender die Möglichkeit – über einen Dialog – Änderungen an den Vorlagen-Daten vorzunehmen (Abb. 6.5). Um sie auch in automatisierten Abläufen (z.B. Skripte, vgl. 5.1) einsetzen zu können, müssen die Factorys jedoch in der Lage sein, ein Objekt auch ohne jegliche Zusatzinformation zu instanziieren.

Für die **Import- und Export-Factorys** wird in Abschnitt 4.8.1 angeführt, dass die Factorys selbst dafür verantwortlich sind, über Anwender-Dialoge Informationen einzuholen, aus welcher Quelle (bzw. in welches Ziel) der Import/Export erfolgen soll (z.B. Datei oder Datenbank). Dieses Konzept wird in der XULU-Implementierung aus Gründen der Anwenderfreundlichkeit nicht vollständig umgesetzt. Betrachtet man

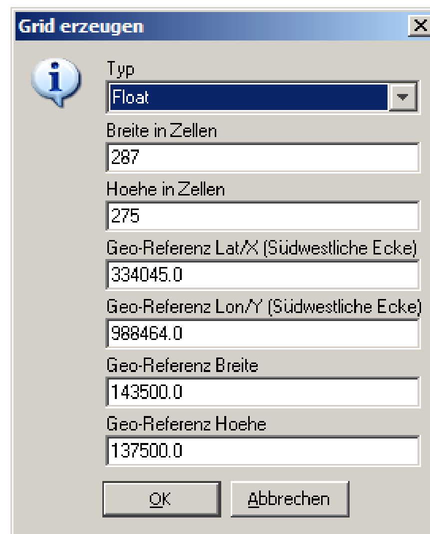


Abbildung 5.10: Um eines neues Raster zu erzeugen, müssen eine Reihe von Parametern spezifiziert werden.

zum Beispiel den Datenimport aus einer Datenbank, so würde aus dem beschriebenen Entwurf resultieren, dass die Verwaltung einer Datenbank-Verbindung innerhalb der Import-Factory zu realisieren ist. Die Abfrage der benötigten Informationen (z.B. Datenbank-Server, User, Passwort, usw.) müsste also für *jeden* einzelnen Import-Vorgang erfolgen. Anwenderfreundlicher ist es, eine Datenbank-Verbindung global innerhalb der XULU-Plattform zu verwalten. Eine bestehende Verbindung kann so für viele Import- und Export-Vorgänge genutzt werden.

Die Schnittstelle zwischen Import-Factory und XULU-Plattform sieht deshalb vor, dass die Factory spezifizieren muss, welche Art von Objekt sie für den Import-Vorgang als Eingabe benötigt (z.B. eine Datei oder eine Datenbank-Verbindung). XULU sorgt daraufhin für die Bereitstellung eines solchen Quell-Objekts, indem es dem Anwender z.B. einen Dateiauswahl-Dialog anzeigt. Das von der Plattform ermittelte Quell-Objekt wird der Factory anschließend zum Daten-Import zur Verfügung gestellt. Abbildung 5.11 zeigt diesen Ablauf in Form eines SDL-Diagramms[10]. Der Export-Vorgang erfolgt weitestgehend analog, mit dem Unterschied, dass die Factory der Plattform den Typ des Export-Ziels mitteilt (anstelle der Quelle).

Zur Zeit bietet die XULU-Plattform noch keine Datenbank-Verbindung als Quell-Objekt an, sondern lediglich die Möglichkeit des Datei-Imports und -Exports. Das dargelegte Konzept berücksichtigt jedoch, dass eine solche Erweiterung ohne die Umgestaltung bestehender Schnittstellen möglich ist.

[10] Anhang G zeigt eine Übersicht, über die verwendeten Symbole. Weitere Informationen zu SDL können [17, Kap. 2] und [28] entnommen werden.

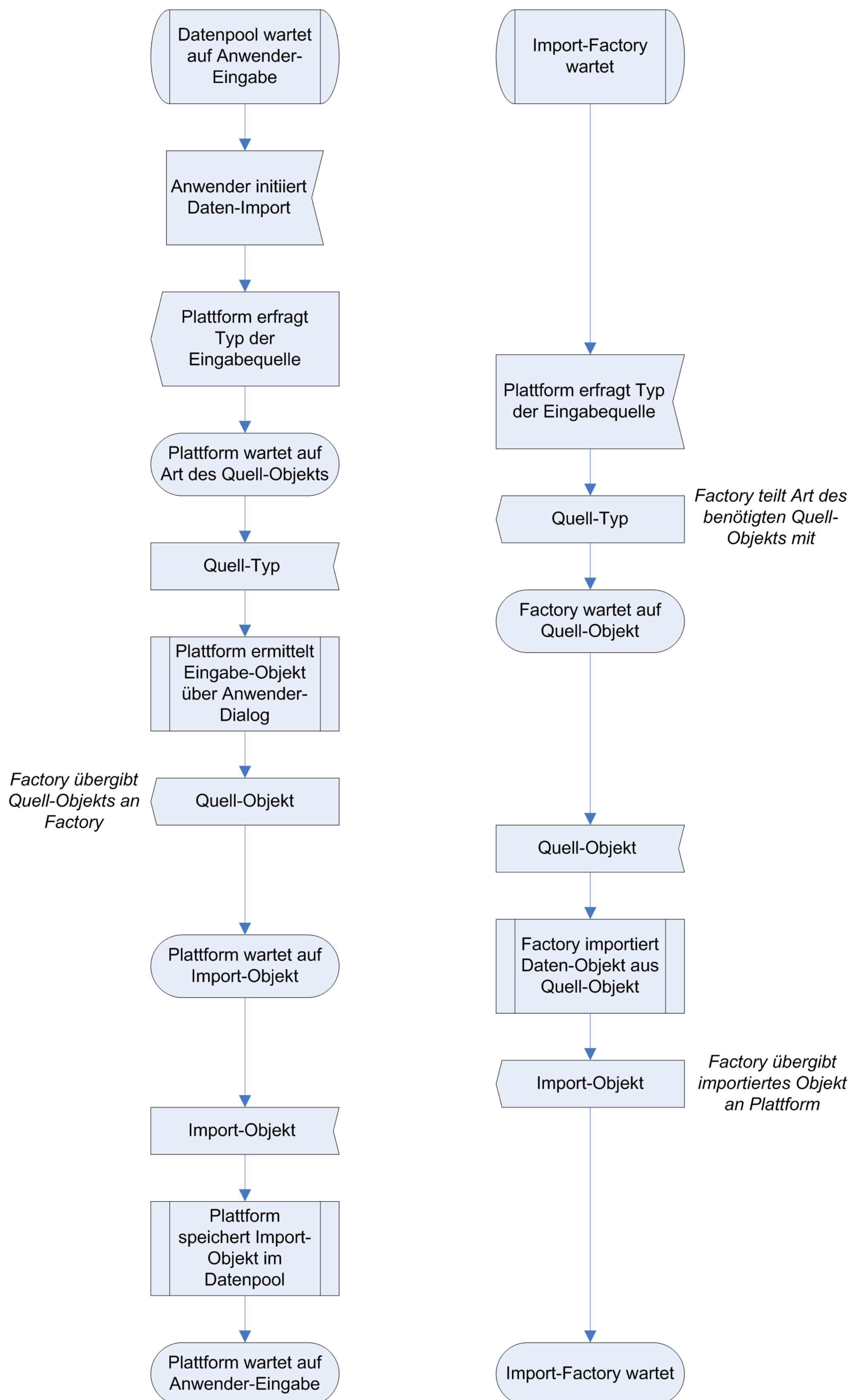


Abbildung 5.11: Der Ablauf eines Datenimport-Vorgangs als SDL-Diagramm.

5.5.3 Der Datentyp Raster

Für die Landnutzungsmodellierung, die dieser Arbeit als Anwendungsbeispiel dient, ist es sehr wichtig, auf geographische Daten zugreifen zu können. Hierzu wurden XULU-Objekte (Datentypen) zur Verwaltung von Raster- und Vektor-Daten implementiert. Exemplarisch möchte ich in diesem Abschnitt den Datentyp „Raster“ vorstellen.
Wie in Abschnitt 4.8.1 angedeutet, gliedert sich die Implementierung in drei Teile:

1. **Datendefinition:**
   Spezifikation einer Schnittstelle für alle Raster-Implementierungen (z.B. *Array* oder *QuadTree*).
2. **Datenorganisation:**
   Implementierung eines XULU-Datentyps *Raster*, entsprechend der Spezifikation.
3. **Datenerzeugung:**
   Implementierung von Factorys, welche die konkrete Raster-Implementierung erzeugen, importieren und exportieren.

Datendefinition

Die Spezifikation eines Rasters erfolgt in XULU durch das JAVA-Interface `WritableGrid`. Dieses garantiert, dass jede Raster-Implementierung die gleichen Methoden besitzt, mit denen auf die Höhe, Breite, Geo-Position, das Zellformat, sowie die einzelnen Zell-Werte zugegriffen werden kann. Darüberhinaus schreibt es Methoden vor, welche die Umrechnung von Raster-Koordinaten (Zellnummern) in Geo-Koordinaten (Latitude/Longitude) ermöglichen.

Datenorganisation

Der XULU-Datentyp zur Speicherung eines einzelnen Rasters heißt `SingleGrid`. Tabelle 5.3 zeigt dessen Aufbau in Propertys. Wie daraus ersichtlich wird, enthält das `SingleGrid` keine Matrix-Property für den Zugriff auf die einzelnen Raster-Zellen, sondern eine skalare Eigenschaft „Grid“, die Zugriff auf ein komplettes `WritableGrid`-Objekt (siehe oben) gewährt.
Für diese Design-Entscheidung gibt es zwei Gründe:

- Für die aktuelle Raster-Implementierung wird auf einer *bestehenden* GEOTOOLS-Klasse aufgebaut [10]. Würde der XULU-Objekttyp direkt von dieser Klasse abgeleitet, wären sämtliche Eigenschaften des XULU-Objekts ein weiteres mal zu implementieren und könnten nicht von einer abstrakten Oberklasse geerbt werden (JAVA-Restriktion der Einfach-Vererbung, vgl. 5.5.1). Um dieses Problem zu umgehen, stellt der XULU-Objekttyp `SingleGrid` nicht selbst ein Raster dar, sondern enthält ein solches Objekt als Property.

- Neben dem `SingleGrid` wurden XULU-Datentypen zur Speicherung *mehrerer* Raster erstellt (vgl. unten). Diese sind ähnlich zum `SingleGrid` aufgebaut, mit dem Unterschied, dass sie statt der skalaren Eigenschaft „Grid“ eine List-Property verwenden. Für neue Formen der Raster-Darstellung (z.B. als Quad-Tree, vgl. Beispiel in 4.8.1) müssen all diese XULU-Datentypen nicht neu erstellt werden, sondern es genügt, dass entsprechende Factorys die „Grid“-Eigenschaft des XULU-Objekts mit der alternativen Raster-Implementierung befüllen.

Neben dem `SingleGrid` wurden in XULU auch Datentypen zur Speicherung *mehrerer* Raster implementiert. Diese werden zum Beispiel für die Repräsentation eines dynamischen Driving Force verwendet (vgl. 2.3.3), um neben dessen aktueller Ausprägung, auch seine Veränderung über die Zeit zu speichern (im selben Objekt). Die entsprechenden XULU-Datentypen heißen `GridList` und `MultiGrid` und bauen auf derselben Raster-Spezifikation (`WritableGrid`) auf wie das `SingleGrid`. Beide speichern Raster-Objekte in einer erweiterbaren Liste (List-Property). Der Unterschied zwischen den beiden Datentypen besteht darin, dass `GridList` beliebige Raster und `MultiGrid` nur gleichartige Raster (gleiche Größe, Auflösung und Geo-Position) aufnehmen kann.

Property	**Typ**	**Bedeutung**
X-Coordinate	`ScalarProperty[Double]`	Latitude der Südwestlichen Ecke
Y-Coordinate	`ScalarProperty[Double]`	Longitude der Südwestlichen Ecke
Width	`ScalarProperty[Double]`	Reale Breite des Rasters
Height	`ScalarProperty[Double]`	Reale Höhe des Rasters
Width in Cells	`ScalarProperty[Integer]`	Breite des Rasters in Zellen
Height in Cells	`ScalarProperty[Integer]`	Höhe des Rasters in Zellen
Cell Width	`ScalarProperty[Double]`	Breite einer Rasterzelle
Cell Height	`ScalarProperty[Double]`	Höhe einer Rasterzelle
Grid	`ScalarProperty[WritableGrid]`	Zell-Inhalte

Tabelle 5.3: Die Eigenschaften eines `SingleGrid`. Das `MultiGrid` unterscheidet sich davon nur dadurch, dass es statt der skalaren Property „Grid“ eine Liste von Rastern verwaltet.

Datenerzeugung

Wie bereits angedeutet wurde, bestimmten die Instanziierungs- und Import-Factorys die Art der Raster-Implementierung, die in der „Grid“-Property des `SingleGrid` gespeichert wird[11]. Die zur Zeit in XULU enthaltenen Factorys verwenden eine Implementierung der existierenden GEOTOOLS-Klassenbibliothek [10].

Für den Daten-Import und -Export unterstützen die Factorys das Dateiformat ARCINFOASCIIGRID (siehe Anhang C). Hierzu werden ebenfalls bestehende GEOTOOLS-Routinen eingesetzt. Eine neuere GEOTOOLS-Version enthält darüberhinaus auch I/O-Routinen für andere Dateiformate (z.B. GEOTIFF). Entsprechende Import- und Export-Factorys sind in XULU jedoch noch nicht implementiert.

Für den Import eines `SingleGrid` muss die XULU-Plattform der Factory eine einzelne Datei als Quell-Objekt zur Verfügung stellen (vgl. 5.5.2), die Factorys für `GridList` und `MultiGrid` fordern entsprechend mehrere. Hingegen muss der Anwender für den Export immer eine *einzelne* Datei angeben, obwohl alle in einer `GridList` oder einem `MultiGrid` enthaltenen Raster in eine einzelne Datei exportiert werden. Die Export-Factorys der `GridList` und des `MultiGrid` erweitern den angegebenen Dateinamen automatisch um den jeweiligen Listen-Index.

Das Diagramm in Abbildung 5.12 verdeutlicht das Zusammenspiel zwischen den verschiedenen Raster-Objekten (blau) und Raster-Factorys (braun), welche die Implementierung des Datentyps bestimmen (hier GEOTOOLS).

5.6 Modell-Schnittstelle und -Steuerung

Die Kommunikation zwischen Modell und XULU-Plattform erfolgt auf Plattform-Seite ausschließlich mit dem XULU-Modell-Manager. Sämtliche anderen XULU-Komponenten sind für ein Modell transparent[12]. Die hierzu in Abschnitt 4.4.4 beschriebene Modell-Schnittstelle wurde im Rahmen der Realisierung in in 2 Teile aufgeteilt, um die technischen Aufgaben der Ressourcen-Verwaltung vom semantischen Bereich des Modell-Ablaufs zu trennen:

a) **`ModelContentManager`**
Implementiert die Spezifikation und Verwaltung der Modell-Ressourcen (Vorlauf-Phase in Abb. 4.7 auf Seite 48, sowie die Ressourcen-Überprüfung).

[11] bzw. in der entsprechenden Property der `GridList` und des `MultiGrid`

[12] **Bemerkung:**
Wenn ein Modell Ereignisse initiiert, kommuniziert es auch mit dem Event-Manager. Diese Kommunikation erfolgt jedoch *passiv*, da sich der Modell-Manager als *Listener* an das Modell ankoppelt. Das Modell initiiert Events an alle Listener, ohne zu berücksichtigen, dass es sich dabei um den XULU-Modell-Manager handelt.

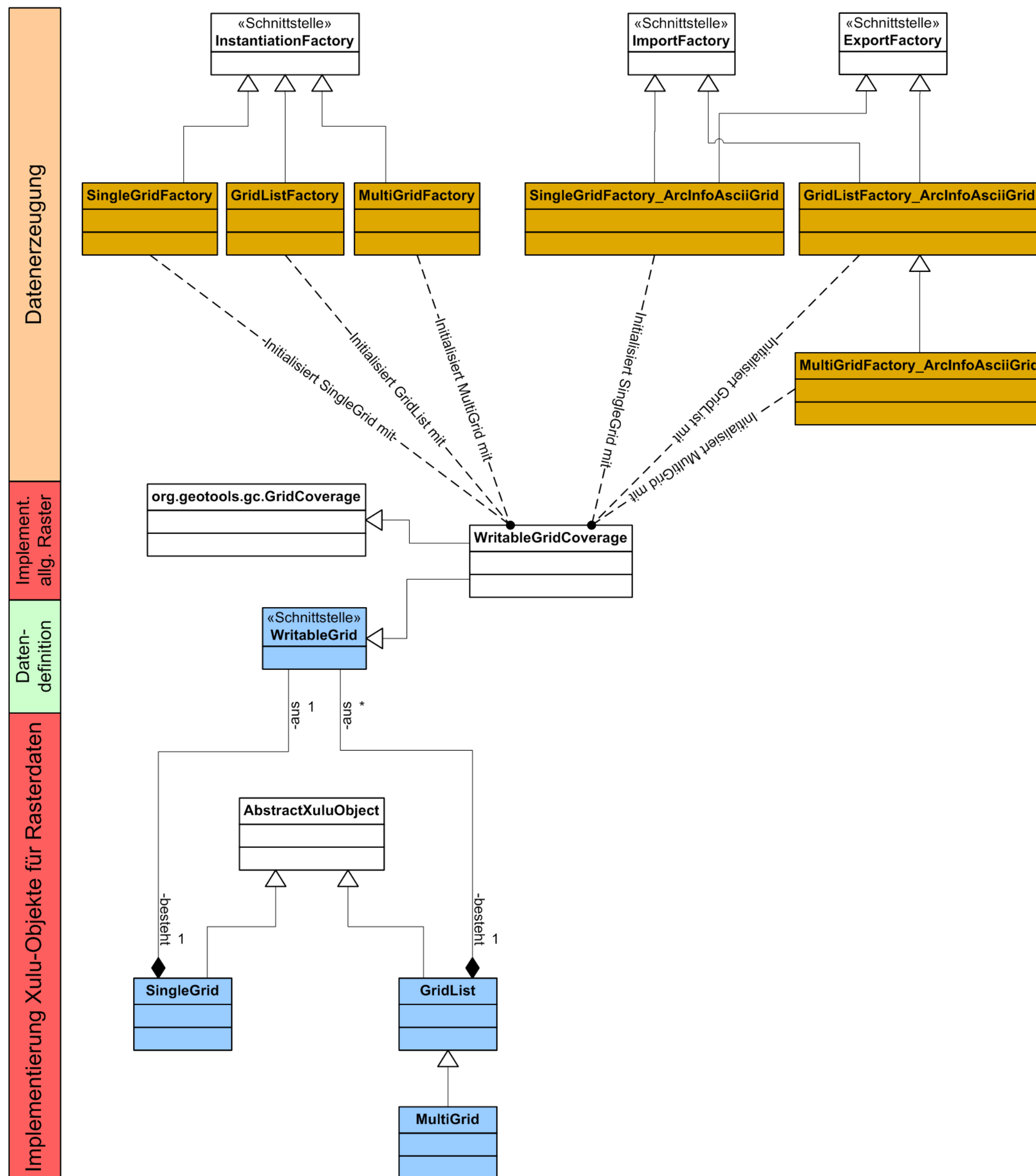


Abbildung 5.12: Der Aufbau der Raster-Datentypen und -Factorys

b) **XuluModel**
Implementiert die Phasen der Initialisierung[13], des Modell-Algorithmus, sowie der Finalisierung.

Die Grundlage für die Einbettung eines Modells in die Plattform bilden die Implementierungen der Klasse `XuluModel`, welche als Plugin in XULU integriert werden (vgl. 5.3). Diese müssen dem Modell-Manager jedoch eine Instanz von `ModelContentManager` bereitstellen. Für beide Teil-Schnittstellen stellt die XULU-Plattform Basis-Implementierungen bereit, welche eine Reihe der unter 4.4.4 beschriebenen Anforderungen bereits implementieren.

Die (abstrakte) Basis-Implementierung des Modells realisiert zum Beispiel bereits die Verwaltung der geplanten Modell-Events in einer Liste. Ereignisse, wie „Modell initialisiert“, „Modell gestartet“ werden bereits definiert und an entsprechenden Stellen initiiert. Verwendet eine Modell-Implementierung weitere Event-Arten, muss es diese lediglich noch dieser Liste hinzufügen. Eine weitere abstrakte Klasse wurde für Modelle implementiert, die in festen Zeitschritten ablaufen und in jedem dieser Zeitschritte denselben Ablauf ausführen. Auf dieser Klasse basiert beispielsweise die in Kapitel 6 beschriebene CLUE-Implementierung. Statt des kompletten Algorithmus muss lediglich noch der Ablauf *eines einzelnen* Zeitschritts realisiert werden. Die Abfolge der Zeitschritte und das Initiieren entsprechender Events („Zeitschritt begonnen“ und „Zeitschritt beendet“) ist bei der Programmierung eines konkreten Modells nicht mehr zu berücksichtigen (Vorbild: *Template Method Pattern* [9, 4]).

5.6.1 Modell-Ressourcen

In Abschnitt 4.4.1 wurde die Struktur einer Modell-Ressource dargestellt. Sie spezifiziert ein für die Modellierung benötigtes Objekt als Tupel

$$\langle \text{Beschreibung, NullableFlag, Datentyp, Datenobjekt} \rangle$$

Dieses Tupel wird in XULU als ein *eigenständiges* Objekt implementiert, das sog. *Ressourcen-Objekt*. Dieses *kapselt* ein Datenobjekt zwischen Datenpool und Modell (siehe Abb. 5.13). Der Content-Manager erzeugt für jedes benötigte Modell-Objekt ein Ressourcen-Objekt und legt dabei dessen Beschreibung, Datentyp (JAVA-Klasse) und Nullable-Flag fest. Die Stelle für das Datenobjekt bleibt zunächst leer. In der Vorlauf-Phase (vgl. 4.4.3) übergibt der Content-Manager sämtliche Ressourcen-Objekte an den Modell-Manager. Dieser generiert daraufhin ein entsprechendes User Interface, durch das der Anwender jeder Ressource ein Objekt aus dem Datenpool zuordnen kann (siehe 5.6.2). Diese Datenobjekte hinterlegt der Modell-Manager anschließend im Ressourcen-Objekt und veranlasst den Content-Manager eine Überprüfung durchzuführen, ob die Zuordnung für das Modell konsistent ist. Ist dies der Fall, kann das Modell im Laufe

[13] Mit Ausnahme der Ressourcen-Überprüfung.

der Initialisierungsphase die Datenobjekte wieder aus den Ressourcen-Objekten „entnehmen“. Während des Modell-Ablaufs wird nur noch auf den Datenobjekten gearbeitet. Die Ressourcen-Objekte („Kapseln“) werden nicht mehr verwendet.

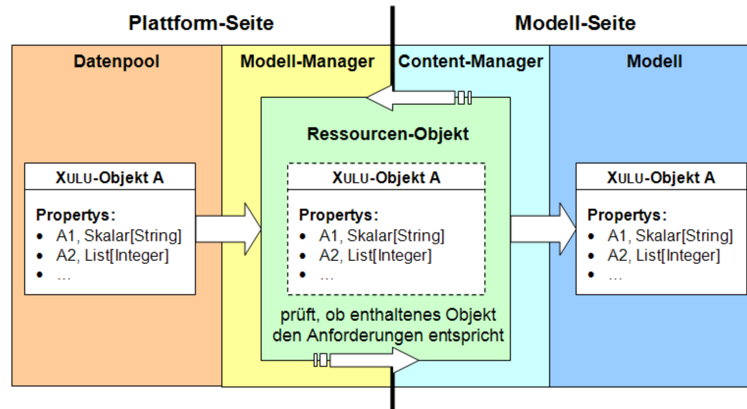


Abbildung 5.13: Eine Ressourcen-Objekt *kapselt* ein Daten-Objekt zwischen Datenpool und Modell.

Ein Punkt, der in der oben geschilderten Vorgehensweise leicht übersehen werden kann, für einen reibungslosen Modellablauf jedoch entscheidende Bedeutung hat, ist das Problem der Daten-Konsistenz. Wenn das Modell im Laufe der Initialisierungsphase ein Datenobjekt aus einem Ressourcen-Objekt entnimmt, muss es davon ausgehen können, dass dieses Objekt den Modell-Anforderungen genügt. Hierbei müssen *lokale* und *globale* Konsistenz gewährleistet sein (siehe Tabelle 5.4).

Während die globalen Bedingungen meist gut aus dem Modell-Kontext heraus erkennbar sind, können die lokalen vom Modell-Entwickler leicht übersehen werden. Doch insbesondere, wenn ein Modell von unerfahrenen Anwendern genutzt wird, sind auch diese wichtig. Allein durch die Spezifizierung des für eine Ressource benötigten Datentyps, kann die lokale Konsistenz in der Regel nicht gewährleistet werden. Dies ist zum Beispiel für das Matrix-Beispiel in Tabelle 5.4 der Fall. Der geforderte Datentyp ist lediglich `MatrixProperty`. Eine bestimmte Anforderung an deren Format wird damit nicht verbunden. Die Konsistenz-Prüfung muss gesondert erfolgen. Dabei sollte der Content-Manager möglichst jeden erdenklichen Fall abdecken, der durch die Zuteilung des Anwenders entstehen kann.

lokale Konsistenz	**globale Konsistenz**
• betrifft eine einzelne Ressource • Check kann lokal an ein Ressourcen-Objekt gebunden werden • Beispiel: Hat eine angeforderte Matrix-Property die richtige Dimension und Größe?	• betrifft die Ressourcen untereinander • Check muss global über den Content-Manager erfolgen • Beispiel: Haben alle Raster-Ressourcen dasselbe Format?

Tabelle 5.4: Der Unterschied zwischen *lokaler* und *globaler* Ressourcen-Konsistenz.

Die Bedeutung einer guten Konsistenz-Überprüfung wird ersichtlich, wenn man sich vor Augen hält, dass komplexe Modelle mitunter sehr lange laufen können. Umso schwerer wiegt ein (später) Modellabbruch, der durch einen einfachen Konsistenz-Check bereits vor dem Modell-Start hätte erkannt werden können. Auf die Definition der Ressourcen und die zugehörige Überprüfung sollte bei der Implementierung des Content-Managers sehr viel Wert gelegt werden, auch wenn hierfür u.U. sehr viel Source-Code notwendig ist.

5.6.2 Genereller Modell-Ablauf

Auf Basis der Ressourcen-Objekte generiert der Modell-Manager ein modellspezifisches User Interface (Abbildung 5.14). Der obere und untere Bereich (Buttons und Status-Feld) ist für alle Modelle gleich. Unterschiedlich ist lediglich der mittlere Teil des Fensters, in dem alle vom Modell benötigten Ressourcen aufgelistet sind. Hierüber kann der Anwender den Ressourcen Objekte aus dem Datenpool zuordnen (sowohl XULU-Objekte, als auch einzelne Propertys). Über den Objekt-Typ, der durch das jeweilige Ressourcen-Objekt gefordert wird, ermittelt der Modell-Manager eine Vorauswahl „passender“ Datenpool-Objekte, aus denen der Anwender nur noch das gewünschte auswählen muss.

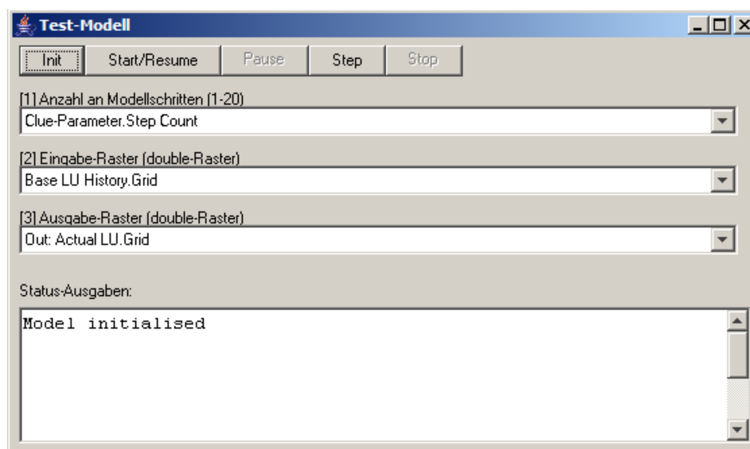


Abbildung 5.14: Das Steuerungs-Fenster wird für jedes Modell auf Basis der geforderten Ressourcen erstellt.

Die Schwierigkeit der Implementierung lag hierbei in der dynamischen Gestaltung der GUI, die von Modell zu Modell variiert. Hierfür konnte jedoch auf im JAVA-Standard bestehende *Layout-Manager* zurückgegriffen werden. Diese übernehmen die automatische Positionierung, Anordnung und Größenanpassung der Ressourcen-Auswahlfelder, entsprechend dem Platz, der im mittleren Fensterbereich zur Verfügung steht.

Bemerkung:
Die dargestellte Ressourcen-Zuordnung über Auswahl-Listen impliziert, dass ein Datenpool-Objekt nur dann einer Ressource zugeordnet werden kann, wenn es exakt dem geforderten Datentyp entspricht. Da XULU-Objekte sich jedoch aus Propertys zusammensetzen (vgl. 5.5.1), wäre es auch denkbar, dass der Anwender die durch die Ressource geforderte Objekt-Struktur dynamisch (durch eine Art Query) aus verschiedenen Datenpool-Objekten zusammensetzt. Eine entsprechende Erweiterung der XULU-Plattform würde die Modell-Schnittstelle nicht beeinflussen, sondern könnte komplett plattform-intern realisiert werden.

Mit Beendigung der Ressourcen-Zuteilung durch den Anwender (Init-Button), erfolgt die Überprüfung der Ressourcen durch den Content-Manager des Modells. Ist dies erfolgreich, erzeugt der Modell-Manager einen Prozess, der die Ausführung des Modell-Algorithmus kapselt. Dies ist wichtig, damit die XULU-Plattform und insbesondere die Steuerungs-Komponente während der Modell-Ausführung nicht blockiert werden. Die Implementierung des Modell-Prozesses basiert auf den Standard-Mechanismen des JAVA-Threading. Die Kapselung des Modells in diesem Thread erfolgt vollständig transparent für die Modell-Implementierung und braucht bei deren Entwicklung nicht berücksichtigt werden.

Abbildung 5.15 zeigt den generellen Modellierungsablauf, der durch den XULU-Modell-Manager vorgegeben ist. Der linke Teil (gelb) erfolgt vollständig transparent für die Modell-Implementierungen. Diese müssen lediglich noch die Teil-Abläufe im mittleren und rechten Bereich (blau, grün) realisieren.

5.7 Zusammenfassung

Die Implementierung der XULU-Modelling-Platform setzt die wesentlichen in Kapitel 4 entworfenen Konzepte um. Auf die Realisierung der dargestellten Mechanismen zur Einbettung von Modell-Synchronisation und interaktiver Datenmanipulation wurde jedoch verzichtet, um den Rahmen dieser Diplomarbeit nicht zu sprengen. Gleiches gilt für die in Kapitel 4 motivierte Datenbank-Anbindung. Diese sollen Gegenstand weiterführender Projekte sein. Aus diesem Grund wurde sehr viel Wert darauf gelegt, durch einen stark komponentenbasierten Aufbau (mit *festen* internen Schnittstellen) zu erreichen, dass die XULU-Plattform auch außerhalb des Plugin-Konzepts flexibel erweiterbar ist.

Während der Implementierung kristallisierten einige Aspekte heraus, die zu einer teilweisen Erweiterung des Plattform- und Schnittstellen-Entwurfs führten:
Probleme im Einsatz der JAVA-*Reflection-API* zur Realisierung der Plugin-Verwaltung (5.3) machten es erforderlich, dass neben der Funktionalität der Plattform auch vermehrt Aspekte der Anwenderfreundlichkeit berücksichtigt werden mussten. Über die XULU-*Script-Interpreter* können verschiedene Abläufe (z.B. Datenimport) innerhalb der XULU-Plattform automatisiert werden (5.1). Die Handhabbarkeit der Plattform wird

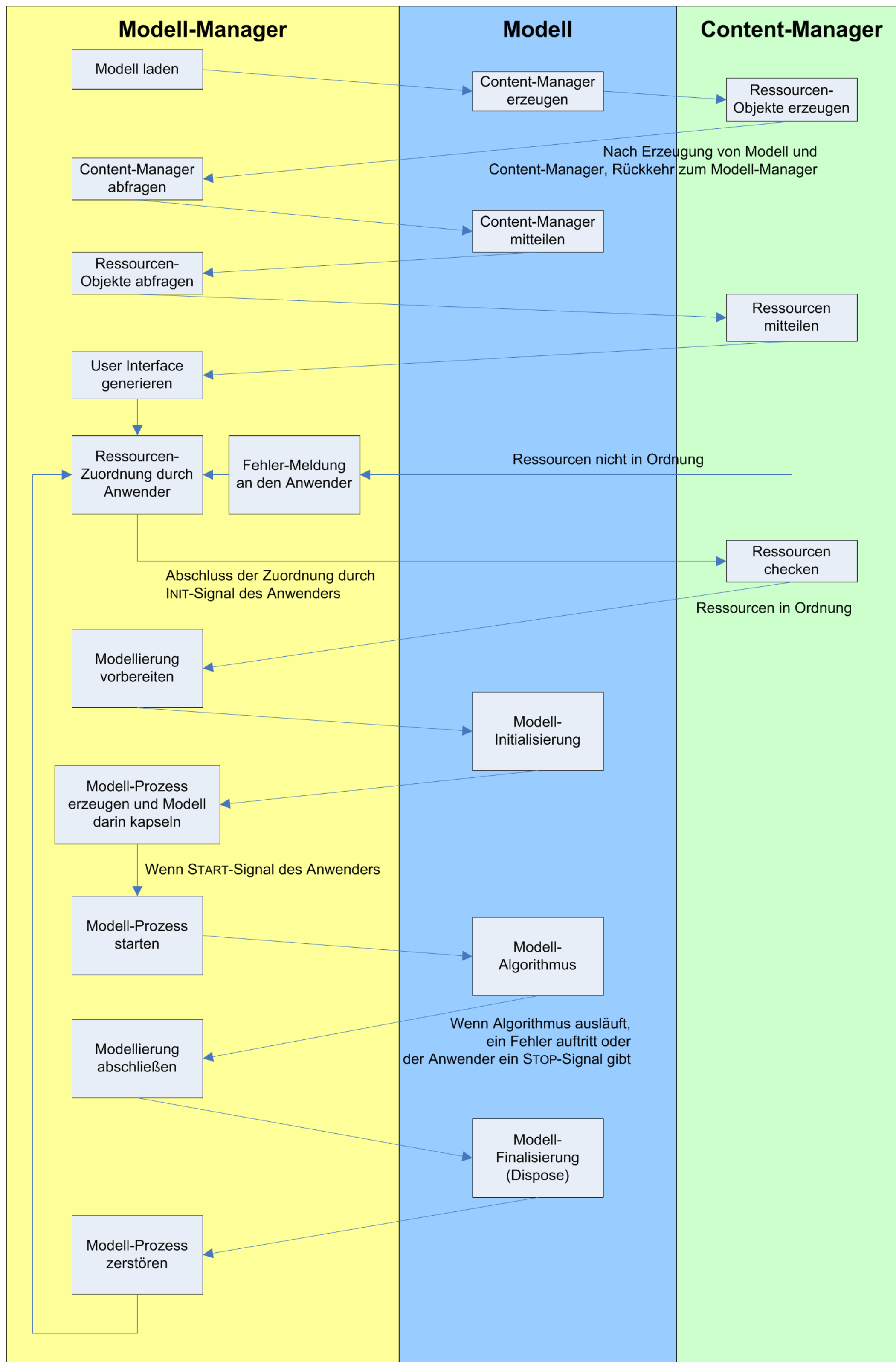


Abbildung 5.15: Der Steuerung des Modellierungsablaufs durch den Modell-Manager.

hierdurch insbesondere für die Phase der Modell-Entwicklung wesentlich verbessert.
Die in Kapitel 4 entworfene Modell-Schnittstellen teilt sich im Zuge der Implementierung in zwei Bereiche (*Modell* und *Content-Manager*), um eine Trennung des semantischen Modell-Ablaufs von der technischen Ressourcen-Verwaltung zu erreichen. Abschnitt 5.6.1 hebt in diesem Zusammenhang die Bedeutung guter Checks für die Ressourcen-Konsistenz hervor.
Die Datentyp-Schnittstelle wurde um das *Property-Konzept* erweitert (5.5.1). Hierdurch war es möglich, Zugriffskontrolle und Änderungspropagierung zentral zu implementieren. Zudem konnte ein *dynamisches* XULU-*Objekt* erstellt werden, das über eine einfache ASCII-Datei generiert (importiert) wird. Eine Erweiterung sieht vor, eine „Baukasten"-Komponente zur interaktiven Datentyp-Erstellung in die XULU-Applikation zu integrieren (5.5.1).

Der allgemeine Plattform-Rahmen ist vollständig unabhängig von einem konkreten Anwendungsgebiet gestaltet. Für die Integration konkreter Plugins (insbesondere Datentypen und Modelle) wurden allgemeine Basis-Implementierungen der entsprechenden Schnittstellen entwickelt, so dass im Rahmen der Weiterentwicklung nicht sämtliche Methoden neu (und redundant) zu implementieren sind.
Diese wurden dazu genutzt, um im Hinblick auf die Integration eines konkreten Landnutzungsmodells (Kapitel 6) exemplarische Plugins zu implementieren, die auf das Geo-Anwendungsgebiet ausgerichtet sind. Hierzu gehören Raster- und Vektor-Datentypen mit entsprechenden Import- und Export-Factorys. Am Beispiel von Raster-Daten beschreibt Abschnitt 5.5.3 den Entwurf und Aufbau eines XULU-Datentyps, sowie das Zusammenspiel mit den Factorys. Darüberhinaus wurde eine layer-basierte Geo-Visualisierung realisiert, die im vorangegangenen Kapitel jedoch nicht näher erläutert wurde.

Kapitel 6

Evaluation von XULU anhand des CLUE-Modells

Nach der Implementierung der Modellierungsplattform XULU stellt sich die Frage, ob und wie deren Entwurf im „Echtbetrieb" die Anforderung des Modell-Entwicklers und Modell-Anwenders (Kapitel 3) erfüllt.

Um dies zu überprüfen, wurde kein komplett neues LUC-Modell entwickelt, sondern eine Nachbildung des CLUE-Modells angestrebt (vgl. Kapitel 2.3). Da es sich dabei um ein bestehendes und bereits mehrfach erfolgreich eingesetztes Landnutzungsmodell handelt, musste der aufwändigste Teil einer Modell-Entwicklung – der semantische Modell-Entwurf – nicht betrachtet werden. Dieser sollte im Rahmen dieser Diplomarbeit ohnehin eine untergeordnete Rolle spielen. Statt dessen konnte das Hauptaugenmerk des Evaluationsprozesses darauf gelegt werden, die (bestehende) CLUE-Algorithmik in die XULU-Modelling-Platform zu integrieren.

Für den Evaluationsprozess wurde mir von der RSRG ein kompletter CLUE-Datensatz für das IMPETUS-Untersuchungsgebiet (Benin) zur Verfügung gestellt (siehe Anhang E). Da dieser Datensatz durch MICHAEL JUDEX bereits erfolgreich in CLUE eingesetzt wurde, ist er gut geeignet, um die Arbeitsweise der XULU-Implementierung von CLUE mit dem Original-CLUE-S zu vergleichen[1]. Die Raster-Dateien können direkt in XULU verwendet werden. Import/Export-Factorys für das ARCINFOASCIIGRID-Format sind bereits Bestandteil der XULU-Plattform. Alle Daten, die für CLUE-S in einzelnen Dateien hinterlegt sind (Steuerungs-Parameter, LU-Conversion-Matrix, LU-Elasticity, LU-Bedarf, usw.), werden hingegen für die Verwendung in XULU in *einer* Datei zusammengefasst und stellen somit auf Seite des XULU-Datenpools ein einziges XULU-Objekt dar (vgl. 5.5.1). Dieses enthält die oben genannten Daten als je eine Objekt-Eigenschaft (Property). Hierdurch wird für den Anwender eine einfache Handhabung der Daten erreicht (übersichtlicher Datenpool, nur ein einziger Import-Vorgang).

[1] Im folgenden ist mit „CLUE-S" die Original-Applikation des CLUE-Modells (samt GUI, Konfigurationsdateiem, usw.) gemeint. Hingegen bezeichnet „XULU-CLUE" die im Rahmen dieser Diplomarbeit erstellte Nachbildung.

6.1 Implementierung des CLUE-Modells

Da es sich bei CLUE-S um eine in C/C++ programmierte, modell-spezifische Applikation handelt, XULU dagegen eine in JAVA implementierte Plattform darstellt, die spezielle Schnittstellen voraussetzt, war es nicht möglich, Teile des Original-CLUE unmittelbar in XULU zu übernehmen. Statt dessen wurde die CLUE-Arbeitsweise für XULU komplett neu implementiert. Für die XULU-Schnittstelle waren 3 Klassen zu erstellen:

- **`ClueModel`:** Implementiert die Modell-Initialisierung und den CLUE-Algorithmus.
- **`ClueModelContentManager`:** Definiert und prüft die für CLUE benötigten Daten (Ressourcen) und verwaltet sie für den Algorithmus.
- **`ClueModel.ClueModelGUI`:** Implementiert ein CLUE-spezifisches UI, das zusätzliche Statusausgaben und Steuerungsfunktionen enthält.

An dieser Stelle sollte darauf hingewiesen werden, dass in XULU nicht alle Features der Original-Software CLUE-S integriert wurden. Statt dessen beschränkt sich das XULU-CLUE auf die von der RSRG zur Simulation des IMPETUS-Untersuchungsgebiets genutzten Funktionen. Beispielsweise wurde auf sog. *Multi-Region*-Szenarien verzichtet. In der XULU-Implementierung besteht das Untersuchungsgebiet immer aus einer einzigen Region. Desweiteren wurde Wert darauf gelegt, den Algorithmus übersichtlich zu implementieren, um die Arbeitsweise von CLUE gut nachvollziehen zu können. Eine besonders effiziente Umsetzung stand nicht im Vordergrund (vgl. 6.3.3).

6.1.1 Integration in die Ressourcen-Philosophie

Die erste Anforderung, die XULU an eine Modell-Implementierung stellt, ist die Definition der Ressourcen. Für das CLUE-Modell spezifiziert die Klasse `ClueModelContentManager` insgesamt 24 Ressourcen. Das daraus vom XULU-Modell-Manager dynamisch generierte Steuerungs-UI ist in Abbildung 6.1 dargestellt.

Wahl der Ressourcen

Auf die semantische Bedeutung der einzelnen Eingabe-Ressourcen (R1-R18, in Abb. 6.1 grün dargestellt) möchte ich an dieser Stelle nicht näher eingehen. Die wichtigsten werden in Kapitel 2.3.1 beschrieben. Weitere Details können der CLUE-Dokumentation entnommen werden [38]. Neben den Eingabe-Ressourcen, fordert das XULU-CLUE noch vier Daten-Objekte zur Speicherung temporärer Zwischenergebnisse (R19-R22, orange), sowie zwei für die eigentliche Modellausgabe (R23-R24, blau).

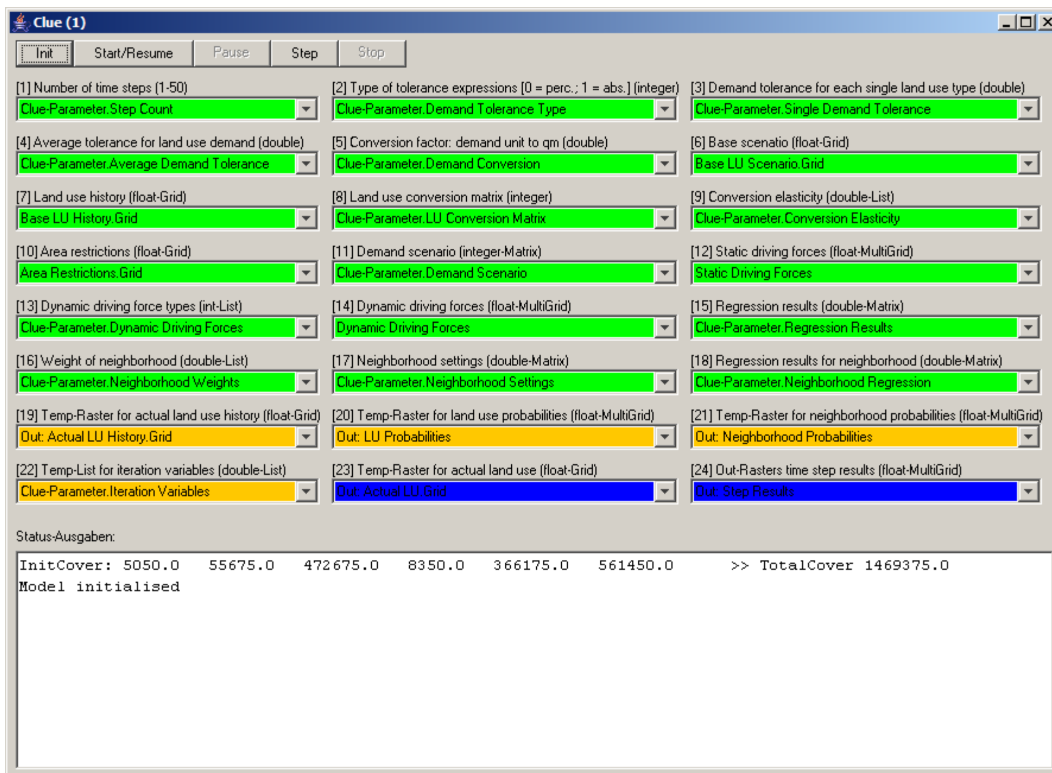


Abbildung 6.1: Das von XULU dynamisch generierte Kontroll-Fenster für das CLUE-Modell.

Da die Wahl dieser Ressourcen nicht nur durch die CLUE-Semantik begründet ist, sondern auch durch die technischen Gegebenheiten der XULU-Plattform, werden diese im Folgenden etwas näher beschrieben:

R19, R23 Zur Zwischen-Speicherung der aktuellen LUC-Konfiguration und -Historie könnten auch die Ressourcen verwendet werden, in denen die jeweiligen Ausgangsdaten hinterlegt sind (R6: *Base Scenario* und R7: *Land Use History*). Um diese jedoch für einen weiteren Modell-Lauf verwenden zu können[2], ist es jedoch sinnvoll, sie nicht zu überschreiben, sondern je ein neues Raster zu verwenden. Da die Verwaltung „grosser" Datentypen in der Verantwortung der XULU-Plattform liegt (vgl. 4.6 und 4.8.1), werden diese Raster als Ressource angefordert.

R20, R21 Zu Beginn eines Zeitschritts (vor der Iteration) werden LUC-Wahrscheinlichkeiten berechnet. Um diese zwischenzuspeichern, werden zwei Raster-Listen benötigt, die als Ressource vom Datenpool zur Verfügung gestellt werden müssen (vgl. 19, 23).

R22 Für den Modellablauf sind die Iterationsvariablen $ITER_U$ ein entscheidender Faktor. Damit der Modell-Entwickler die Möglichkeit hat, deren Verlauf zu analysieren, müssen sie sich im Daten-

[2] Ohne sie neu importieren zu müssen.

pool befinden (also in Form einer Ressource vorliegen).

R24 Da in XULU noch kein Event-Handler zum automatischen Festschreiben (Export) von Daten-Objekten implementiert ist, wird eine Raster-Liste benötigt, in der die simulierte Landnutzung eines jedes Zeitschritts hinterlegt wird[3].

Die für die Ressource R22 erläuterte „Analyse-Möglichkeit" ist ein zusätzlicher Vorteil, der sich – durch die Verwaltung *als Ressource* – auch für die temporären Modell-Daten (R19-R21, R23) ergibt.

Bemerkung:
Im Gegensatz zu CLUE-S wird in XULU auf die explizite Angabe einiger Modell-Parameter verzichtet, da diese implizit durch die übrigen Ressourcen gegeben sind:

- Die **Anzahl an LUC-Typen** ist implizit durch die Größe der LU-Conversion-Matrix (R8) gegeben.
- Die **Gesamt-Anzahl an Driving Forces** ist implizit durch die Listengröße der statischen Driving Forces (R12) gegeben.
- **Die Ausmaße des Untersuchungsgebiets** werden durch das Start-Szenario (R6) festgelegt.

Abschließend sollte noch ein Sachverhalt im Zusammenhang mit der Wahl der Eingabe-Ressourcen erläutert werden:
Wie zu Beginn des Kapitels erwähnt wurde, werden auf der Datenpool-Seite alle Nicht-Rasterdaten in einem Objekt zusammengefasst. Von Modell-Seite her wird jedoch bewusst darauf verzichtet, exakt *dieses* Objekt (mit fest vorgegebenen Eigenschaften) als Ressource anzufordern. Statt dessen werden alle Parameter als einzelne Ressourcen definiert. Zwar erhöht sich hierdurch die Anzahl der für den Anwender zuzuordnenden Ressourcen, gleichzeitig jedoch auch die Flexibilität. Um z.B. zwischen alternativen *LU-Conversion-Matrizen* oder *Regression Results* hin und her zu wechseln, kann der Anwender die alternativen Konfigurationen als zusätzliche Objekte in den Datenpool laden und je nach Bedarf einer Ressource zuordnen. Verwendet man als Ressource hingegen einen konkreten Objekttyp mit *festen* Propertys, müsste das Ressourcen-Objekt für jede Änderung manuell angepasst werden. Aufgrund der in XULU noch sehr begrenzten Möglichkeiten der Datenmanipulation, wäre sogar jeweils ein Abändern und Neu-Importieren der zugrunde liegenden Datei erforderlich.

[3] Alternativ müsste der Anwender das Modell schrittweise ausführen und den Inhalt von Ressource 23 nach jedem Zeitschritt manuell exportieren.

Implementierung der Ressourcen

Da XULU bereits eine modell-unabhängige Basis-Implementierung für die Schnittstelle des Content-Managers zur Verfügung stellt[4], sind die Verwaltung der Ressourcen, sowie entsprechende Zugriffsmethoden für den CLUE-Content-Manager nicht mehr zu implementieren. Die einzelnen Ressourcen müssen lediglich noch definiert und auf Konsistenz untereinander geprüft werden.

Um den Content-Manager übersichtlich und strukturiert zu gestalten, wird für *jede* der 24 Ressourcen eine eigene Klasse definiert[5]. Diese sind in Tabelle 6.1 aufgeführt. Wie daraus ersichtlich ist, wird der überwiegende Teil der Ressourcen als *Property* (Objekt-Eigenschaft) angefordert. Der Grund hierfür liegt im Access- und Event-Management der XULU-Plattform. Dieses wird über die `Property`-Klassen implementiert und stünde für den Modell-Ablauf nicht zur Verfügung, wenn z.B. ein `Integer`- oder `WritableGrid`-Typ direkt als Ressourcen-Typ definiert würde.

Die einzigen Ressourcen, die nicht als Property definiert sind, sind die Raster-Listen (R12, R14, R20, R21, R24). Hierfür wird der konkrete XULU-Objekttyp `MultiGrid` vorgeschrieben. Alternativ wäre auch eine `ListProperty[WritableGrid]` möglich. Diese garantiert jedoch nicht, dass alle in der Liste enthaltenen Raster das gleiche Format haben (Größe, Geo-Position, usw.). Durch das `MultiGrid` wird diese Forderung automatisch gewährleistet. Darüberhinaus bietet es die Möglichkeit, die Raster-Liste automatisch um ein neues „passendes“ Raster zu erweitern. Je nach Anzahl der modellierten LUC-Typen (durch R8 definiert), können die Ausgabe-Ressourcen R20 und R21 *durch das Modell* (während der Initialisierung) auf die passende Größe erweitert werden. Dies muss nicht vorab durch den Anwender durchgeführt werden. Auf dieselbe Weise wird R24 sukzessive (während des Modellablaufs) vergrößert. Es ist nicht erforderlich, dass zu Beginn der Simulation bereits Ausgaberaster für *alle* Zeitschritte in der Liste vorhanden sind.

Im Rahmen Ressourcen-Implementierung wurde die Überprüfung der *lokalen* und *globalen* Konsistenz (vgl. 5.6.1) getrennt und der lokale Check direkt an das Ressourcen-Objekt gebunden. Hierzu implementiert jede der 24 Ressourcen-Klassen – neben der Definition des Datentyps und einer Beschreibung – eine Methode, mit der die *lokale* Korrektheit des (durch den Anwender) zugeordneten Objekts geprüft wird (z.B. $R2 \in \{0,1\}$, $R3 \geq 0$, $R2 \geq 0$).
Die Konsistenz-Prüfung für die Ressourcen untereinander – z.B. ob die Größe der Bedarf-Matrix (R11) mit der Anzahl zu simulierender Zeitschritte (R1) verträglich ist – erfolgt in einer *globalen* Routine des Content-Managers, nachdem die *lokale* Konsistenz für alle Ressourcen sicher gestellt ist.

[4] Die abstrakte Klasse `AbstractModelContentManager`.

[5] Als Inline-Klasse von `ClueModelContentManager`. Alternativ wäre auch eine Klassendefinition zum Instanziierungszeitpunkt (bei Zuweisung zur Instanz-Variablen) möglich. Dies spart zwar Sourcecode, ist jedoch – bei 24 Ressourcen – sehr unübersichtlich.

	Ressource-Klasse	**Datentyp**
R1	`ModelStepResource`	`ScalarProperty[Number]`
R2	`ToleranceTypeResource`	`ScalarProperty[Integer]`
R3	`SingleToleranceResource`	`ScalarProperty[Number]`
R4	`AverageToleranceResource`	`ScalarProperty[Number]`
R5	`ConversionFactorQMResource`	`ScalarProperty[Double]`
R6	`InitialScenarioResource`	`ScalarProperty[WritableGrid]`
R7	`LUHistoryResource`	`ScalarProperty[WritableGrid]`
R8	`LUConversionMatrixResource`	`MatrixProperty[Integer]`
R9	`ConversionElasticityResource`	`ListProperty[Double]`
R10	`AreaRestrictionsResource`	`ScalarProperty[WritableGrid]`
R11	`DemandScenarioResource`	`MatrixProperty[Integer]`
R12	`StaticDrivingForcesResource`	`MultiGrid`
R13	`DynamicDrivingForceTypesResource`	`ListProperty[Integer]`
R14	`DynamicDrivingForcesResource`	`MultiGrid`
R15	`RegressionResultsResource`	`MatrixProperty[Double]`
R16	`NeighborhoodWeightsResource`	`ListProperty[Double]`
R17	`NeighborhoodSettingsResource`	`MatrixProperty[Double]`
R18	`NeighborhoodRegressionResultsResource`	`MatrixProperty[Double]`
R19	`ActualLUHistoryResource`	`ScalarProperty[WritableGrid]`
R20	`LUProbabilitiesResource`	`MultiGrid`
R21	`NeighborhoodProbabilitiesResource`	`MultiGrid`
R22	`IterationVariablesResource`	`ListProperty[Double]`
R23	`ActualLUResource`	`ScalarProperty[WritableGrid]`
R24	`StepResultsResource`	`MultiGrid`

Tabelle 6.1: Ressourcen-Definitionen durch den Content-Manager des CLUE-Modells.

6.1.2 Modell-Ablauf

Wie für den Content-Manager enthält die XULU-Plattform bereits eine Basis-Implementierung für Modelle, die in festen Zeitschritten ablaufen[6]. Die Abfolge der Zeitschritte (und die Erzeugung entsprechender Events) muss somit von `ClueModel` nicht mehr realisiert werden, sondern lediglich je eine Methode zur Modell-Initialisierung, zur „Modell-Zerstörung“ (Dispose) und zur Implementierung eines Modell-Schritts (Algorithmus).

Initialisierungsphase

Während des Modell-Ablaufs werden vom XULU-CLUE zwei Ereignisse initiiert[7]. Zu Beginn der Initialisierungsphase werden die entsprechenden Ereignistypen in eine (von der Oberklasse vorgefertigte) Liste aufgenommen, damit sie dem Anwender für die Definition von benutzerdefinierten Event-Handlern zur Verfügung stehen (vgl. 5.4).

Anschließend werden der Übersicht halber die für die Modellierung benötigten Daten (bzw. die Referenzen darauf) aus den Ressourcen-Objekten in lokale Variablen kopiert[8]. Hierbei werden gleichzeitig die jeweiligen Lese- und Schreib-Rechte auf die Daten-Objekte reserviert, die somit für den kompletten Modellablauf gehalten werden. Desweiteren wird ein *interner* Array initialisiert, in dem innerhalb eines Iterationsschritts *on-the-fly* die aktuelle Gesamt-Flächendeckung pro LUC-Typ hochgezählt wird. Für das Abbruch-Kriterium der Iteration (Vergleich zwischen LUC-Bedeckung und Bedarf) muss das Raster somit nicht nochmals abgearbeitet werden. Darüberhinaus stehen hierdurch permanent auch die Bedeckungswerte der vorangegangenen Zeitschritte zur Verfügung. Diese spielen u.a. bei der Berechnung der Nachbarschafts-Wahrscheinlichkeiten eine Rolle.
Zum Abschluss der Initialisierungsphase wird die CLUE-spezifische GUI erzeugt (vgl. 6.1.3). Dies kann zuvor noch nicht geschehen, da erst nach erfolgter Ressourcen-Zuteilung feststeht, wie viele LUC-Typen simuliert werden. Entsprechend viele Status-Balken müssen in die GUI integriert werden.

Dispose-Phase

In der Dispose-Phase werden die seit der Initialisierungsphase gehaltenen Zugriffsrechte auf die Datenobjekte (bzw. Objekt-Propertys) wieder freigegeben. Da die Dispose-Phase von der Plattform nicht nur nach dem Modellablauf initiiert wird, sondern auch im Falle einer Fehlersituation, ist gewährleistet, dass die Ressourcen nicht unkontrolliert gesperrt bleiben.

[6] Die abstrakte Klasse `AbstractStepModel`.

[7] `ModelIterationStepStartedEvent` und `ModelIterationStepFinishedEvent`

[8] Zudem ist der Daten-Zugriff über den Content-Manager und die Ressourcen-Objekte mit zusätzlichen Funktionsaufrufen verbunden. Insbesondere bei häufig verwendeten Objekten (wie Wahrscheinlichkeits- und Ausgaberaster), wirkt sich dies unnötig negativ auf die Effizienz aus (vgl. 6.3.3).

Ein Modell-Schritt

Der Ablauf eines Modell-Schritts erfolgt weitestgehend so, wie in Kapitel 2.3.2 beschrieben, mit drei Ausnahmen:

a) Sofern mit dynamischen Driving Forces gearbeitet wird, müssen die LUC-Wahrscheinlichkeiten in jedem Zeitschritt neu berechnet werden.

b) Neben den LUC-Wahrscheinlichkeiten gehen im XULU-CLUE auch Nachbarschaftsbeziehungen in die Gesamtwahrscheinlichkeit $TPROB_{i,U}$ ein. Da sich die LUC-Konfiguration permanent ändert, müssen diese Nachbarschafts-Wahrscheinlichkeiten in jedem Zeitschritt neu berechnet werden.

c) Die Raster-Zellen, die in einem Zeitschritt *nicht* für einen LUC-Wechsel in Frage kommen, werden *on-the-fly* identifiziert und nicht – wie in 2.3.2 beschrieben – in einem isolierten Schritt.

Auf sämtliche Details der Programmierung einzugehen, würde den Rahmen dieses Abschnitts sprengen. Deshalb beschränke ich mich an dieser Stelle auf die grobe Vorgehensweise, welche in Abbildung 6.2 als Pseudocode dargestellt ist.

Die **Berechnung der LUC-Wahrscheinlichkeiten** (Kapitel 2.3.1, Formel 2.5) erfolgt naiv über 4 geschachtelte FOR-Schleifen. Diese laufen über alle Rasterzellen (x,y), alle LUC-Typen u, sowie alle Driving Forces m und summieren pro LUC-Typ den Faktor aus Regressions-Beta (R15) und Ausprägung des Driving Force (R12/R14) auf. Die Wahrscheinlichkeit ergibt sich aus dem in Formel 2.5 dargestellten Quotienten, in dem in Zähler und Nenner die ermittelte Summe eingesetzt wird. Die so errechnete Wahrscheinlichkeit wird in Ressource R20 hinterlegt (der Wert des vorangegangenen Zeitschritts wird überschrieben).

Die **Berechnung der Nachbarschafts-Wahrscheinlichkeiten** soll an dieser Stelle nur angedeutet werden. Sie basiert im Kern auf der Analyse, wie viele Zellen eines jeden LUC-Typs sich in der Umgebung (R17) einer Zelle befinden. Da die Umgebung pro LUC-Typ differenziert betrachtet (gewichtet) wird, ergeben sich im Wesentlichen 5 geschachtelte FOR-Schleifen: Für jede Raster-Zelle (x,y) wird pro LUC-Typ u über die Umgebung (rx_u, ry_u) aufsummiert.

Nach Berechnung der Wahrscheinlichkeiten startet die **CLUE-Iteration**. Diese besteht aus einer WHILE-Schleife, mit 3 FOR-Schleifen im Inneren, die für jede Raster-Zelle $i = (x,y)$ die Gesamt-Wahrscheinlichkeit $TPROB_{i,U}$ pro LUC-Typ u errechnet. Statt diese – wie in 2.3.2 dargestellt – zunächst für alle LUC-Typen zu berechnen und anschließend den Typ U^* mit der größten Wahrscheinlichkeit herauszusuchen, wird immer nur die „aktuell beste“ Wahrscheinlichkeit (und der entsprechende LUC-Typ) gespeichert[9]. Diese Vorgehensweise spart einige unnötige Schleifen-Durchläufe. Dies gilt ebenso für die bereits angesprochene *on-the-fly*-Strategie zur Bestimmung der Zellen,

[9] Denn nur diese Informationen sind für die LUC-Allokation relevant.

die ihre LUC nicht wechseln dürfen. Diese wird im XULU-CLUE in zwei Teile gesplittet:

- **`checkLUCCGeneralAllowed(..)`**
  Prüft aufgrund der *Area Restrictions* (R10) und der *Conversion Elasticity* (R9), ob ein genereller Wechsel einer Zelle möglich ist. Wird als Abbruchkriterium bereits unmittelbar *vor* der dritten FOR-Schleife eingesetzt (Zeile 10 in Abb. 6.2).

- **`checkLUCCAllowed(..)`**
  Prüft aufgrund der *Conversion Elasticity* (R9), *LU-Conversion Matrix* (R8) und *LU History* (R7/R19) ob eine Zelle in einen *bestimmten* LUC-Typ wechseln darf. Wird als Ausschlusskriterium unmittelbar *nach* der dritten FOR-Schleife eingesetzt (Zeile 12 in Abb. 6.2).

Wie in Abschnitt 2.3.2 beschrieben ist, wird der Ablauf der Iteration durch die Variablen $ITER_U$ bestimmt. Die **Neuberechnung der Iterationsvariablen** erfolgt *on-the-fly* während der Überprüfung der Iterations-Bedingung (Zeile 4 in Abb. 6.2).

```
 1  Berechne LUC-Wahrscheinlichkeiten (R20)
 2  Berechne Nachbarschafts-Wahrscheinlichkeiten (R21)
 3
 4  WHILE Bedarf noch nicht gedeckt DO
 5    Initiiere ein "Iterationsschritt gestartet"-Event
 6    FOR ALL Rasterzeilen x DO
 7      FOR ALL Rasterspalten y DO
 8        bestLUC  = aktuelle LUC
 9        bestProb = 0
10        IF genereller Wechsel für (x,y) erlaubt THEN
11          FOR ALL LUC-Typen u DO
12            IF Wechsel für (x,y) nach u erlaubt THEN
13              Berechne TPROB für Zelle (x,y) und LUC-Typ u
14              IF TPROB > bestProb THEN
15                 bestProb = TPROB
16                 bestLUC  = u
17              FI
18            FI
19          OD
20        FI
21        Setze LUC für Zelle (x,y) auf bestLUC (R23)
22      OD
23    OD
24    Initiiere ein "Iterationsschritt beendet"-Event
25  OD
```

Abbildung 6.2: Ein CLUE-Zeitschritt in Pseudocode.

6.1.3 GUI

Das CLUE-Modell erfordert eigentlich keine besonderen Statusausgaben oder Steuerungsfunktionen[10]. Trotzdem wurde zur Evaluation der Schnittstelle eine modell-spezifische GUI implementiert (siehe Abb. 6.3). Hierbei war jedoch nicht das gesamte Fenster zu erstellen, sondern lediglich die modell-spezifischen Status- und Steuerungselemente. Deren Integration in die XULU-Applikation (in diesem Fall als eigenes Fenster) erfolgt durch die GUI des XULU-Modell-Managers.

Neben der Gesamt- und Durchschnittsabweichung der aktuell modellierten Bedeckung vom geforderten Bedarf, zeigt die CLUE-GUI auch die Abweichung für jeden einzelnen LUC-Typ[11]. Die Darstellung in Form von Balken-Diagrammen ist für den Anwender wesentlich anschaulicher, als eine Ausgabe im textuellen Status-Feld des generischen Kontrollfensters (Abb. 6.1). Darüberhinaus bietet das modell-spezifische

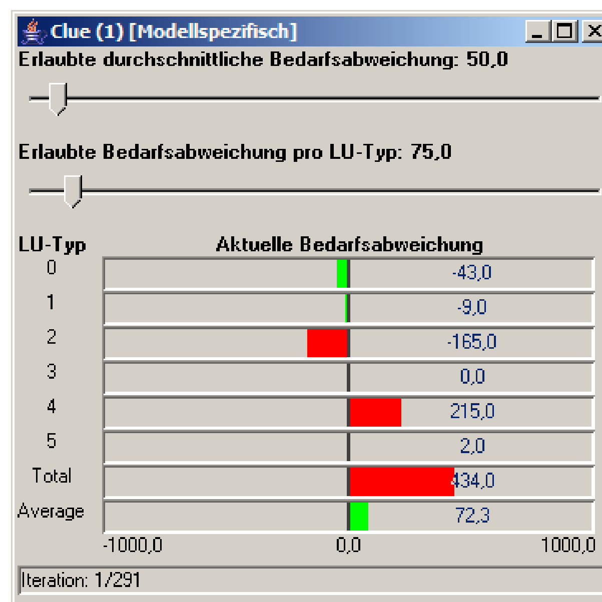


Abbildung 6.3: Die modell-spezifische GUI des CLUE-Modells.

UI zwei Schiebe-Regler, über die der Anwender die CLUE-Iteration auch *während* des Modellablauf steuern kann. Für das Ende der Iteration müssen (gleichzeitig) zwei Bedingungen erfüllt sein. Die Abweichung zwischen Bedarf und modellierter LUC ...

... muss *für jeden* LUC-Typ unterhalb eines Toleranzwertes liegen.

... muss *im Durchschnitt* (über alle LUC-Typen) unterhalb eines Toleranzwertes liegen.

[10] CLUE-S besitzt lediglich einen Start/Stop-Button und einen Fortschritts-Balken, auf den jedoch auch verzichtet werden könnte.

[11] Je nach Wert der Ressource *Type of tolerance expressions* in prozentualen oder absoluten Werten. Im IMPETUS-Szenario werden absolute ha-Werte verwendet.

Kommt es zu Situationen, in denen es dem Modell nicht gelingt, die beiden Bedingungen zu erfüllen, da z.B. ein einzelner LUC-Typ permanent (ein wenig) oberhalb der Toleranz liegt, hat der Anwender die Möglichkeit manuell einzugreifen, in dem er die Toleranz (kurzzeitig) erhöht, um die Iteration zum Abschluss zu bringen. Anschließend kann er sie für den folgenden Zeitschritt wieder zurück setzen. Diese Option bildet für den Modell-Anwender eine nützliche Erweiterung gegenüber der original CLUE-Applikation.

Die Details der GUI-Implementierung spielen für dieses Kapitel eine untergeordnete Rolle. Von Bedeutung ist jedoch, dass `ClueModelGUI` als *Inline-Klasse* von `ClueModel` implementiert ist. Hierdurch besteht für die Steuererungselemente ein direkter Zugriff auf die entsprechenden interen (privaten) Variablen/Objekte der Modell-Klasse.

6.2 Anwendung

Um die beschriebene CLUE-Implementierung in XULU auf ein konkretes Untersuchungsgebiet anzuwenden, sind im Wesentlichen drei Schritte durchzuführen, bevor das Modell gestartet werden kann[12]:

1. Modell-Instanz in die Plattform (Modell-Manager) laden
2. Für die Modell-Ressourcen benötigte Daten in den Datenpool laden
3. Daten-Objekte den Ressourcen zuordnen und Modell initialisieren

6.2.1 Modell-Instanz laden

Über den Menüpunkt *Datei ⇒ Neu ⇒ Modell* wird eine Modell-Instanz in den XULU-Modell-Manager geladen. Hierbei werden dem Anwender alle in der XULU-Registry registrierten Modelle angezeigt. Für CLUE ist die Klasse `ClueModel` auszuwählen und ein Name für die Modell-Instanz anzugeben[13]. Anschließend erscheint das Modell im Übersichtsfenster (Abb. 6.4). Über einen Doppelklick auf das CLUE-Modell wird das Steuerungsfenster angezeigt, in dem ersichtlich wird, welche Ressourcen das Modell benötigt (Abb. 6.1).

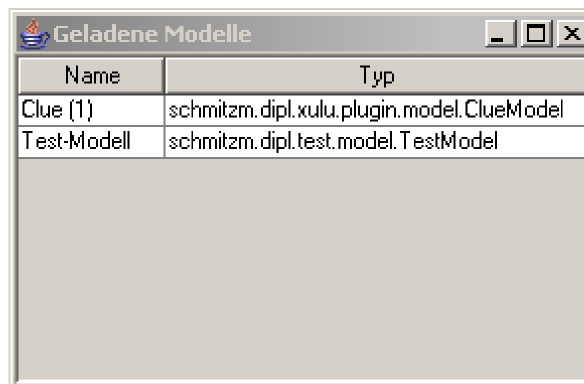


Abbildung 6.4: Im Fenster des Modell-Managers sind alle geladenen Modelle aufgelistet.

[12] Dabei wird davon ausgegangen, dass alle notwendigen Klassen (Modell, Datentypen, Factorys, usw.) bereits in der XULU-Registry eingetragen sind.

[13] Der Name wird dann wichtig, wenn *mehrere* Instanzen einer Modell-Klasse in die Plattform geladen werden.

6.2.2 Daten laden

Nachdem durch das Steuerungsfenster ersichtlich wird, welche Ressourcen das Modell benötigt, müssen die entsprechenden Daten im XULU-Datenpool erzeugt werden. Im Falle des in Anhang E beschriebenen CLUE-Datensatzes müssen 6 Objekte importiert werden (R1-R18, R22)[14]. Diese sind in Tabelle 6.2 dargestellt. Beim Import der Multi-Grids ist darauf zu achten, dass die Dateien in der richtigen Reihenfolge angegeben werden, da sie entsprechend in der Grid-Liste hinterlegt werden! Nach dem Daten-Import sollte der Anwender die Objekte aussagekräftig umbenennen (z.B. laut Tabellen-Spalte 1).

Daten-Objekt	Import-Routine (Factory)	Datei(en)
ClueModelParameter	Dyn. XULU-Objekt aus ASCII-File	`ClueModelParameter.dxo`
Base LU Scenario	SingleGrid aus ArcInfoASCII	`cov_all.0`
Base LU History	SingleGrid aus ArcInfoASCII	`age.0`
Area Restrictions	SingleGrid aus ArcInfoASCII	`region1.fil`
Static Driving Forces	MultiGrid aus ArcInfoASCII	`sc1gr0.fil` .. `sc1gr7.fil`
Dynamic Driving Forces	MultiGrid aus ArcInfoASCII	`sc1gr0.0` .. `sc1gr0.25` `sc1gr1.0` .. `sc1gr1.25`

Tabelle 6.2: Aus dem CLUE-Datensatz erzeugte XULU-Objekte.

Darüberhinaus werden 5 „neue“ Objekte für die Modellausgabe benötigt (R19-R21, R23-R24). Diese können komplett neu (über entsprechende Factorys) erzeugt werden. Jedoch müsste dann für jedes Grid das komplette Raster-Format angegeben werden (Abb. 6.5). Der Datenpool (bzw. die Instanziierungs-Factorys) bietet statt dessen die Möglichkeit, neue Objekte als *strukturelle Kopie* eines anderen Objekts zu erzeugen, was dem Anwender einigen Aufwand erspart. Tabelle 6.3 zeigt die für CLUE anzulegenden Objekte, samt den beim Datenimport erzeugten Objekten, die als Vorlage dienen können. Wie die Import-Objekte, sollte der Anwender auch die neu erzeugten Objekte nach dem Kopieren aussagekräftig umbenennen.

Daten-Objekt	Datentyp	Vorlagen-Objekt
Out: Actual LU History	`SingleGrid`	Base LU Scenario
Out: LU Probabilities	`MultiGrid`	Static Driving Forces
Out: Neighborhood Probabilities	`MultiGrid`	Static Driving Forces
Out: Actual LU Grid	`SingleGrid`	Base LU Scenario
Out: Step Results	`MultiGrid`	Static Driving Forces

Tabelle 6.3: Für das CLUE-Modell benötigte XULU-Objekte.

[14] R22 stellt eigentlich eine Ausgabe-Ressource dar. Aus technischen Gründen wird diese jedoch zusammen mit anderen Daten „importiert“.

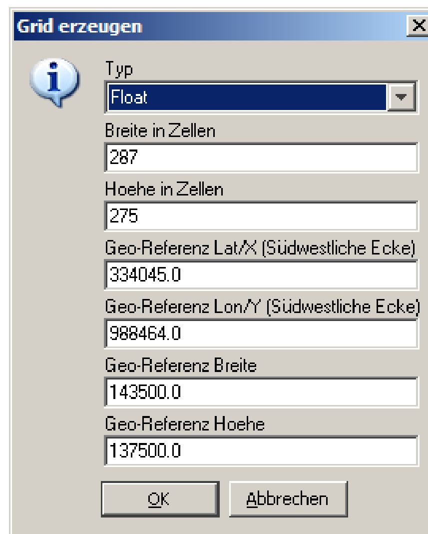


Abbildung 6.5: Um ein neues Raster zu erzeugen, müssen eine Reihe von Parametern spezifiziert werden.

6.2.3 Ressourcen zuordnen und Modell initialisieren

Eine geeignete Zuteilung der Ressourcen für das CLUE-Modell kann Abbildung 6.1 (Seite 99) entnommen werden. Dabei werden dem Anwender alle prinzipiell passenden Objekte aus dem Datenpool vorgeblendet[15]. Er muss lediglich noch die gewünschten Objekte auswählen. Durch den *Init*-Button wird die Ressourcen-Zuteilung auf Korrektheit geprüft und die Modell-Instanz (sowie die CLUE-spezifische GUI, Abb. 6.3) initialisiert.

6.2.4 Modell-Ergebnisse verarbeiten

Nach Beendigung des Modellablaufs (automatisch oder manuell durch *Stop*-Button) können die Modell-Ergebnisse in einem Visualisierungstool betrachtet oder durch Datenexport gesichert werden. Abbildung 6.6 zeigt exemplarisch die Ergebisse des ersten und zweiten Modellschritts (Jahr 1 und 2). Auf eine semantische Erläuterung dieser Ergebnisse möchte ich an dieser Stelle verzichten.

Es sollte jedoch erwähnt werden, dass es im Rahmen der Diplomarbeit leider *nicht* gelungen ist, ein exaktes Abbild der original CLUE-Arbeitsweise zu schaffen. Die implementierten Algorithmen unterscheiden sich an einer mir bisher unbekannten Stelle. Zwar scheint es sich dabei nur um geringfügige Differenzen zu handeln[16], diese führen jedoch zu einer deutlichen Abweichung im Modell-Ergebnis. Abbildung 6.7 zeigt die Ergebisse des Original-CLUE für den ersten und zweiten Modellierungsschritt.

[15] Diese Entscheidung wird vom Modell-Manager-UI auf Basis des über die Ressourcen definierten Datentyps getroffen.

[16] Die Berechnung der Wahrscheinlichkeiten funktioniert korrekt. Vermutlich liegt der Unterschied in den Ausschluss/Abbruchkriterien.

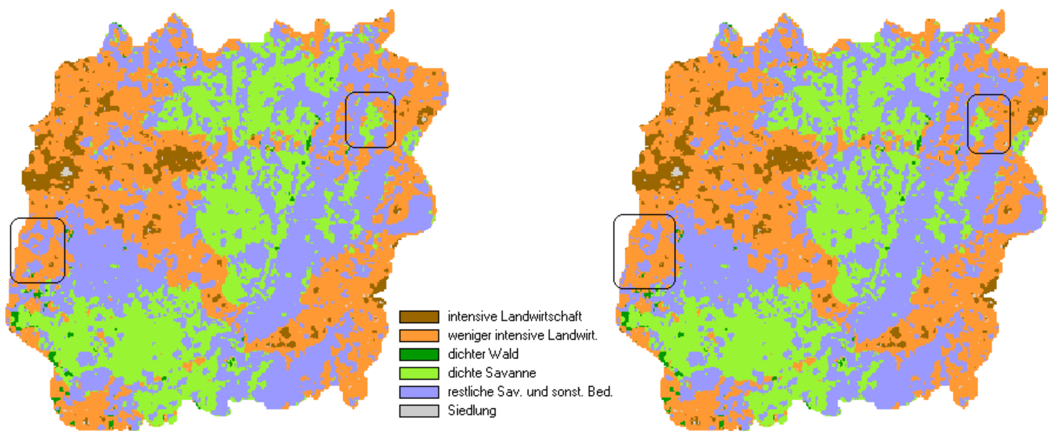


Abbildung 6.6: Die Ergebnisse des XULU-CLUE für das erste und zweite simulierte Jahr.

Die Ursache des Problems liegt darin, dass mir zur Implementierung von CLUE keine exakte Spezifikation vorlag. Zunächst habe ich mich an die öffentlichen Dokumentationen [37, 38, 39] gehalten. Da die internen Abläufe darin jedoch nicht detailliert und vollständig aufgeführt sind (insbesondere die exakte Berechnung und Verwendung der Iterationsvariablen $ITER_U$), kam es zu deutlichen Unterschieden im Modellablauf. Auf Nachfrage bei CLUE-Entwickler PETER VERBURG, erhielt ich Einblick in die original CLUE-Sourcecodes. Daraufhin konnte ich die Arbeitsweise des XULU-CLUE zwar verbessern, zu demselben Ergebnis kommt es dennoch nicht.

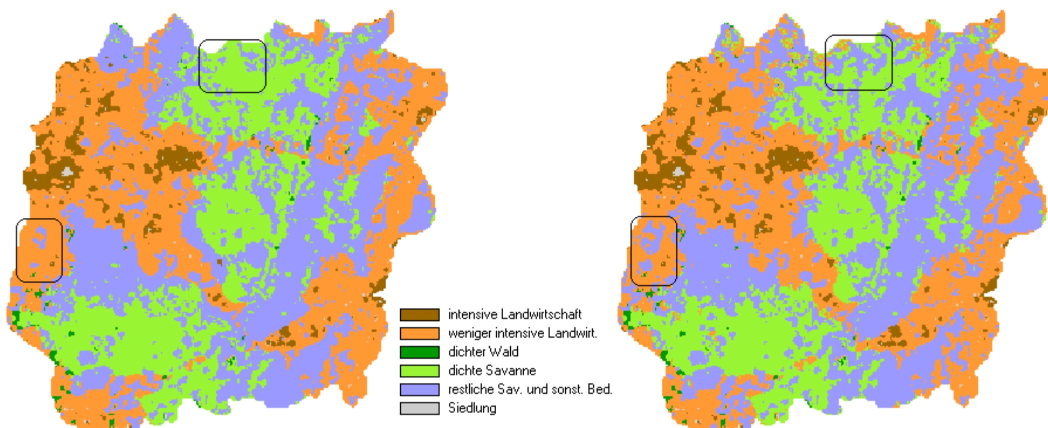


Abbildung 6.7: Die Ergebnisse des Original-CLUE für das erste und zweite simulierte Jahr. Veränderungen sind u.a. in den markierten Bereichen erkennbar.

Da das Hauptaugenmerk dieser Diplomarbeit auf der generischen Modellierungsplattform XULU liegt und nicht in einer 1:1-Nachbildung des CLUE-Modells, habe ich diesen Umstand vernachlässigt. Erwähnenswert ist jedoch der Punkt, dass die XULU-Variante das Ende eines Zeitschritts meist nach wenigen 100 Iterationsschritten erreicht, während das Original-CLUE in der Regel über 1000 Iterationen benötigt. Eine qualitative Bewertung, ob das – durch weniger Iterationsschritte generierte – Modellergebnis vergleichbar ist mit dem des Original-CLUE (oder besser, oder schlechter), konnte im Rahmen dieser Diplomarbeit noch nicht durchgeführt werden.

6.3 Bewertung

6.3.1 Phase der Modell-Programmierung

Die Integration des Modell-Algorithmus in die XULU-Schnittstelle `XuluModel` verlief problemlos. Die Schnittstelle verursachte diesbezüglich keinerlei Einschränkungen. Durch die von der Plattform zur Verfügung gestellten *modell-unabhängigen* Basis-Implementierungen der Schnittstelle (vgl. 5.6) reduzierte sich die Implementierung des CLUE-Algorithmus auf die eines einzelnen Zeitschritts. Funktionen, wie das zeitschrittweise Voranschreiten im Modellablauf und das Initiieren der entsprechenden Ereignisse (an den XULU-Event-Manager), werden bereits durch die Oberklassen implementiert und mussten für die Modell-Implementierung nicht mehr berücksichtigt werden. Einschränkungen an den Modell-Ablauf entstanden durch die Verwendung von Oberklassen jedoch nicht.

Das Ressourcen-Konzept vereinfacht das generische Handling der Modell-Daten sowohl auf Plattform-Seite (Modell-Manager), als auch auf Modell-Seite erheblich. Der Content-Manager des Modells muss nur die Art der benötigten Daten festlegen, alles weitere – z.B. wie die Ressourcen mit Daten „befüllt“ werden – erledigt der Modell-Manager generisch und ist im Rahmen der Modell-Implementierung irrelevant. Jedoch bestätigt der „Feldversuch“ die in Abschnitt 5.6.1 geäußerte Befürchtung, dass die „Verwaltungsaufgaben“ – wie die Definition der Ressourcen und der Check auf Ressourcen-Konsistenz – im Vergleich zum eigentlichen Algorithmus sehr viel Source-Code einnehmen. Dieser Effekt verstärkt sich dadurch, dass bei der CLUE-Implementierung darauf geachtet wurde, klare Strukturen zu bilden (vgl. 6.1.1). Zwischen Algorithmus-Code und Verwaltungs-Code besteht ein 1:1-Verhältnis[17]. Zwar ist der Code des Content-Managers nicht kompliziert, aufgrund der 24 Ressourcen entstand jedoch ein erheblicher „Schreibaufwand“ (vgl. 5.6.1). Hieraus entspringt der Gedanke, XULU um eine Art „Code-Generator“ zu erweitern, der diese Aufgabe automatisiert.

Doch auch ohne einen solchen Code-Generator, kann XULU der zentralen Entwickler-Anforderung nachkommen, Modellideen „schnell einmal ausprobieren“ zu können (E2, E4). Betrachtet man das Problem nämlich genauer, so stellt man fest, dass sich „schnell einmal ausprobieren“ auf die Phase der *Modell-Entwicklung* bezieht. In dieser kann jedoch z.B. auf aufwendige Konsistenz-Checks der Ressourcen verzichtet und auf den *Good-Will* des Anwenders[18] vertraut werden. Genauso können während der Entwicklung weniger (aber dafür komplexere) Ressourcen-Objekte verwendet werden, auch wenn dies zu Lasten der Flexibilität geht (vgl. Ende Abschnitt „WAHL DER RESSOURCEN“, 6.1.1). Erst wenn das Modell einem breiteren Anwenderfeld zugänglich gemacht wird (also am Ende der Entwicklungsphase), können flexiblere Ressourcen und genauere Ressourcen-Checks implementiert werden. Es ist also durchaus möglich,

[17] Jeweils ca. 800 Zeilen Sourcecode für CLUE-Algorithmus und Content-Manager, sowie ca. 500 Zeilen für die CLUE-spezifische GUI.

[18] Der Anwender ist in dieser Phase ja meist der Entwickler selbst!

Modellideen in XULU ad hoc zu realisieren, wenn man (zunächst) auf sichere Datenkonsistenz und Flexibilität verzichtet.

6.3.2 Modell-Steuerung

Im Rahmen der Modell-Anwendung erweist sich die Modell-Steuerung und das Prinzip der Ressourcen-Zuordnung als sehr flexibel. Um im Original-CLUE zwischen zwei Start-Szenarien zu wechseln, ist das Umkopieren von Dateien (`cov_all.0`) notwendig. Um zwischen alternativen Konfigurationen der LU-Conversion-Matrix zu wechseln, muss eine Datei manuell abgeändert werden. In XULU können hingegen die alternativen Datensätze *parallel* in den Datenpool geladen und im Steuerungs-UI (Abb. 6.1) ohne jeglichen Aufwand zwischen den jeweiligen Ressourcen hin und her gewechselt werden (vgl. 6.1.1).

Insbesondere während der Modell-Entwicklung (Implementierung) zeigen sich die Vorzüge der in Abschnitt 5.1 beschriebenen XULU-Skripte. Wird das Importieren der Daten in den Datenpool, welches – aufgrund des XULU-Neustarts – nach jeder Modell-Änderung vorzunehmen ist, manuell durchgeführt, entsteht ein recht hoher Zeitaufwand. Das in Abschnitt 5.1 und Anhang B dargestellte Datenpool-Skript vereinfacht diese – für den Modell-Entwickler zunehmend lästige – Tätigkeit entscheidend. Statt 6 einzelner Import-Vorgänge – beim Import der dynamischen Driving Forces sind zudem 52 Dateien auszuwählen! – und 5 einzelner Objekt-Erzeugungen (vgl. 6.2.2), reicht *ein einziger* Skript-Aufruf aus, um alle diese Vorgänge durchzuführen.
Die Integration einer ähnlichen Vorgehensweise für die Ressourcen-Zuteilung sollte unbedingt im Rahmen eines baldigen Evolutionsschritts der XULU-Plattform realisiert werden, da dies die Handhabung für den Modell-Entwickler weiter vereinfacht.

6.3.3 Effizienz-Vergleich mit dem Original-CLUE

Die Modellierungsplattform XULU bietet die Möglichkeit, effiziente Datenstrukturen, die z.B. speziell auf häufigen Zugriff zugeschnitten sind, global zu integrieren. Über das Plugin-Konzept (vgl. 4.2 und 5.3) ist der Entwickler in der Lage, diese in XULU einzubinden, ohne die Plattform selbst anzupassen. Und mit Hilfe des Factory-Konzepts (vgl. 4.8.1 und 5.5.2) können sie automatisch (und transparent) für alle in XULU implementieren Modelle eingesetzt werden.

Im Rahmen der Implementierung des XULU-CLUE wurde jedoch der Aspekt spezieller, effizienter Datentypen vernachlässigt und primär Wert darauf gelegt, die *Arbeitsweise* des Original-CLUE nachzubilden. Eine „übersichtliche“ und gut nachvollziehbare Gestaltung des Codes verbessert zwar die Lesbarkeit (und Wartbarkeit), führt jedoch zu vielen verschachtelten Methoden-Aufrufen. Beide Punkte wirken sich negativ auf die Effizienz aus:

> Das in C/C++ implementierte CLUE-S arbeitet wesentlich effizienter als die XULU-Variante. Trotz weniger Iterationen benötigt das XULU-CLUE ca. 3mal mehr Rechenzeit für die Simulation eines Zeitschritts (Jahr) als CLUE-S[19].

Es ist jedoch zu erwarten, dass dieses Verhältnis durch die Entwicklung eines effizienteren Raster-Datentyps erheblich verbessert werden kann[20]. Die zur Zeit in XULU verwendete Raster-Implementierung (`WritableGridCoverage`) setzt auf der GEOTOOLS-Klasse `GridCoverage` auf, welche intern wiederum auf `WritableRaster` (JAVA-Standard) basiert. Da jede Klasse ihre eigenen Zugriffsmethoden verwendet, finden sehr viele Aufrufe statt, die nach ein paar zusätzlichen Berechnungen auf die entsprechenden Methoden der Basis-Klasse verweisen.

Durch diese – in JAVA häufig zum Einsatz kommenden – Objekt-Strukturen und -Hierarchien entstehen wesentlich mehr interne Methoden-Aufrufe, als dies beim direkten Speicher/Array-Zugriff des in C/C++ implementierten CLUE-S der Fall ist. Zudem wird innerhalb der oben genannten Raster-Klassen sehr häufig mit überladenen Methoden gearbeitet (Polymorphie), von denen nur eine die eigentliche Funktionalität implementiert. Die anderen formatieren lediglich die Eingabeparameter um (oder setzen Default-Werte), um anschließend eine überladene Methode aufzurufen:

Beispiel:

```
private Object[][] data = ...

public Object getValue(double x, double y) {
    return getValue( new Point2D.Double(x,y) );
}

public Object getValue(java.awt.geom.Point2D p) {
    // Geo-Koordinaten in Raster-Koordinaten umrechnen
    int rx = ..
    int ry = ..
    return data[rx][ry];
}
```

Wie ein kleiner Test gezeigt hat (vgl. Anhang F), entstehen insbesondere dann extreme zeitliche Mehr-Kosten, wenn dabei neue (temporäre) Objekte erzeugt werden[21]. Weiterer Zeit-Aufwand entsteht, wenn diese Hilfs-Objekte wieder aus dem Speicher entfernt werden müssen (automatisch durch die JAVA-*Garbage-Collection*).

Aus der Sicht objekt-orientierter Programmierung ist die geschilderte Art der Polymorphie sehr sinnvoll, da eine grosse Methoden-Vielfalt ohne redundanten Code geschaffen wird. Auf die Effizienz wirkt sich dies jedoch negativ aus, wenn die betreffende Routine sehr häufig ($\geq$ 100000mal) ausgeführt werden muss. Da dies im Falle

[19] Auf einem TARGA TRAVELLER Notebook mit MOBILE AMD ATHLON 64 PROCESSOR (1.99GHZ) benötigt das XULU-CLUE ca. 27 Minuten für die Simulation von 5 Jahren, CLUE-S dagegen nur 6 Minuten.

[20] Wie oben angesprochen, muss die eigentliche CLUE-Implementierung dafür *nicht* verändert werden!

[21] im Beispiel: `new Point2D.Double(x,y)`

des CLUE-Modells insbesondere auf den Raster-Datentyp zutrifft, sollte bei der Neu-Implementierung dieses Typs ein gewisses Mass an Redundanz in Kauf genommen werden und statt geschachtelter und überladener Methodenaufrufe, direkt auf die Speicherstruktur (z.B. Array) zugegriffen werden.

Abschließend möchte ich anmerken, dass auch die direkte Modell-Algorithmik noch einigen Effizienz-Spielraum nach oben enthält, sowohl beim Original-CLUE, als auch beim XULU-CLUE. Beispielsweise ist die Berechnung der LUC-Wahrscheinlichkeiten sehr naiv implementiert und erfolgt in jedem Zeitschritt über 4 geschachtelte FOR-Schleifen. Diese laufen insbesondere über *alle* Driving Forces, auch wenn sich (wie im Fall des IMPETUS-Szenarios) zwischen den Zeitschritten nur 2 Driving Forces verändern. Bei der Berechnung der Nachbarschaftseinflüsse sind es sogar 5 FOR-Schleifen und die Umgebung wird für *jede* Rasterzelle *komplett* betrachtet, obwohl ein Teil dieser Nachbarschaft für die vorangegangene Zelle bereits durchlaufen wurde (Abb. 6.8). Für beide Berechnungen könnten unter Umständen geschicktere Algorithmen gefunden werden.

	A	B	C	D	E	F	G	H
1	1	2	1	4	1	2	2	3
2	2	3	2	3	4	4	1	1
3	1	2	3	4	1	3	2	2
4	5	1	5	1	2	1	3	1
5	5	2	1	4	1	2	4	2
6	2	4	2	3	2	4	3	3
7	5	2	1	4	3	2	1	2
8	4	2	3	1	3	2	4	1

	A	B	C	D	E	F	G	H
1	1	2	1	4	1	2	2	3
2	2	3	2	3	4	4	1	1
3	1	2	3	4	1	3	2	2
4	5	1	5	1	2	1	3	1
5	5	2	1	4	1	2	4	2
6	2	4	2	3	2	4	3	3
7	5	2	1	4	3	2	1	2
8	4	2	3	1	3	2	4	1

Abbildung 6.8: Bei einem Nachbarschaftsradium von 2 Zellen, werden die schattierten Bereiche sowohl für die Zelle D4, als auch für E4 betrachtet.

6.3.4 Zusammenfassung

Der objekt-orientierte und komponenten-basierte Ansatz der XULU-Plattform hat sich bezüglich der Flexibiltät der Modell-Implementierungen und Datentypen bewährt. Das CLUE-Modell konnte – von der technischen Seite aus gesehen – problemlos in die XULU-Plattform integriert werden. Durch die Realisierung als JAVA-Klasse entstand zwar ein größerer Implementierungsaufwand, als dies bei einer interaktiven *Drag&Drop*-Modellerstellung der Fall wäre. Im Gegensatz zur Implementierung einer kompletten Modell-Anwendung gestaltete sich der Aufwand jedoch vergleichsweise gering. Zudem stand für die Umsetzung die komplette Mächtigkeit der Programmiersprache JAVA zur Verfügung. Mit Hilfe der durch XULU bereitgestellten Basis-Implementierungen der Schnittstellen beschränkte sich die Umsetzung der CLUE-Semantik weitestgehend auf den Algorithmus. Die Definition und Verwaltung der benötigten Ressourcen verursachte hingegen den bereits vermuteten Mehraufwand, da

versucht wurde, zum einen möglichst viele Fehlerquellen durch eine gute Konsistenz-Prüfungen abzufangen und zum anderen den Content-Manager übersichtlich zu gestalten. An dieser Stelle besteht Erweiterungsbedarf, z.B. durch einen Code-Generator.
Die Ressourcen-Zuteilung erwies sich im Rahmen der Modell-Anwendung als sehr flexibel, während der Modell-Entwicklung (*trial-and-error*) jedoch mitunter als zeitaufwändig. Aufgrund der offenen Plugin-Schnittstellen kann hier jedoch Abhilfe geschaffen werden, ohne die XULU-Anwendung selbst anpassen zu müssen (Ressourcen-Zuordnung via Skript).

Gegenüber der modell-spezifischen Applikation CLUE-S wurden Effizienz-Nachteile festgestellt, die zum einen auf die direkte Modell-Implementierung und zum anderen auf die gängigen OO-Strategien zurückzuführen sind (Vererben, Überladen), die insbesondere auch bei der Implementierung des XULU-Raster-Datentyps verfolgt wurden. Der Vorteil von XULU besteht jedoch darin, dass effizienz-steigernde Massnahmen – wie spezielle Datentypen oder verteiltes Arbeiten auf parallelen Rechnern – unabhängig vom Modell realisiert werden können und transparent für *alle* in XULU implementierten Modelle verwendbar sind. Im Fall modell-spezifischer Applikationen müssen alle bestehenden Programme abgeändert werden. Unter Umständen ist – aufgrund veränderter Datentypen – sogar die Umstrukturierung der Modell-Algorithmik erforderlich.

Zusammenfassung und Ausblick

Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung und Realisierung einer allgemeinen JAVA-Plattform, welche die Implementierung neuer Simulations-Modelle wesentlich vereinfacht.

Der Fokus liegt dabei auf dem Anwendungsgebiet der modernen Landnutzungsmodellierung. Hierbei ist Simulation unumgänglich, da sie sich mit der Erforschung möglicher Zukunftszenarien beschäftigt. Zunächst wurde die Vielfältigkeit betrachtet, mit dem in diesem Anwendungsbereich Modellierung betrieben werden kann. Es wurde festgestellt, dass sich viele Ansätze überschneiden, es aber kein einheitliches Schema gibt, wie die Modelle aufgebaut sind oder vorgehen. Es wurden zwei konkrete Modelle, bzw. Modell-Applikationen vorgestellt: MAPMODELS und CLUE-S. Diese werden von der RSRG im Rahmen des IMPETUS-Projekts für Benin (Westafrika) eingesetzt und dienen in dieser Arbeit dazu, einige der gängigen Arbeitsweisen moderner LUC-Modelle zu verdeutlichen. Anhand des CLUE-Modells wird zudem die Funktionalität der in dieser Arbeit realisierten Plattform evaluiert (Kap. 6). Zunächst werden die beiden Anwendungsbeispiele jedoch dazu genutzt, die Probleme aufzuzeigen, welche die Verwendung bestehender Modell-Applikationen und die Entwicklung neuer Landnutzungsmodelle erschweren. Für die Entwicklung der Modellierungsplattform XULU standen vor allem technische Probleme im Vordergrund. Derzeitige Modell-Applikationen sind meist als eigenständige und unabhängige Programme implementiert. Eine Kombination von bestehende Modellen oder Modell-Teilen ist somit nur schwer möglich. Bereits der Datenaustausch gestaltet sich schwierig, da jedes Programm seine eigenen Daten- und Dateiformate verwendet. Zudem muss für ein neues Modell sehr häufig eine vollständige Anwendung (GUI, Datenimport, usw.) programmiert werden.

Diese Probleme motivieren die Entwicklung der XULU-Plattform, welche die wesentlichen, für die Modellierung benötigten Anwendungsfunktionen – wie Datenverwaltung, Visualisierung, Modell-Steuerung und entsprechende grafische Benutzerschnittstellen – generisch realisiert und zur Verfügung stellt, so dass sie im Rahmen der Modell-Entwicklung und -Implementierung nicht mehr berücksichtigt werden müssen. Aus den erarbeiteten Anforderungen von Modell-Anwender und -Entwickler (Kap. 3), wurde das Konzept einer *Stand-Alone*-Applikation entwickelt, welche für verschiedenste Modelle verwendet werden kann. Über die dynamische Integration von Datentyp-,

Visualisierungs- und Import/Export-Plugins kann XULU zu einer Modellierungsplattform für unterschiedlichste Anwendungsgebiete – also auch über das der Landnutzungsmodellierung hinaus – erweitert werden (4.2). Modelle werden als fest programmierte JAVA-Klassen implementiert und ebenfalls als Plugins in XULU eingebunden.

Auch beim Entwurf der internen Abläufe stand die Erweiterbarkeit der Plattform im Blickpunkt. Hierzu wurde XULU in einzelne Module unterteilt, die über feste Schnittstellen miteinander agieren. Neben den Plugins können somit auch die einzelnen internen XULU-Komponenten ohne grosse programmtechnische Anpassungen erweitert oder ausgewechselt werden. An verschiedenen Stellen konnte auf bestehenden *Design Patterns* aufgebaut werden.
Eine zentrale Komponente der XULU-Datenverwaltung bildet der *Datenpool*, den alle in die Plattform geladenen Modelle *gemeinsam* nutzen. Der Datenaustausch zwischen den XULU-Modellen erfolgt über die Verwendung einheitlicher Datentypen und den Zugriff auf dieselben Daten-Objekte. Der Geo-Datentyp Raster motiviert die Trennung von Datendefinition, Datenorganisation und Datenerzeugung. Hierbei wird auf das *Abstract Factory Pattern* zurückgegriffen, welches um Import und Export zu einem umfangreicheren *Factory-Konzept* erweitert wurde (4.8).
Der XULU-Modell-Manager schreibt einen generellen Modellierungsablauf vor, der die mögliche interne Modell-Algorithmik jedoch nicht einschränkt. Die Philosophie der Plattform besteht darin, dass der Anwender individuell bestimmt, welche konkreten Objekte ein Modell für die Simulation verwendet, welche Objekte auf welche Weise visualisiert werden und wann ggf. eine Aktualisierung erfolgen soll. Deshalb werden die Modelle von der restlichen Plattform isoliert, und ein direkter Zugriff auf die einzelnen Komponenten (Datenpool, Visualisierung, usw.) verhindert (4.4). Speicherverwaltung und Datenorganisation bleiben im Verantwortungsbereich der Plattform und sind für die Modelle transparent. Somit wird erreicht, dass Datentyp-Implementierungen (z.B. gegen zeiteffizientere) ausgetauscht werden können, ohne ein Modell anpassen zu müssen. Das *Ressourcen-Konzept* vereinheitlicht den Daten-Zugriff der Modelle und sieht vor, dass ein Modell lediglich die Art der benötigten Daten spezifiziert. Mit welchen konkreten Objekten aus dem Datenpool die jeweilige Simulation parametrisiert wird, bestimmt der Anwender. Über das *Event-Konzept* hat der Anwender die Möglichkeit, automatische Reaktionen auf Modell-Ereignisse zu definieren (z.B. Visualisierungsupdate am Ende eines Zeitschritts). Im Zusammenhang mit dem Modellablauf wurden Probleme diskutiert, die bei komplexen Multi-Modell-Szenarien entstehen (4.4.5).
Die Entwicklung der einzelnen Plattform-Module führte zu einer allgemeinen Datentyp-Schnittstelle, die alle Datentyp-Plugins implementieren müssen (4.8.3).

Die Implementierung der XULU-Modelling-Platform setzt die wesentlichen entworfenen Konzepte um. Dabei flossen zusätzliche Aspekte der Anwenderfreundlichkeit in die Plattform ein. XULU-*ScriptInterpreter* ermöglichen es, bestimmte Abläufe innerhalb der Plattform (z.B. eine Folge von Daten-Importen) zu automatisieren.
Die Modellschnittstelle wurde im Zuge der Implementierung in einen technischen (*Content-Manager*) und einen semantischen Bereich (*Modell*) aufgeteilt (5.6). Da die Ressourcen-Zuordnung individuell durch den Anwender erfolgt, wurde die Bedeutung guter Konsistenz-Prüfungen hervorgehoben, welche im Rahmen der Modell-

Entwicklung ebenso implementiert werden müssen, wie der eigentliche Modellalgorithmus.
Die entworfene Datentyp-Schnittstelle wurde um ein *Property-Konzept* erweitert, damit gewisse Datentyp-Anforderungen, wie Zugriffskontrolle und Änderungspropagierung, zentral implementiert werden konnten. Darüberhinaus ermöglichte das Property-Konzept die Entwicklung eines *dynamischen* XULU-*Objekts* (5.5.1). Dieses kann dazu genutzt werden, individuelle Datentypen zu entwickeln, ohne neue JAVA-Klassen zu implementieren.
Für die Plugin-Schnittstellen der XULU-Plattform wurden Basis-Implementierungen erstellt, welche die Realisierung neuer Modelle und Datentypen erleichtern. Diese wurden unmittelbar dazu genutzt – neben dem Plattform-Rahmen – konkrete Plugins für das Anwendungsgebiet der Landnutzungsmodellierung zu realisieren: Hierzu gehören Datentypen zur Verwaltung von Raster- und Vektor-Daten mit entsprechenden Factorys, sowie eine Geo-Visualisierung. Grundlage bei der Umsetzung bildet die GEOTOOLS-Klassenbibliothek. Exemplarisch wurde die Implementierung der Raster-Datentypen genauer betrachtet (5.5.3).

Diese Geo-Plugins kommen im Rahmen des Evaluationsprozesses zum Einsatz, in dem ein Abbild des CLUE-Modells in XULU integriert wurde (Kapitel 6). Dieser zeigt, dass das zentrale Ziel dieser Arbeit erreicht wurde, den Implementierungsaufwand für ein konkretes Modell wesentlich zu verringern. Die entworfene Ressourcen-Philosophie erwies sich als sehr hilfreich. Im Gegensatz zu CLUE-S konnte sehr flexibel zwischen verschiedenen Ausgangsszenarien und Konfigurationen gewechselt werden. Das entwickelte Visualisierungs-Plugin ermöglichte es zudem, die Modell-Ergebnisse direkt innerhalb der XULU-Plattform zu evaluieren.
Daneben wurden im Rahmen der Evaluation einige Ansatzpunkte diskutiert, wie die komponentenbasierte und modellunabhängige Gestaltung der Plattform dazu genutzt werden kann, XULU noch benutzerfreundlicher und effizienter zu gestalten.

Ausblick

Wie der vorangegangene Abschnitt zusammenfassend darlegt, stellt die XULU-Modelling-Platform eine funktionstüchtige Applikation dar, in die mit CLUE auch bereits ein erstes Modell mit konkretem Anwendungsbezug integriert wurde. Während des Entwurfs, der Implementierung und der Evaluation wurden jedoch eine Reihe von Stellen identifiziert, an denen XULU sinnvoll erweitert und verbessert werden kann. Deshalb ist geplant, die Entwicklung von XULU durch weiterführende Projekte sukzessive fortzuführen. Hierfür gibt es drei generelle Ansatzpunkte, die durch den modularen Aufbau von XULU weitestgehend unabhängig voneinander angegangen werden können:

- **Anwendungs-Plugins:**
  Erweiterung der Plugin-Bibliothek um neue (effizientere) Datentypen, Import/Export-Routinen oder Visualisierungstools, sowie die Erschließung komplett neuer Anwendungsgebiete.

- **XULU-interne Konzepte:**
  Erweiterung des XULU-Funktionsumfangs (z.B. interaktive Datenmanipulation/Modellerstellung) und Verbesserung der Anwenderfreundlichkeit (z.B. komfortablere GUI, mehr Event-Handler, Abspeichern von Konfigurationen).

- **Modell-Entwicklung:**
  Weiterentwicklung des implementierten CLUE-Modells und Evaluation neuer Konzepte der Landnutzungsmodellierung.

Durch die enge Zusammenarbeit mit der RSRG, besteht insbesondere Interesse daran, XULU hinsichtlich der Landnutzungsmodellierung weiter zu entwickeln, vor allem, um neue Modellierungsideen evaluieren zu können. Aber auch technische Erweiterungen, wie zum Beispiel eine Datenbank- oder Internet-Anbindung (z.B. WMS, WFS) stehen im Blickfeld des Anwenderinteresses. Angedacht ist beispielsweise eine Art Monitoring- oder Frühwarnsystem, welches vollautomatisch, auf Basis täglich über das Internet aktualisierter Daten, kurzfristige Zukunftsszenarien modelliert und über bedrohliche Tendenzen informiert. Ein konkretes Anwendungsbeispiel hierfür stellt die Überwachung von Brandflächen im IMPETUS-Gebiet dar.

Um die Effizienz von XULU zu steigern, ist die Implementierung neuer Datentypen nicht der einzige Ansatz. Es stellt sich zum Beispiel die Frage, wie die Möglichkeiten des verteilten/parallelen Rechnens in XULU integriert werden können. Eine solche Erweiterung ist vor allem deshalb interessant, weil hierdurch die Modellierung in sämtlichen Anwendungsbereichen verbessert werden kann.
Gleiches gilt für die in dieser Arbeit bereits beschriebene Problematik der Multi-Modell-Szenarien. Während z.B. eine Internet-Anbindung oder neue Import/Export-Routinen relativ leicht umgesetzt werden können, gehört die Entwicklung von Strategien zur automatischen und generischen Modellsynchronisation zu den anspruchsvolleren Erweiterungsgebieten, welche die Mächtigkeit von XULU jedoch entscheidend erhöhen würden.

Abschließend sei angemerkt, dass es generell sehr interessant wäre, XULU in einem komplett anderen Anwendungsgebiet als der Landnutzungmodellierung einzusetzen, beispielsweise für finanzwirtschaftliche Simulationen oder im Rahmen der Spieltheorie. Durch solch neue Blickwinkel ergäben sich sehr wahrscheinlich noch viele Erweiterungsaspekte, die in dieser Arbeit noch unbeachtet geblieben sind.

Anhang A

XULU-Registry-Datei

Auf der nachfolgenden Seite ist ein exemplarisches Beispiel einer XULU-Registry-Datei dargestellt, die von der Klasse `XuluRegistryReader_BasicAscii` interpretiert wird. Alle `[..]`-Zeilen definieren Plugin-Sections. Leerzeilen, sowie alle Zeilen, die mit `\\` oder `#` beginnen, stellen Kommentar-Zeilen dar und werden nicht interpretiert [41].

```
//########### Klassen werden "fuer alles" registriert ###########
[Global]
//########### Klassen werden als Datentyp registriert ###########
[DataType]
schmitzm.dipl.xulu.data.DynamicXuluObject
schmitzm.dipl.xulu.plugin.data.SingleGrid
//########### Klassen als Default-Factory registriert ###########
[DefaultFactory]
schmitzm.dipl.xulu.data.DynamicXuluObject$DefaultFactory
schmitzm.dipl.xulu.plugin.io.SingleGridFactory
schmitzm.dipl.xulu.plugin.io.GridCoverageFactory
//########### Klassen als Import-Factory registriert ###########
[ImportFactory]
schmitzm.dipl.xulu.plugin.io.SingleGridFactory_ArcInfoAsciiGrid       SingleGrid aus ArcInfoASCII
schmitzm.dipl.xulu.plugin.io.GridListFactory_ArcInfoAsciiGrid         GridList aus ArcausfoASCII
schmitzm.dipl.xulu.plugin.io.MultiGridFactory_ArcInfoAsciiGrid        MultiGrid aus ArcInfoASCII
//########### Klassen als Export-Factory registriert ###########
[ExportFactory]
schmitzm.dipl.xulu.plugin.io.SingleGridFactory_ArcInfoAsciiGrid        SingleGrid in ArcInfoASCII
schmitzm.dipl.xulu.plugin.io.GridListFactory_ArcInfoAsciiGrid          GridList in ArcInfoASCII
schmitzm.dipl.xulu.plugin.io.MultiGridFactory_ArcInfoAsciiGrid         MultiGrid in ArcInfoASCII
//########### TypeMapping ###########
[TypeMapping]
// DatenTyp-Klasse                         Default-Factory-Klasse                          [Im/Export-Klassen ...]
//------------------------------------------------------------------------------------------------------------------------------
schmitzm.dipl.xulu.plugin.data.SingleGrid  schmitzm.dipl.xulu.plugin.io.SingleGridFactory  schmitzm.dipl.xulu.plugin.io.Sing...
schmitzm.dipl.xulu.plugin.data.GridList    schmitzm.dipl.xulu.plugin.io.GridListFactory    schmitzm.dipl.xulu.plugin.io.Grid...
schmitzm.dipl.xulu.plugin.data.MultiGrid   schmitzm.dipl.xulu.plugin.io.MultiGridFactory   schmitzm.dipl.xulu.plugin.io.Mult...
//########### Visualisierungstools ###########
[Visualisation]
schmitzm.dipl.xulu.plugin.vis.LayeredGeoVisualisation
//########### Modellklassen ###########
[Model]
schmitzm.dipl.xulu.plugin.model.ClueModel
//########### Skript-Interpreter ###########
[Script]
schmitzm.dipl.xulu.plugin.appl.DataScriptInterpreter_Basic   Datenpool-Skript
```

Anhang B

Dateiformat für Datenpool-Skripte

Auf der nachfolgenden Seite ist ein exemplarisches Beispiel einer ASCII-Datei dargestellt, aus der die Klasse `DataScriptInterpreter_Basic` I/O-Befehle für den XULU-Datenpool einließt und ausführt. Alle `[..]`-Zeilen definieren einen Befehl, dessen Parameter in den darauf folgenden Zeilen angegeben werden. Leerzeilen, sowie alle Zeilen, die mit `\\` oder `#` beginnen, stellen Kommentar-Zeilen dar und werden nicht interpretiert.

Befehl Parameter	
`[Create]`	**XULU-Objekt neu erzeugen**
`type = ...`	Klasse des Datentyps (Factory wird aus der Registry ermittelt)
`name = ...`	Name für das neue Objekt
`dialog = true|false`	Flag, ob Factory einen Dialog anzeigen darf (optional)
`[Import]`	**XULU-Objekt aus Datei importieren**
`factory = ...`	Bezeichnung (oder Klasse) einer registrierten Import-Factory
`name = ...`	Name für das neue Objekt
`source = ...`	*eine* Quell-Datei (Parameter kann mehrfach angegeben werden!)
`[CopyStructure]`	**Struktur eines XULU-Objekt kopieren**
`source = ...`	Name des Objekts, das kopiert werden soll
`dest = ...`	Name für das neue Objekt
`dialog = true|false`	Flag, ob Factory einen Dialog anzeigen darf (optional)
`[Rename]`	**XULU-Objekt umbenennen**
`source = ...`	Name des Objekts, das umbenannt werden soll
`dest = ...`	Neuer Name für das Objekt
`[BaseDir]`	**Basis-Verzeichnis für folgende `[Import]`-Befehle setzen**
`...`	Absolute Verzeichnis-Angabe

```
######################################################################
# Xulu-Script zum Erzeugen/Importieren der fuer Clue benoetigten Daten
######################################################################
[BaseDir] E:\Studium\Diplom\Daten\CLUE-Datensatz (Xulu)

//*************** Eingabe-Daten importieren ****************
# Modell-Parameter [Import] factory =
schmitzm.dipl.xulu.plugin.io.DynamicXuluObjectFactory_BasicStructure
source  = ClueModelParameter.dxo name    = Clue-Parameter

# Base LU Scenario [Import] factory =
schmitzm.dipl.xulu.plugin.io.SingleGridFactory_ArcInfoAsciiGrid
source  = cov_all.0 name    = Base LU Scenario

# Static Driving Forces [Import] factory =
schmitzm.dipl.xulu.plugin.io.MultiGridFactory_ArcInfoAsciiGrid
source  = sc1gr0.0 source  = sc1gr1.0 source  = sc1gr2.fil source
= sc1gr3.fil source  = sc1gr4.fil source  = sc1gr5.fil source  =
sc1gr6.fil source  = sc1gr7.fil name    = Static Driving Forces

//*************** Ausgabe-Daten erzeugen ****************
# Raster fuer aktuelle LU [CopyStructure] source  = Base LU
Scenario dest    = Out: Actual LU dialog  = false

# Raster fuer LU-Wahrscheinlichkeiten [CopyStructure] source  =
Static Driving Forces dest    = Out: LU Properties
// es muessen so viele Grids erstellt werden, wie LU-Typen
// modelliert werden
dialog  = true

# Raster fuer Nachbarschafts-Wahrscheinlichkeiten [CopyStructure]
source  = Out: LU Properties dest    = Out: Neighborhood
Properties
// zuvor erstelltes MultiGrid hat bereits die richtige Anzahl
dialog  = false
```

Anhang C

Dateiformat ARCINFOASCIIGRID

Im folgenden ist ein kleines exemplarisches Beispiel einer Raster-Datei im ARCINFOASCIIGRID-Format aufgeführt. Die ersten 6 Zeilen stellen Meta-Informationen dar (Rasterbreite und Rasterhöhe in Zellen, Geo-Position der südwestlichen Ecke in Latitude und Longitude, Breite einer Zelle in Metern, Rasterwert der für „kein Wert“ steht):

```
ncols         14
nrows         24
xllcorner     334045
yllcorner     988464
cellsize      500
NODATA_value  -9999
-9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999
-9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999
-9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999
-9999 -9999 -9999 -9999 -9999     3     4     3 -9999 -9999 -9999 -9999 -9999 -9999
-9999 -9999 -9999 -9999     5     5     5     5     4 -9999 -9999 -9999 -9999 -9999
-9999 -9999 -9999     5     5     5     5     5     5     3     3 -9999 -9999 -9999
-9999 -9999 -9999     5     5     5     5     5     5     5     2     2 -9999 -9999
-9999 -9999 -9999     5     5     5     5     2     3     5     5     5     4 -9999
-9999 -9999 -9999     5     5     5     2     2     2     5     5     4     2     4
-9999 -9999 -9999     5     5     5     2     2     2     5     5     5     4     5
-9999 -9999     4     5     5     5     5     2     2     2     5     5     4     4
-9999     4     5     5     5     5     4     2     2     2     5     5     5     5
-9999     5     5     5     5     5     5     2     2     2     5     5     5     5
-9999     5     5     5     5     5     5     4     2     4     5     5     5     5
    5     5     5     5     4     5     5     4     2     2     5     5     5     5
    5     5     5     5     2     2     2     4     4     4     2     2     2     2
    5     5     5     2     2     2     2     2     2     5     2     2     2     2
    5     5     5     5     2     2     2     2     5     5     2     2     2     2
    5     5     5     5     5     2     2     2     5     2     2     2     2     2
    5     5     5     5     5     5     5     5     5     2     2     2     2     2
    5     5     5     5     5     5     5     5     2     2     2     2     2     2
    2     2     2     2     5     5     5     5     5     5     5     2     2     2
    2     2     2     2     2     2     5     5     5     5     2     2     2     2
    2     2     2     2     2     5     5     5     5     5     2     2     2     2
```

Anhang D

Dateiformat für dynamische XULU-Objekte

Auf den nachfolgenden Seiten ist ein exemplarisches Beispiel einer ASCII-Datei dargestellt, aus der die Klasse `DynamicXuluObjectFactory_BasicStructure` eine Instanz von `DynamicXuluObject` generiert. Leerzeilen, sowie alle Zeilen, die mit `\\` oder # beginnen, stellen Kommentar-Zeilen dar und werden nicht interpretiert.

Zu Beginn der Datei werden Objekt-spezifische Parameter definiert. Hier kann (zur Zeit) nur der Name des XULU-Objekts festgelegt werden (über die Zeile `XuluObjectName = ...`).

Jede Property-Definition beginnt mit einer `[..]`-Zeile, die gleichzeitig den Namen der Eigenschaft definiert:

```
[PropertyName]
Type     = <package.class> [Typ-spez. Parameter]
DataType = <package.class>
Value    = ...
Value    = ...
...
```

`Type` definiert die Art der Eigenschaft (Skalar, Liste oder Matrix). Wird er weggelassen, wird `ScalarProperty` oder `ListProperty` angenommen, je nachdem, ob die (erste) `Value`-Zeile nur einen oder mehrere Werte enthält. `DataType` definiert den Datentyp der Werte, die in der Property gespeichert werden können (z.B. `Integer` oder `String`). Wird er weggelassen, bestimmt der erste Wert in der `Value`-Zeile den Datentyp (`Integer`, `Double` oder `String`). Die `Value`-Zeilen bestimmen den Wert, den die Eigenschaft annimmt. Für Skalar und Liste ist nur eine Zeile anzugeben. Für Listen sind die Werte durch Semikolons zu trennen.

Für Matrizen ist es notwendig, die `Type`-Zeile anzugeben. Als Typ-spezifische Parameter müssen die Matrix-Größen jeder Dimension angegeben werden, also z.B. für eine 3-dimensionale Matrix:

```
Type = schmitzm.data.property.MatrixProperty; 5; 4; 3
```

Die Matrix-Werte werden dann blockweise durch `Value`-Zeilen der Länge der ersten Dimension angegeben. Also im obigen Beispiel:

```
Value = v111; v211; v311; v411; v511
Value = v121; v221; v321; v421; v521
Value = v131; v231; v331; v431; v531
Value = v141; v241; v341; v441; v541

Value = v112; v212; v312; v412; v512
Value = v122; v222; v322; v422; v522
Value = v132; v232; v332; v432; v532
Value = v142; v242; v342; v442; v542

Value = v113; v213; v313; v413; v513
Value = v123; v223; v323; v423; v523
Value = v133; v233; v333; v433; v533
Value = v143; v243; v343; v443; v543
```

Entsprechend werden 4- oder 5-dimensionale Matrizen angegeben. Im folgenden ist eine exemplarische Beispieldatei für ein dynamisches XULU-Objekt aufgeführt, das den Namen „ClueModelParameter“ und 7 Propertys (drei Skalare, zwei Listen und eine 6×6-Matrix) erhält.

```
#===================================
# Name des Xulu-Objekts
#===================================
XuluObjectName = ClueModelParameter

#------------------
# Simulierte Jahre
#------------------
[Step Count]
Value = 10

#-------------------------------------------------------------
# Maximale Durchschnitts-Abweichung der simulierten LU vom Bedarf
#-------------------------------------------------------------
[Average Demand Tolerance]
Value = 50.0

#-------------------------------------------------------------------
# Maximale Abweichung der simulierten LU eines jeden LU-Typs vom Bedarf
#-------------------------------------------------------------------
[Single Demand Tolerance]
Value = 75.0

#---------------------
# Simulierte LU-Typen
#---------------------
[LU Types]
Value = Siedlung; intensive Landwirtschaft; weniger intensive Landwirtschaft; dich...

#---------------------------------------
# Nummern der dynamischen Driving Forces
#---------------------------------------
[Dynamic Driving Forces]
Value = 0; 1

#-----------------------------------------------------
# LU Conversion Matrix
# Veraenderung von Typ in Zeile r nach Typ in Spalte c:
#      0   -> Wechsel niemals moeglich
#      1   -> Wechsel immer moeglich
#    100+X -> Wechsel erst moeglich wenn Zelle bereits
#             X Zeitschritten fuer r genutzt wurde
#   -100-X -> Wechsel NICHT mehr moeglich wenn Zelle
#             bereits X Zeitschritten fuer r genutzt wurde
#-----------------------------------------------------
[LU Conversion Matrix]
Type  = schmitzm.data.property.MatrixProperty; 6; 6
#       Siedl.   iLW   wiLW  diWald  diSav   Rest
Value =     1;    0;     0;      0;     0;     0;
Value =     1;    1;   115;      0;     0;     0;
Value =     1;  110;     1;      0;   110;   120;
Value =     0;    0;     1;      1;     1;     0;
Value =     0;    0;   115;      1;     1;     0;
Value =     0;    1;     1;      1;     1;     1;

#-----------------------------------------------
# LU Conversion Elasticity fuer jeden LU-Typ u
#-----------------------------------------------
[Conversion Elasticity]
DataType = java.lang.Double
#       Siedl.   iLW  wiLW  diWald  diSav  Rest
Value =     1;  0.5;  0.2;      1;   0.2;   0.2
```

Anhang E

CLUE-Datensatz für das IMPETUS-Gebiet

Der komplette CLUE-Datensatz, den die RSRG für die Simulation von 25 Jahren des IMPETUS-Untersuchungsgebiets verwendet, besteht aus insgesamt 69 Eingabe-Dateien:

Datei(en)	Bedeutung
`main.1`	Steuerungsparameter für CLUE
`demand.in7`	LUC-Bedarfe für die 25 simulierten Jahre
`cov_all.0`	Initiale LUC-Konfiguration
`age.0`	Vergangenheitsinformationen über die initiale LUC-Konfiguration
`allow.txt`	Land-Use-Conversion-Matrix
`alloc1.reg`	Regressionsparameter der Driving-Force-Statistik
`alloc2.reg`	Regressionsparameter der Nachbarschafts-Statistik
`neighmat.txt`	Konfiguration der Nachbarschafts-Statistik
`region1.fil`	Definition von nicht zu berücksichtigenden Bereichen (Schutzgebiete)
`sc1gr0.fil` bis `sc1gr7.fil`	Ausprägung der statischen Driving Force 0 bis 7 `sc1gr0.fil` und `sc1gr1.fil` werden nicht verwendet, wenn dynamische Driving Forces verwendet werden.
`sc1gr0.0` bis `sc1gr0.25`	Ausprägung des dynamischen Driving Force 0 (*Bevölkerung*) für jedes Jahr
`sc1gr1.0` bis `sc1gr1.25`	Ausprägung des dynamischen Driving Force 1 (*Distanz zur Strasse*) für jedes Jahr

Für die Verwendung in XULU wurden die Inhalte der Dateien `main.1`, `demand.1`, `allow.txt`, `alloc1.reg`, `alloc2.reg` und `neighmat.txt` umformatiert und in einer zentralen Datei `ClueModelParameter.dxo` hinterlegt[1]. Die genannten 6 Dateien werden somit für XULU überflüssig.

[1] Die Dateierweiterung `dxo` steht als Abkürzung für „**D**ynamic **X**ulu **O**bject"

Anhang F

Test der JAVA-Polymorphie

Folgendes JAVA-Programm prüft die Performanz der Verwendung überladener Methoden-Aufrufe.

```
import java.awt.Dimension;
import java.awt.geom.Point2D;

public class PerformanceTest {
    private static final long TIMES = 100000000;

    public static void main(String[] argv) {
        long start;
        long end;

        //////////////// TEST A //////////////////
        // Point direkt erzeugen
        start = System.currentTimeMillis();
        for (int i=0; i<TIMES; i++) { Point2D newP = createPoint2D(10.0,200.0); }
        end = System.currentTimeMillis();
        System.out.println((end-start)+"ms bei direkter Objekterzeugung");

        //////////////// TEST B //////////////////
        // Point ueber Polymorphie erzeugen
        Dimension d = new Dimension(10,200);
        start = System.currentTimeMillis();
        for (int i=0; i<TIMES; i++) { Point2D newP = createPoint2D(d); }
        end = System.currentTimeMillis();
        System.out.println((end-start)+"ms bei Polymorphie");

        //////////////// TEST C //////////////////
        // Point ueber Zwischen-Objekt und Polymorphie erzeugen
        start = System.currentTimeMillis();
        for (int i=0; i<TIMES; i++) { Point2D newP = createPoint2D(new Dimension(10,200)); }
        end = System.currentTimeMillis();
        System.out.println((end-start)+"ms bei Polymorphie mit temp. Zwischen-Objekten");

    }
    // Polymorphie Ebene 1
    private static Point2D createPoint2D(Dimension d) {
        return createPoint2D(d.getWidth(),d.getHeight());
    }
    // Polymorphie Ebene 2
    private static Point2D createPoint2D(int x, int y) {
        return createPoint2D((double)x,(double)y);
    }
    // Eigentliche Objekt-Erzeugung
    private static Point2D createPoint2D(double x, double y) {
        return new Point2D.Double(x,y);
    }
}
```

Um einen realistischen Vergleich vorzunehmen, wurde das Programm *nicht* als Gesamtes ausgeführt. Statt dessen wurden für einen Teil-Tests (A, B oder C) die jeweils anderen beiden Test-Routinen auskommentiert. Es kam zu folgendem Ergebnis[1]:

A: 4188ms bei direkter Objekt-Erzeugung

B: 4375ms bei Polymorphie

C: 6719ms bei Polymorphie mit temporären Zwischen-Objekten

Es zeigt sich, dass die zusätzlichen Methodenaufrufe der Polymorphie (B) und insbesondere die Verwendung temporärer Zwischen-Objekte (C) merklichen Einfluss auf die Performanz haben.

Bemerkung:
Das obige Programm als Gesamtes führt zu folgendem Messergebnis:

A: 4156ms bei direkter Objekt-Erzeugung

B: 5750ms bei Polymorphie

C: 24391ms bei Polymorphie mit temporären Zwischen-Objekten

Der sehr viel höhere Wert für Test C lässt sich damit begründen, dass zwischenzeitlich die JAVA *Garbage Collection* aktiv wird, um die hohe Anzahl nicht mehr referenzierter Objekte aus dem Speicher zu entfernen.
Für einen Vergleich der Routinen A, B und C untereinander ist die Gesamt-Ausführung des Test-Programms also nicht geeignet!

[1] Auf einem TARGA TRAVELLER Notebook mit MOBILE AMD ATHLON 64 PROCESSOR (1.99GHz)

Anhang G

SDL – Specification and Description Language

SDL dient dazu die Kommunikation zwischen Prozessen zu beschreiben. Auch wenn in dieser Arbeit keine vollständig unabhängigen System-Prozesse miteinander kommunizieren, ist SDL gut geeignet, um die Abläufe zwischen den einzelnen XULU-Komponenten zu beschreiben. Folgende Abbildung zeigt, die in dieser Arbeit verwendeten SDL-Symbole und ihre Bedeutung. Das Grund-Prinzip eines SDL-Diagramms besteht darin, dass ein Prozess (beginnend im Start-Zustand) solange in einem Zustand verweilt, bis ein Eingabe-Signal (eines anderen Prozesses) eintrifft. In Abhängigkeit davon, welches Signal eintrifft, verzweigt der Prozess und führt solange „Aktionen“ aus, bis der nächstfolgende Zustand erreicht wird. Nähere Informationen zu SDL sind in [17, Kap. 2] und [28] zu finden.

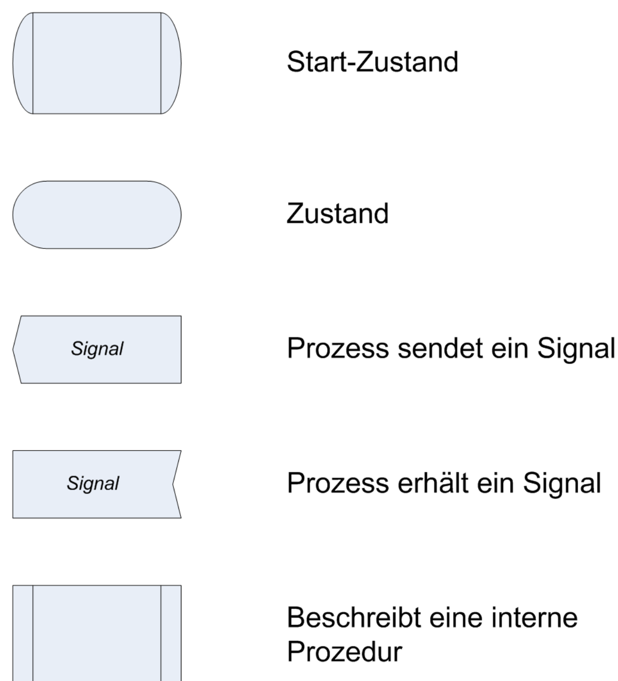


Abbildung G.1: Die in dieser Arbeit verwendeten SDL-Symbole.

Literaturverzeichnis

[1] Chetan Agarwal, Glen M. Green, J. Morgan Grove, Tom P. Evans und Charles M. Schweik. *A Review and Assessment of Land-Use Change Models: Dynamics of Space and Time, and Human Choice*. United States Department of Agricuture (USDA), 2002.

[2] *Benin – Länder- und Reiseinformationen des Auswärtigen Amts*.
http://www.auswaertiges-amt.de.

[3] Jana Borgwardt. *Diplomarbeit: Modellierung der Landnutzungsänderung im Upper Ouémé Catchment, Benin mit Hilfe von zellularen Automaten auf der Grundlage von multitemporalen Landsat-Satellitendaten*. 2004.
http://www.rsrg.uni-bonn.de/RSRGwww/Deutsch/Diplomarbeiten/Diplomarbeiten.html.

[4] Bernd Brügge und Allen H. Dutoit. *Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java*. Pearson, 2004.

[5] Helen Briassoulis. *Analysis of Land Use Change: Theoretical and Modeling*. National Research Council, 2000.
http://www.rri.wvu.edu/WebBook/Briassoulis/contents.htm.

[6] CLUE *„The Conversion of Land Use and its Effects"*.
http://www.dow.wau.nl/clue/.

[7] Univ.-Prof. Dr. Armin B. Cremers, Dr. Günter Kniesel und Daniel Speicher. Vorlesung „Softwaretechnologie", 2004.
http://www.informatik.uni-bonn.de/III/lehre/vorlesungen/SWT/SS2004/software.html.

[8] ESRI ARCVIEW – *GIS and Mapping Software*.
http://www.esri.com/.

[9] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addisson-Wesley, 1998.

[10] GEOTOOLS-*Projekt – Ein open source java GIS toolkit*.
http://www.geotools.org/ (Version 2)
http://lite.geotools.org/ (Version 1).

[11] Dale Green. *Java – The Reflection API*.
`http://java.sun.com/docs/books/tutorial/reflect/`.

[12] IMPETUS – *Integratives Management-Projekt für einen Effizienten und Tragfähigen Umgang mit Süßwasser in Westafrika*.
`http://www.impetus.uni-koeln.de/` (offizielle Homepage)
`http://www.rsrg.uni-bonn.de/RSRGwww/Deutsch/Forschung/IMPETUS/Impetus.html` (Hompage der RSRG).

[13] Alfons Kemper und Andre Eickler. *Datenbanksysteme*. Oldenbourg, 2004.

[14] Tim Lindholm und Frank Yellin. *The Java Virtual Machine Specification (Second Edition)*.
`http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

[15] Diana Livermann, Emilio F. Moran, Ronald R. Rindfuss und Paul C. Stern. *People and Pixels - Linking Remote Sensing and Social Science*. National Research Council, 2000.
`http://www.nap.edu/books/0309064082/html/index.html`.

[16] Prof. Dr. Rainer Manthey. Vorlesung „Informationssysteme“, 2002.
`http://www.informatik.uni-bonn.de/III/lehre/vorlesungen/Informationssysteme/WS02/folien.html`.

[17] Prof. Dr. Peter Martini. Vorlesung „Rechnernetze I“, 2001.
`http://web.informatik.uni-bonn.de/IV/martini/Lehre/Veranstaltungen/WS0102/index.html`.

[18] John O’Conner. *Java Internationalization: An Overview*.
`http://java.sun.com/developer/technicalArticles/Intl/IntlIntro/`.

[19] John O’Conner. *Java Internationalization: Localization with ResourceBundles*.
`http://java.sun.com/developer/technicalArticles/Intl/ResourceBundles/index.html`.

[20] Dawn C. Parker, Thomas Berger und Steven M. Manson. *Agent-Based Models of Land-Use and Land-Cover Change*. Report and Review of an International Workshop, Irvine/California, USA, 2001.
`http://www.indiana.edu/%7eact/focus1/abm_report6.pdf`.

[21] Dawn C. Parker, Thomas Berger und Steven M. Manson. *Meeting the Challenge of Complexity*. Proceedings of a Special Workshop on Land-Use/Land-Cover Change, Irvine/California, USA, 2001.
`http://www.csiss.org/events/other/agent-based/additional/proceedings.pdf`.

[22] Leopold Riedl. MAPMODELS – *User Manual*.
http://srf.tuwien.ac.at/MapModels/MapModDocu.zip.

[23] Leopold Riedl. *POSSIBLE CITIES - Simulation von Siedlungsentwicklung mit zellularen Automaten*. 1999.
http://www.srf.tuwien.ac.at/mapmodels/pdf/riedl_possiblecities_corp99.pdf.

[24] Leopold Riedl und Robert Kalasek. *MapModels - Programmieren mit Datenfluß-graphen*. 1998.
http://www.srf.tuwien.ac.at/mapmodels/agit98/paper.htm.

[25] Leopold Riedl und Robert Kalasek. *Hierarchisches Modellieren mit MapModels*. 2002.
http://www.srf.tuwien.ac.at/mapmodels/pdf/riedlkalasek_agit02_mapmodels_hierarchisch.pdf.

[26] Leopold Riedl, Harald Vacik und Robert Kalasek. *MapModels: a new approach for spatial decision support in silvicultural decision making*. 2000.
http://www.srf.tuwien.ac.at/mapmodels/pdf/riedlvacikkalasek_2000_mapmodels_silviculture.pdf.

[27] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[28] SDL-Forum.
http://www.sdl-forum.org/.

[29] SPSS – *Ein Programm zur statistischen Analyse*.
http://www.spss.com.

[30] Dr. Hans-Peter Thamm. Storyline: Landnutzungsänderung in Benin, 2004.

[31] Dr. Hans-Peter Thamm, Michael Judex und Jana Borgwardt. Persönliche Gespräche und Erfahrungsberichte der RSRG.

[32] Hans-Peter Thamm, Michael Judex und G. Menz. *Modelling of Land-Use and Land-Cover Change (LUCC) in Western Africa using Remote Sensing*. 2005.

[33] C. D. Tomlin. *Geographic Information Systems and Cartographic Modelling*. Prentice-Hall, Englewood Cliff, New Jersey, 1990.

[34] Tanja Tötzer, Leopold Riedl und Klaus Steinnocher. *Räumliche Nachklassifikation von Landbedeckungsdaten mit MapModels*. 2000.
http://www.srf.tuwien.ac.at/mapmodels/pdf/toetzerriedlsteinnocher_corp2000_mapmodels_landbedeckung.pdf.

[35] A. Veldkamp, Peter H. Verburg, Paul Schott und Martin Dijst. *Land use change modelling: current practice and research priorities*. GeoJournal, 2002.

[36] Peter H. Verburg und A. Veldkamp. *Methodology for simulating the spatial dynamics of land use change in forest fringes: The CLUE-S Model*. Laboratory of Soil Science and Geology, Wageningen University, The Netherlands, 2002. `http://www.gwdg.de/~symp2002/pdf/100_methodology_veldkamp_final.pdf`.

[37] Peter H. Verburg, A. Veldkamp und Victoria Espaldon. *Modeling the Spatial Dynamics of Regional Land Use: The CLUE-S Model*. Springer-Verlag New York, 2002.

[38] Peter H. Verburg, Tom Veldkamp, , Koen Overmars, Jan Peter Lesschen und Kasper Kok. *Manual for the* CLUE-*S model*. `http://www.dow.wau.nl/clue/clue_manual.pdf`.

[39] Peter H. Verburg, Tom Veldkamp und Jan Peter Lesschen. *Exercises for the* CLUE-*S model*. `http://www.dow.wau.nl/clue/Exercises.pdf`.

[40] Michael F. Worboys. *GIS - A Computing Perspective*. Taylor & Francis, 1995.

[41] *JavaDoc zur Klassenbibliothek der* XULU-*Modelling-Plattform.*