Fachpraktikum
Erweiterbare Datenbanksysteme
Wintersemester 2012/13
Aufgabe der Phase 2

„Rasterdaten“

Ralf Hartmut Güting, Thomas Behr,
Simone Jandt, Fabio Valdés

15.11.2012

Lehrgebiet Datenbanksysteme für neue Anwendungen
Fakultät für Mathematik und Informatik

FernUniversität in Hagen

1 Motivation

Wie Sie bereits wissen, unterstützt SECONDO neben den klassischen Datentypen *int*, *real*, *bool*, *string* usw. auch deren zeitabhängige Versionen (z.B. moving int oder kurz *mint*). Solch ein Datentyp repräsentiert eine Funktion von der Zeit in den jeweiligen Basisdatentyp. Nun liegt es nahe, dass Werte nicht nur zeitlich sondern auch räumlich variieren. Dies bedeutet, dass ein solches Objekt an verschiedenen Positionen im Raum unterschiedliche Werte annehmen kann - beispielsweise hängt die Höhe der Erdoberfläche bezüglich des Meeresspiegels von geographischen Koordinaten ab. Andererseits können die Objektwerte sowohl raum- als auch zeitabhängig sein, z.B. die Temperaturverteilung in Alaska im Tagesverlauf.

Bislang werden solche Datentypen von SECONDO nicht unterstützt. Dies sollen Sie durch die Bearbeitung der folgenden Aufgabe ändern.

2 Datentypen

2.1 Datentypen für räumliche Abhängigkeit

Im Folgenden seien $T, U, V \in \{\textit{bool}, \textit{int}, \textit{real}, \textit{string}\}$ sowie $x, y \in \{\textit{s}, \textit{ms}\}$.

Zur Darstellung räumlich abhängiger Objekte in SECONDO verwenden wir ein gleichmäßiges quadratisches Raster, das durch den Datentyp *grid2* definiert wird. Für die Definition eines solchen Gitters wird mindestens dessen Ursprung sowie die Kantenlänge einer Zelle benötigt. Somit besteht der Datentyp *grid2* aus insgesamt drei reellwertigen Komponenten, von denen die ersten beiden die Koordinaten des Ursprungs und die dritte die Kantenlänge repräsentieren. Um jeden Raumpunkt eindeutig einer Gitterzelle zuordnen zu können, legen wir fest, dass nur der linke sowie der untere Rand zur jeweiligen Zelle gehören. Das Gitter selbst kann in sämtlichen Richtungen um den Ursprung beliebige Ausmaße annehmen.

Um nun räumlich abhängige Objekte zu implementieren, wird jeder Gitterzelle ein konstanter Wert eines Datentyps T zugewiesen. Der Wert einer Zelle kann auch undefiniert sein. Somit sind die vier räumlich abhängigen Datentypen *sint*, *sreal*, *sbool* oder *sstring* zu implementieren.

Für eine effiziente Implementierung sind Teilgitter in SMI-Keyed-files (mit Records konstanter Größe) abzulegen. Die Anzahl der Zellen wird dabei durch die Seitengröße des Systems bestimmt, genauer gesagt, soll jedes Teilgitter eine Seite belegen. So lassen sich z.B. bei einer Seitengröße von 4 KB (abzüglich 8 Byte für den Zellenschlüssel) 1022 Integer der Länge 32 bit in einem Teilgitter unterbringen, so dass insgesamt ein 31×31-Teilgitter pro Seite gespeichert werden kann, wenn wir den Wert `NaN` als undefinierten Wert betrachten. Es kann Objekte geben, bei denen nur wenige Teilgitter definierte Werte haben. Teilgitter, die ausschließlich undefinierte Werte enthalten, sollen nicht gespeichert werden. Um existierende Teilgitter schnell finden zu können, soll zusätzlich zum SMI-Keyed-file ein R-Baum zur Indexierung vorhandener Teilgitter verwendet werden. Dies dient insbesondere der Beschleunigung einiger Operationen (siehe unten).

Eine Besonderheit stellt der Typkonstruktor *sstring* dar. Erstens benötigen die zu einer Zelle gehörenden Werte im Gegensatz zu *int* und *real* viel Speicherplatz, und zweitens werden in den meisten Fällen nur wenige unterschiedliche Werte verwendet. Daher sind hier in den Gitterzellen nicht die *string*-Werte selbst zu speichern sondern Verweise in eine separate Struktur, welche die verschiedenen Ausprägungen der *string*s enthält.

2.2 Datentypen für räumliche und zeitliche Abhängigkeit

Wie bei den nur räumlich abhängigen Datentypen wird ein gleichmäßiges Gitter verwendet. Allerdings benötigen wir in diesem Fall eine weitere Gitterdimension für die Zeitachse, die ebenfalls gleichmäßig partitioniert wird, deren Größe aber unabhängig von der räumlichen Zellgröße ist. Die räumliche Zuordnung eines Punktes auf einem Zellenrand erfolgt wie bei *grid2*, während in der zeitlichen Dimension das gesamte Zeitintervall abgesehen vom spätesten Zeitpunkt der Zelle zuzuordnen ist. Ein solches Gitter wird durch den Datentyp *grid3* definiert, der zusätzlich zu *grid2* einen *duration*-Wert enthält, der die Dauer des zu einer Zelle gehörenden Zeitintervalls beschreibt. Wie bereits bei den räumlich abhängigen Datentypen ist der Inhalt jeder Zelle konstant.

Die Implementierung der räumlich und zeitlich abhängigen Datentypen (*msint*, *msreal*, *msbool* und *msstring*) erfolgt analog zu den nur räumlich abhängigen Datentypen. Auch hier erfährt der letztgenannte Datentyp die o.g. Spezialbehandlung.

Zusätzlich werden die Datentypen *isint*, *isreal*, *isbool* und *isstring* benötigt, die jeweils durch einen Zeitpunkt (*instant*) und einen Wert des entsprechenden räumlich abhängigen Datentyps definiert sind.

3 Operatoren

3.1 Import von Rasterdaten

Da Rasterdaten aus verschiedenen Internetquellen bezogen werden können, ist die Implementierung von Importoperatoren sinnvoll und notwendig. Für die Lösung der Aufgabe sind mindestens die folgenden Formate zu importieren:

1. HGT; Bei diesem Format enthält der Dateiname die Information über die geographischen Koordinaten der südlichen und westlichen Begrenzung des abgedeckten Gitters. In der Datei selbst stehen lediglich 16 Bit vorzeichenbehaftete Integer, die ein Gitter der Größe 1201x1201 oder 3601x3601 beschreiben. Die Dateigröße bestimmt dabei die Gittergröße.[1] Sollten beim Importieren fehlerhafte oder nicht vorhandene Dateien auftreten, sind diese zu ignorieren.
2. ESRI grid; [2]
3. ESRI ASCII raster format; [3]

Die Importoperatoren sollen jeweils einen kompletten Strom von Dateinamen (*text*) in ein entsprechendes räumlich abhängiges Objekt umwandeln. Passt ein in einer Datei definiertes Gitter nicht zu den bereits bearbeiteten Datensätzen, oder sollen bereits belegte Zellen überschrieben werden, so wird die entsprechende Datei ignoriert.

3.2 Umwandlung von räumlichen Objekten in *sbool* und zurück

In der `SpatialAlgebra` sind u.a. die Datentypen *line* und *region* definiert. Offensichtlich besteht die Möglichkeit diese als *sbool* darzustellen. Die entsprechende Typumwandlung sollen die Operatoren **fromline** bzw. **fromregion** realisieren. Als Argumente erhalten diese das umzuwandelnde räumliche Objekt sowie eine Gitterdefinition vom Typ *grid2*. Beim Operator **fromline** sind genau diejenigen Zellen mit `true` zu belegen, die von der Linie geschnitten werden. Bei der Umwandlung einer *region* durch **fromregion** werden genau diejenigen Zellen mit `true` belegt, deren Mittelpunkt innerhalb des Gebiets liegt. In allen Teilgittern, bei denen mindestens eine Zelle auf `true` gesetzt wird, werden die verbleibenden Zellen mit `false` markiert; die restlichen Teilgitter bleiben undefiniert und werden somit nicht gespeichert.

Die entgegengesetzte Umwandlung von *sbool* funktioniert ausschließlich nach *region* und benötigt außer dem Objekt keine weiteren Argumente. Der zu implementierende Operator heißt **toregion** und basiert auf einer Zyklensuche innerhalb des *sbool*-Objekts.

3.3 Hinzufügen zeitlicher Abhängigkeit

Durch den Operator **s2ms** lässt sich einem rein räumlichen Rasterobjekt eine zeitliche Komponente hinzufügen. Genauer gesagt, werden dem Rasterobjekt das für den Typ *grid3* notwendige *duration*-Objekt sowie ein Zeitintervall (gegeben durch einen Start- und einen Endzeitpunkt, jeweils vom Typ *instant*) übergeben. Jede Zelle des Ergebnisobjekts erhält den Wert der ursprünglichen Zelle, falls wenigstens die Hälfte ihres Zeitintervalls im Argumentintervall liegt, anderenfalls ist sie undefiniert.

[1] Weitere Informationen zum Format sowie Beispieldaten finden Sie u.a. unter: `http://dds.cr.usgs.gov/srtm/version2_1/`

[2] `http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#//009t0000000w000000`

[3] `http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#/ESRI_ASCII_raster_format/009t0000000z000000/`; Daten für dieses und das vorherige Format finden Sie unter `http://www.worldclim.org/current` oder `https://werdis.dwd.de/werdis/toSimpleSearchGetInstances.do;jsessionid=193D01515C2AD4B4630DF42374851DEE`

3.4 Einschränkung auf Raum bzw. Zeit

Um herauszufinden, welcher Wert an einer bestimmten Position vorliegt, verwendet man den Operator **atlocation**. Dieser ist kompatibel zu allen erwähnten räumlich und räumlich-zeitlich abhängigen Datentypen und erhält als zweites Argument ein Objekt vom Typ *point*. Beachten Sie, dass das Ergebnis bei räumlich-zeitlich abhängigen Objekten ein bewegtes Objekt (*mint*, *mreal*, *mbool* oder *mstring*) ist.

Ein Rasterobjekt lässt sich nicht nur auf einen Punkt sondern auch auf ein Rechteck beschränken. Im Unterschied zu **atlocation** ist das resultierende Objekt weiterhin räumlich (bzw. räumlich-zeitlich) abhängig. Der Operator **atrange** erhält ein Rasterobjekt und ein Rechteck und schränkt das Gitter auf diejenigen Zellen ein, die das übergebene Rechteck schneiden. Im Fall eines räumlich-zeitlich abhängigen Objekts kann der Operator optional zwei Zeitpunkte (*instant*) verarbeiten, wodurch zusätzlich eine zeitliche Einschränkung erreicht wird.

Möchte man ein zeitabhängiges Rasterobjekt lediglich auf eine bestimmte Menge von Zeitintervallen beschränken, so ist der Operator **atperiods** zu verwenden. Dieser erhält als zweites Argument ein Objekt vom Typ *periods* und liefert als Ergebnis wieder ein zeitabhängiges Rasterobjekt, das nur diejenigen Zellen enthält, die die Zeitintervalle schneiden.

Das zugrundeliegende Gitter des ursprünglichen Objekts bleibt bei Ausführung der Operatoren **atrange** und **atperiods** unverändert.

Beschränkt man ein zeitlich abhängiges Rasterobjekt durch den Operator **atinstant** auf einen einzelnen Zeitpunkt, so erhält man den Ergebnistyp *isT*. Für den Zugriff auf eine der beiden Komponenten (Zeitpunkt oder Rasterobjekt) eines solchen Ergebnisses wird der Operator **inst** bzw. **val** verwendet.

3.5 Ermittlung von Rasterinformationen

Die einem Rasterobjekt zugrundeliegende Gitterdefinition ist durch den Operator **getgrid** abrufbar, der ein Objekt vom Typ *grid2* oder *grid3* zurückgibt. Letzteres ist bei zeitlich abhängigen Objekten der Fall.

Diejenigen Zeitintervalle, während derer ein zeitlich abhängiges Rasterobjekt mindestens einen definierten Zellwert besitzt, werden vom Operator **deftime** als Objekt vom Typ *periods* berechnet.

Der Operator **bbox** bestimmt das minimale begrenzende Rechteck (MBR) für die räumliche Ausdehnung aller definierten Gitterzellen. Der Rückgabetyp ist *rect*. Für eine effiziente Ausführung sollte das MBR Teil der jeweiligen Datenstruktur sein.

3.6 Aggregation

Die Operatoren **minimum** und **maximum** errechnen jeweils den kleinsten bzw. größten definierten Wert innerhalb der Menge aller Zellen eines Rasterobjekts von einem der Datentypen *sT* oder *msT*. Zur effizienten Unterstützung dieser Operatoren sollten diese Werte in den Datenstrukturen der Rasterobjekte vorliegen.

3.7 Compose-Operation

Die Werte eines rein räumlichen Rasterobjekts entlang einer als *mpoint* gegebenen Bewegung ermittelt der Operator **compose**. Für ein Rasterobjekt vom Typ *sT* besitzt das Ergebnis den Typ *mT*.

3.8 Manipulation des Rasterinhalts

Der Operator **map** wendet eine gegebene Funktion, die den Wert des Zelleninhalts auf einen neuen Wert abbildet, auf jede Gitterzelle mit definiertem Inhalt an. Die Funktion muss dabei nicht typerhaltend sein sondern kann beliebig zwischen *int*, *real*, *bool* und *string* abbilden. Um einfache Notationen innerhalb von SECONDO-Anfragen, die diesen Operator benutzen, zu ermöglichen, wird der Typemapping-Operator **CELL1** benötigt. Bei Typemapping-Operatoren ist die Valuemapping-Funktion auf 0 gesetzt, als Argumenttypen erhalten sie die Argumente der sie umgebenden Funktion. Dem Operator **CELL1** werden ein Rastertyp *xT* sowie weitere Argumente übergeben. Er bildet seine Argumente auf T ab.

3.9 Verschmelzung zweier Raster

Zwei Raster mit verträglicher Gitterdefinition lassen sich durch den Operator **map2** mit Hilfe einer Funktion zu einem neuen Raster verschmelzen. Dabei nennen wir zwei Gitterdefinitionen verträglich, wenn die Gitterzellen räumlich übereinstimmen und, im Fall zweier dreidimensionaler Gitter, auch die Zeitkomponenten übereinstimmen. Beachten Sie, dass die Ursprünge beider Gitter nicht identisch sein müssen. Die erwähnte Funktion berechnet aus den zwei Werten der zusammengehörigen Zellen der beiden Raster den Zellenwert für die resultierende Rasterzelle und wird für solche Paare von Zellwerten aufgerufen, bei denen mindestens einer der beiden Eingabewerte definiert ist. Ähnlich wie bei **map** können sowohl die Eingabetypen als auch der Ergebnistyp beliebig aus *int*, *real*, *bool* und *string* und damit insbesondere verschieden sein. Auch hier soll eine benutzerfreundliche Eingabe der Parameterfunktion möglich sein. Das erste Argument können wir bereits mit dem Typemapping-Operator **CELL1** behandeln. Um auch den zweiten Argumenttyp abbilden zu können, ist der zusätzliche Typemapping-Operator **CELL2** zu implementieren. Dieser prüft, ob sein zweites Argument vom Typ xT ist und liefert T zurück.

Da Rasterobjekte aus verschiedenen Quellen nicht immer miteinander verträgliche Gitterdefinitionen besitzen, bietet der Operator **matchgrid** die Möglichkeit, die Gitterdefinition eines Rasterobjekts zu manipulieren. Hierzu erhält er neben dem Objekt o die neue Gitterdefinition g (*grid2* oder *grid3*), eine Funktion, die eine Relation $rel(tuple([Elem : T]))$ zu einem Wert vom Typ U aggregiert, sowie einen booleschen Wert, der angibt, ob die Größe der Zellenüberlappung als Gewichtung bei der Aggregation verwendet wird. Das letztgenannte Argument ist nur für $T \in \{int, real\}$ relevant und wird anderenfalls ignoriert. Die erwähnte Relation enthält dabei alle (evtl. gewichteten) Werte, die sich in denjenigen Zellen aus o befinden, die durch die aus g erzeugte Zelle geschnitten werden.

Für eine benutzerfreundliche Verwendung dieses Operators ist daher noch der Typemapping-Operator **CELLS** zu implementieren. Der **CELLS**-Operator liefert als Ergebnistyp die entsprechende Relation (s.o.).

4 Visualisierung

Natürlich möchte man sich die Ergebnisse der Arbeit auch komfortabel darstellen lassen. Erweitern Sie hierzu den Höse-Viewer um Displayklassen für die verschiedenen Datentypen.

5 Zusammenfassung

Als Hilfestellung erhalten Sie nachfolgend eine Übersicht aller zu implementierenden Datentypen, Displayklassen und Operatoren.

5.1 Zu implementierende Datentypen

sint	*sreal*	*sbool*	*sstring*
msint	*msreal*	*msbool*	*msstring*
isint	*isreal*	*isbool*	*isstring*
grid2	*grid3*		

5.2 Zu implementierende Displayklassen

Dsplsint	Dsplsreal	Dsplsbool	Dsplsstring
Dsplmsint	Dsplmsreal	Dsplmsbool	Dsplmsstring
Dsplisint	Dsplisreal	Dsplisbool	Dsplisstring
Dsplgrid2	Dsplgrid3		

5.3 Zu implementierende Operatoren

atlocation	: $\underline{sT} \times \underline{point} \to T$
	: $\underline{msT} \times \underline{point} \to \underline{mT}$
atinstant	: $\underline{msT} \times \underline{instant} \to \underline{isT}$
inst	: $\underline{isT} \to \underline{instant}$
val	: $\underline{isT} \to \underline{sT}$
atperiods	: $\underline{msT} \times \underline{periods} \to \underline{msT}$
atrange	: $\underline{xT} \times \underline{rect} \to \underline{xT}$
	: $\underline{msT} \times \underline{rect} \times \underline{instant} \times \underline{instant} \to \underline{msT}$
deftime	: $\underline{msT} \to \underline{periods}$
bbox	: $\underline{xT} \to \underline{rect}$
minimum	: $\underline{xT} \to T$
maximum	: $\underline{xT} \to T$
map	: $\underline{xT} \times (T \to U) \to \underline{xU}$
map2	: $\underline{sT} \times \underline{sU} \times (T \times U \to V) \to \underline{sV}$
	: $\underline{msT} \times \underline{sU} \times (T \times U \to V) \to \underline{msV}$
	: $\underline{sT} \times \underline{msU} \times (T \times U \to V) \to \underline{msV}$
	: $\underline{msT} \times \underline{msU} \times (T \times U \to V) \to \underline{msV}$
fromregion	: $\underline{region} \times \underline{grid} \to \underline{sbool}$
toregion	: $\underline{sbool} \to \underline{region}$
s2ms	: $\underline{sT} \times \underline{duration} \times \underline{instant} \times \underline{instant} \to \underline{msT}$
compose	: $\underline{mpoint} \times \underline{sT} \to \underline{mT}$
matchgrid	: $\underline{sT} \times \underline{grid2} \times (\underline{rel}(\underline{tuple}([Elem : T])) \to U) \times \underline{bool} \to \underline{sU}$
	: $\underline{msT} \times \underline{grid3} \times (\underline{rel}(\underline{tuple}([Elem : T])) \to U) \times \underline{bool} \to \underline{msU}$
getgrid	: $\underline{sT} \to \underline{grid2}$
	: $\underline{msT} \to \underline{grid3}$
importHgt	: $\underline{stream}(\underline{text}) \to \underline{sint}$
importEsriGrid	: $\underline{stream}(\underline{text}) \to \underline{sT}$
importEsriRaster	: $\underline{stream}(\underline{text}) \to \underline{sT}$
CELL1	: $\underline{xT} \times \ldots \to T$
CELL2	: $\underline{xT} \times \underline{yU} \times \ldots \to U$
CELLS	: $\underline{xT} \times \ldots \to \underline{rel}(\underline{tuple}([Elem : T]))$

6 Beispielanfragen

Im Folgenden präsentieren wir zahlreiche SECONDO-Anfragen, die die Anwendung der neu zu implementierenden Operatoren und Datentypen veranschaulichen. Aufgrund der fehlenden Operatoren und Datentypen war es den Autoren leider nicht möglich, die nachstehenden Anfragen zu testen.

Dazu seien `N61W151.hgt` bis `N70W160.hgt` insgesamt 100 Dateien im HGT-Format, die sich im Verzeichnis `secondo/bin/height/` befinden, die die Höhe der Erdoberfläche über dem Meeresspiegel in einem Teil Alaskas beschreiben. Diese lassen sich durch das folgende Kommando in SECONDO importieren:

```
let elevation = getDirectory("height") filter [. endswith "hgt"] importHgt;
```

Analog importieren wir die Schneehöhen aus dem Verzeichnis `secondo/bin/snow/` in das Objekt *snow*.

Weiterhin gehen wir davon aus, dass die Bewegung des Rentiers *Klara* als $\underline{mpoint}$ in der Datenbank gespeichert ist. Sei schließlich *temperature* ein Objekt vom Typ $\underline{msreal}$, welches die Temperaturverteilung für Alaska innerhalb der letzten drei Wochen beschreibt.

Die Schneehöhe am Punkt $(65.0, -154.7)$ lässt sich wie folgt bestimmen:

```
query snow atlocation makepoint(65.0, -154.7);
```

Den Temperaturverlauf der letzten drei Wochen für den gleichen Punkt erhalten wir durch:

```
query temperature atlocation makepoint(65.0, -154.7);
```

Die Temperaturverteilung am 08.11.2012 um 09:56 Uhr in Alaska bestimmen wir folgendermaßen:

```
query val(temperature atinstant [const instant value "2012-11-08-09:56"]);
```

Um die Menge der Zeitintervalle zu erhalten, während derer der Temperaturverlauf definiert ist, verwenden wir die folgende Anfrage:

```
query deftime(temperature);
```

Will man aus speicherplatztechnischen Gründen nur noch die Temperaturverteilung am 08.11.2012 in der Datenbank behalten, realisiert man dies durch:

```
update temperature := temperature atperiods
                      [const periods value (("2012-11-08" "2012-11-09" TRUE FALSE))];
```

Aus ähnlichen Motiven soll nun die Schneehöhe auf ein kleineres Gebiet begrenzt werden:

```
update snow := snow atrange [const rect value (62 64 -155 -153)];
```

Das MBR des durch die Temperaturverteilung abgedeckten Gebiets bestimmen wir durch:

```
query bbox(temperature);
```

Die am 08.11.2012 minimale Temperatur ermitteln wir nun mit der Anfrage:

```
query minimum(temperature);
```

Die maximale Schneehöhe erhalten wir durch den Befehl:

```
query maximum(snow);
```

Um diejenige Region zu bestimmen, in der die Schneehöhe mehr als 3 m beträgt, verwenden wir das nachstehende Kommando:

```
query snow map[. > 3.0] toregion;
```

Da *snow* und *elevation* unterschiedliche Gitterauflösungen besitzen, können sie nur unter Zuhilfenahme des Operators **matchgrid** miteinander verschmolzen werden. Um die tatsächliche Höhe am Punkt $(65.0, -154.7)$ zu berechnen:

```
query map2(elevation, matchgrid(snow, getgrid(elevation),
           cells feed max[Elem], FALSE), fun(e:real, h:real) e+h)
           atlocation makepoint(65.0, -154.7);
```

Schließlich möchten wir mit Hilfe der Objekte *snow* und *Klara* bestimmen, wann *Klara* sich durch welche Schneehöhe kämpfen musste. Der Ergebnistyp ist also *mreal*. Wir erhalten das Ergebnis mit folgender Anfrage:

```
query Klara snow compose;
```

A Richtlinien

1. Der Programmcode muss so in PD kommentiert sein, dass sich SECONDO-kundige Fremde problemlos in den Code einarbeiten und denselben anpassen und erweitern können.

2. Natürlich sollen sie Ihren Code auch testen und zwar nicht nur mit kleinen Testbeispielen, sondern auch über den Testrunner mit großen Mengen von Testdaten bzw. großen Testdateien. Quellen hierzu haben wir Ihnen bereits angegeben. Dabei sollten auch die Tests verständlich kommentiert und dokumentiert sein.

A.1 Für die Teamarbeit der Phase 2

1. Die Systementwicklung sollte in mehrere Etappen aufgeteilt werden. Am Ende jeder Etappe (auch schon der ersten) sollte ein lauffähiges Teilsystem vorliegen. In Anbetracht der Gesamtzeit von etwa zwei Monaten sollten die Etappen eine Länge von 2-3 Wochen haben.

2. In der 2. Präsenzphase legt die Gruppe die Etappen und die Aufgabenverteilung fest. Aufgaben sollten möglichst so verteilt werden, dass jeder in jeder Etappe etwas beitragen kann. Die Etappen sind entsprechend zu planen.

3. Es soll ein Gruppensprecher bzw. eine Gruppensprecherin ausgewählt werden, der jeweils zu den Etappenendterminen der Betreuerin bzw. dem Betreuer über die Erreichung der Ziele berichtet. Das Teilsystem der Etappe n liegt jeweils auf dem CVS-Server vor und kann somit auch vom Betreuer bzw. von der Betreuerin getestet werden.

4. Das Team ist gemeinsam für die Lösung der gestellten Aufgabe verantwortlich. Sollten Mitglieder des Teams nicht angemessen mitarbeiten, sollte zunächst versucht werden, das Problem innerhalb des Teams zu lösen. Falls das nicht gelingt, ist die Betreuerin bzw. der Betreuer zu informieren.

5. Die letzte Etappe endet spätestens 2 Wochen vor der 3. Präsenzphase. Zu diesem Zeitpunkt sollte die Softwareentwicklung abgeschlossen sein. In der 2. Woche vor der Präsenzphase werden die Betreuer und Betreuerinnen das entwickelte System ausprobieren. Ggf. können dabei noch entdeckte Fehler korrigiert werden.

6. Die letzte Woche sollte von den Teilnehmern und Teilnehmerinnen dazu genutzt werden, die Abschlusspräsentation vorzubereiten. In der Abschlusspräsentation sollten einerseits die Konzepte der Implementierung erklärt werden, andererseits natürlich das System vorgeführt werden. Dafür steht je etwa eine Stunde zur Verfügung.

B Wichtige CVS-Kommandos

Achtung: Lässt man ein bei einem CVS-Kommando aufgeführtes [*Dateiname*(*n*)] weg, so bezieht sich das Kommando auf sämtliche Dateien des Dateisystemteilbaums, dessen Wurzel das aktuelle Verzeichnis darstellt.

B.1 Einloggen

Tragen Sie in die Datei `/.bashrc` folgende Zeile ein:

```
export CVSROOT=":pserver:<user>@agnesi.fernuni-hagen.de:2401/home/cvsroot2"
```

Dabei ist `<user>` durch den kleingeschriebenen Nachnamen zu ersetzen. Starten Sie dann eine neue Konsole und führen Sie das CVS-Kommando `cvs login` aus. Der erste Versuch schlägt fehl. Wiederholen Sie das Kommando – diesmal sollte es funktionieren.

B.2 Online-Hilfe

`cvs -help-commands` = Auflistung aller CVS-Kommandos
`cvs -H <command>` Hilfe zu bestimmten Kommando

B.3 Arbeitskopie erstellen und aktualisieren

cvs co secondo Holt Version vom Server.

cvs update [*Dateinamen***]** Aktualisiert Dateien unterhalb des aktuellen Verzeichnisses. Neue Verzeichnisse werden ignoriert.

cvs update -d [*Dateinamen***]** Auch neue Verzeichnisse werden geholt.

cvs update -Pd [*Dateinamen***]** Neue, nicht-leere Verzeichnisse werden berücksichtigt.

cvs update -PdA [*Dateinamen***]** Entfernt zusätzlich Sticky Tags.

cvs update -r<version> [*Dateinamen***]** Holt eine bestimmte Version vom Server; Setzt Sticky Tag.

B.4 Änderungen beobachten

cvs -n update [*Dateinamen***]** Ohne Angabe von Dateinamen werden alle Dateien des aktuellen Verzeichnisses und seiner Unterverzeichnisse angezeigt. Es bedeuten:

- ? Datei nicht durch cvs verwaltet
- M Datei wurde lokal verändert
- U Es gibt ein Update auf dem Server
- C Konflikte zwischen eigenen Änderungen und Serverupdate

cvs stat [*Dateinamen***]** zeigt Zustand der aktuellen Datei an (Versionsnummer, ...)

cvs log [*Dateinamen***]** Zeigt Versionshistorie an.

B.5 Eigene Änderungen auf Server bringen

cvs ci -m "Kommentar" [*Dateinamen***]** Bringt lokale Änderungen auf den Server.

cvs add *Dateinamen* Fügt eine Datei/Verzeichnis dem CVS hinzu. Für Dateien ist zusätzlich ein `cvs ci Dateiname` notwendig.

cvs delete *Dateinamen* Löscht eine Datei. Auf alte Versionen kann jedoch weiterhin zugegriffen werden.