Prof. Dr. Eckart Modrow

Didaktik der Informatik
Universität Göttingen
Goldschmidtstrasse 7
37077 Göttingen
emodrow@informatik.uni-goettingen.de

OOP mit BYOB

Abstract: Die auf Scratch basierende grafische Entwicklungsumgebung BYOB[1] geht in einigen Bereichen weit über die Möglichkeiten von sonst in der Schule benutzten Sprachen hinaus. Die Grundfunktionalität und die Eignung für die Oberstufeninformatik etwa im Bereich der Datenstrukturen wurden an anderer Stelle dargestellt[2]. In diesem Artikel geht es um die Möglichkeiten im Bereich der Objektorientierten Programmierung (OOP).

1. Die Kommunikation zwischen Objekten

Der in vielen, aus ehemals imperativen Sprachen entstandenen Systemen begangene Weg zur OOP führt von den Klassen zu den Objekten, also vom Abstrakten zum Konkreten. Er ist ein typisches Top-Down-Verfahren, setzt also zu Beginn der Arbeit eine recht konkrete Vorstellung vom zu erreichenden Endprodukt voraus – und überfordert damit den größten Teil der Lernenden im Anfangsunterricht. Da im Fach Informatik ein nennenswerter Teil der Schülerinnen und Schüler nur den Anfangsunterricht durchläuft, kommt es zu den bekannten Problemen. Lerntheoretisch ist der Weg mehr als problematisch und wird entsprechend kritisiert. Geht es aber anders?

Wenn man in BYOB von konkreten Objekten ausgeht, diese entwickelt, testet und modifiziert, kann man später entweder Kopien oder Klone dieser Objekte erzeugen und diese so vervielfältigen. Das ursprüngliche Objekt dient dann als Referenz, als Prototyp seiner Klasse. Das Vorgehen beruht auf dem alten Delegation-Modell von Henry Lieberman[3].

Sehen wir es uns einmal genauer an einem Beispiel an: Wir erzeugen als Objekt (als BYOB-Sprite) einen einfachen Datenspeicher mit einer lokalen Liste Inhalte, den wir durch eine Kommode repräsentieren. Dieses statten wir mit lokalen Zugriffsmethoden auf die Daten aus, indem wir die Methoden rein <daten> und raus implementieren. Wir erhalten eine simple Queue. So können wir zwar beliebige Inhalte in die Liste schreiben und daraus entfernen, aber ausreichend ist das nicht, weil ein Datenspeicher sich ja nicht selbst „befüllt“. Damit benötigen wir einen Zugriff auf die Methoden des Objekts von außen.

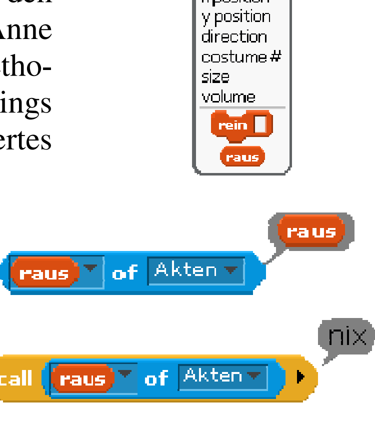
Um das zu demonstrieren, erzeugen wir zwei Kopien[4] unseres Speicher-Sprites, die wir Akten und Andenken nennen, sowie eine IT-Beauftragte Anne und erhalten die nebenstehende Konfiguration.

Wie kann Anne auf ihre Datenspeicher zugreifen?

---

[1] Build Your Own Blocks. Download unter http://byob.berkeley.edu/
[2] [MSM11], [Mo11]
[3] [Li86]
[4] Kopien enthalten schon anfangs ihre eigenen Attribute, hier die Liste. Erzeugten wir Klone, dann würden alle mit der gleichen Liste arbeiten, die wir erst überschreiben müssten, um unabhängige Speicher zu erhalten.

In BYOB sind die Scratch-Blöcke <attribute> of <sprite> und set <variable> to <value> stark modifiziert, und zwei neue, object und attribute, sind hinzu gekommen. Mit diesen können wir auf die Attribute und Methoden eines anderen Objekts zugreifen, wenn wir uns klarmachen, dass BYOB nur mit „erstklassigen“ Komponenten umgeht. So können auch Skripte entweder als Programme oder als Daten angesehen und in unterschiedlichen Kontexten evaluiert werden.



Wählen wir im Skriptbereich von Anne im Block <attribut> of <sprite> einen der Aktenschränke, dann erscheinen dort zusätzlich zu den Standardattributen die lokalen Methoden des anderen Sprites: Anne kann also „sehen“, was ein Aktenschrank kann. Wählen wir die Methode raus und klicken dann auf den Block, dann werden wir allerdings enttäuscht: das Ergebnis der Operation ist nicht etwa ein gespeichertes Element des Aktenschranks, sondern die Methode raus selbst. Das ist auch sinnvoll, denn würde die Methode im Kontext von Anne evaluiert, erhielten wir einen Fehler: Anne verfügt über kein solches Skript. Wir müssen also das übergebene Skript mithilfe des call-Blocks im richtigen Kontext ausführen, und das ist der des Aktenschranks. Ist der noch leer, dann erhalten wir die entsprechende Antwort.

Auf eine ähnliche Weise schreiben wir Daten in den Aktenschrank: wir rufen das richtige Skript im richtigen Kontext auf. Da es sich bei rein um einen command-Block handelt, führen wir diesen mit run aus und übergeben als Parameter (input) die zu speichernden Daten.

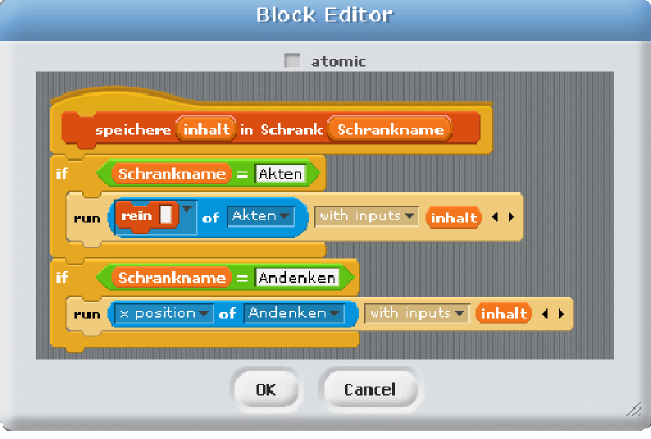
Die Sache wird klarer, wenn wir sie in Einzelschritte zerlegen:

1. Anne sieht bei einem Objekt nach, über welche lokalen Methoden dieses verfügt.
2. Anne wählt eine dieser Methoden aus und weist sie einer Variablen namens Skript zu. Das geht, weil Skripte erstklassig sind und genau dieses zulassen.
3. Handelt es sich um eine Funktion (reporter), dann sendet Anne dem Objekt mithilfe des call-Blocks die Botschaft, die im Skript gespeicherte Operation auszuführen. Bei Bedarf fügt sie die notwendigen Parameter hinzu (call with inputs). Das Ergebnis des Funktionsaufrufs kann sie dann weiter verarbeiten. Handelt es sich um eine Anweisung (command), dann verfährt sie entsprechend mithilfe des run-Blocks.

Wollen wir darüber hinaus auch Attribute eines Sprites von außen verändern, dann kann der in BYOB erweiterte set-Block benutzt werden – aber auch wieder im richtigen Kontext: wir rufen ihn mit run dort auf.

Anne als gut ausgebildete IT-Beauftragte kann solche Befehle natürlich absetzen, ein normaler Benutzer aber wohl nicht. Anne stellt deshalb neue Blöcke zur Verfügung, die als Parameter zusätzlich den zu benutzenden Aktenschrank erhalten. Damit wird die Benutzung sehr vereinfacht.

speichere Hallo in Schrank Akten



Auf ähnliche Weise organisiert sie den Zugriff auf gespeicherte Daten.

set inhalt to hole Daten aus Schrank Akten

Was bedeuteten diese Möglichkeiten für den Unterricht? Die bisher behandelten Aufgabenstellungen können leicht ergänzt werden:

1. Bei den Aktenschränken selbst oder auch bei der IT-Beauftragten kann eine Zugriffskontrolle implementieren werden, ganz einfach durch eine Passwordabfrage oder komplexer mit Benutzerlisten und zugeordneten Passwords. Es ergibt sich eine Fülle kleiner Detailprobleme, die von den Lernenden bearbeitet und sehr anschaulich gelöst werden können, da die bearbeiteten Daten jederzeit durch Setzen eines Häkchens am Bildschirm zu visualisieren sind.

2. Dabei kann und sollte arbeitsteilig gearbeitet werden, denn die Sprites können ja leicht gespeichert und in andere Projekte importiert werden – mit allen erstellten Operationen, ob global oder lokal implementiert.

3. Die Daten selbst können natürlich auch verarbeitet werden. Plausibilitätsprüfungen und Verschlüsselung, Organisationsformen in Listen, Reihungen, Stapeln, Schlangen, Bäumen, Dictionaries ergeben sich aus dem Kontext.

4. Mithilfe der Mesh-Möglichkeiten von BYOB lassen sich die Aufgaben auf unterschiedliche Instanzen von BYOB, die auf unterschiedlichen Rechnern laufen, verteilen. Eine echte Datenübermittlung lässt sich leicht realisieren und anschaulich repräsentieren.

2. Vererbung durch Delegation

Bisher haben wir als Prototypen eines Datenspeichers unsere Kommode, von der wir Kopien ziehen, an die wir – wie in anderen OOP-Sprachen auch – Botschaften in Form von Methodenaufrufen senden. Zur OOP gehört aber zentral das Konzept der Vererbung.

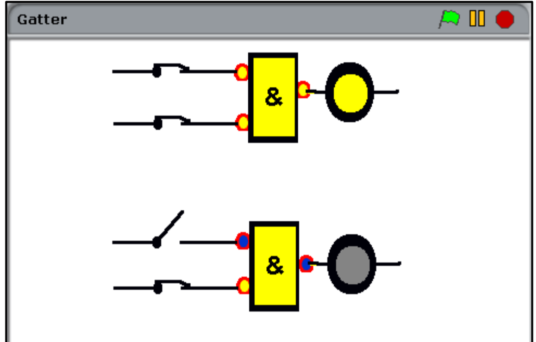
Im Originalartikel von Lieberman[5] werden Objekte als Verkörperung der Konzepte ihrer Klasse verstanden. So steht dort der Elefant Clyde für alles, was der Betrachter unter einem Elefanten versteht. Stellt sich dieser einen Elefanten vor, dann erscheint vor seinem geistigen Auge nicht etwa die abstrakte Klasse der Elefanten, sondern eben Clyde. Spricht er über einen anderen Elefanten, hier: Fred, dann beschreibt er diesen etwa so: *„Fred ist genauso wie Clyde, bloß weiß.“*

Was bedeutet dieser Ansatz für den Lernprozess? Kennt der Lernende nur ein Exemplar einer Klasse (hier: Clyde), dann beschreibt der Prototyp seine Kenntnisse vollständig, eine Abstraktion ist für ihn sinnlos. Lernt er danach andere Exemplare kennen und beschreibt diese durch Modifikationen am Original, ersetzt also einige Methoden durch andere, verändert Attribute und ergänzt neue, dann entsteht langsam das Bild der Klasse selbst als Schnittmenge der gemeinsamen Eigenschaften. Erst jetzt ist der Abstraktionsvorgang für ihn nachvollziehbar und nach einigen Versuchen auch selbst gangbar. Delegation ist damit ein Verfahren, das den Lernprozess selbst abbildet, indem statt Klassen Prototypen erstellt werden.

In BYOB werden Sprites als Prototypen erzeugt und mit den gewünschten Attributen und Methoden ausgestattet. Ist deren Verhalten genügend erprobt worden, dann können Klone entweder im Sprite-Fenster oder mithilfe des clone-Blocks erzeugt werden. Für jedes Sprite kann angezeigt werden, von welchem Sprite es abgeleitet wurde (parent) und über welche Kinder es verfügt (children…). Die Parent-Eigenschaft kann auch nachträglich gesetzt und/oder verändert werden, sodass das System der Abhängigkeiten dynamisch ist.

Ein Klon erbt anfangs alle Attribute und Methoden des Mutterobjekts. Angezeigt wird dieses durch eine „blassere“ Darstellung in den Paletten. Überschreibt ein Sprite geerbte Attribute oder Methoden, dann ersetzen diese wie üblich diejenigen des Prototypen. Löscht man die Überschreibungen wieder (ggf. auch dynamisch mithilfe des delete-Blocks), dann erscheinen erneut die geerbten.

Wir wollen das Verfahren demonstrieren, indem wir ansatzweise einen Digitalsimulator schreiben. Dieser enthält Gatter, hier: UND-Gatter, die alle auf die gleiche Weise arbeiten. Es lohnt sich also, einen Prototypen davon zu bauen und nach Bedarf zu klonen. Ein Gatter besteht aus einem in der Simulation ziemlich funktionslosen Rumpf und einigen Buchsen, die je nach Verdrahtung reagieren. Die „schlauen“ Buchsen sollten also zusammen mit dem Rumpf eine Einheit bilden: eine Aggregation aus Objekten.



[5] [Li86]

In BYOB erreichen wir das entweder interaktiv, indem wir ein Sprite aus dem Sprite-Fenster unten auf ein anderes Sprite auf der Arbeitsebene darüber ziehen. Dieses reagiert darauf sichtbar und fügt das untere Sprite in seine parts-Liste ein. Diese erreichen wir wie gewohnt über den attribute-Block. Hat das Gatter G1 also drei Buchsen als Teile, dann könnte das Ergebnis wie abgebildet aussehen.

Unser Anliegen besteht aber daraus, neue Gatter bei Bedarf dynamisch aus den Prototypen abzuleiten. Dazu müssen wir alle Teile eines Gatters durch Klonen erzeugen und die erforderlichen Beziehungen herstellen. Da wir unterschiedliche Gatter klonen wollen, übergeben wir den Prototypen als Muster und erstellen daraus ein neues Exemplar.

```
new clone of parent with parts
script variables theClone theParts i nextPart xParent yParent xNextPart yNextPart
set xParent to x position of parent
set yParent to y position of parent
set theClone to call the clone block of parent
set theParts to call the attribute parts block of parent
set i to 1
repeat length of theParts
    set nextPart to call the clone block of item i of theParts
    set the attribute anchor block of nextPart to theClone
    set xNextPart to x position of item i of theParts
    set yNextPart to y position of item i of theParts
    set the attribute x position block of nextPart to x position of nextPart + xNextPart - xParent
    set the attribute y position block of nextPart to y position of nextPart + yNextPart - yParent
    change i by 1
report theClone
```

Der Prototyp parent wird als Parameter übergeben.

lokale Variable

Position des Parent bestimmen.

Klon des Parent erzeugen.

Liste der Teile des Parent bestimmen.

Mit allen Teilen tue …

Klon des Teils erzeugen ....

… und mit dem Klon des Gatters verbinden.

Position des Teils relativ zum Gatterklon setzen.

Neues Gatter zurück geben.

Wollen wir Klone löschen, dann müssen wir die verknüpften Teile mit abräumen. Es empfiehlt sich, dafür frühzeitig eine eigene Methode zu schreiben, weil sonst Massen von Klones „per Hand“ zu löschen sind. Das Beispiel demonstriert hoffentlich einsichtig den dynamischen Einsatz des delete-Blocks. Inhaltlich unterscheidet es sich nicht von der Arbeitsweise mit anderen Sprachen.

Zuletzt wollen wir Attribute eines dynamisch erzeugten Objekts verändern können. Dazu schreiben wir die nebenstehende Methode. Man beachte den Stern am Attributnamen (a*): da das übergebene Attribut nicht schon beim rufenden Objekt evaluiert werden soll, wird es im Blockeditor auf unevaluated gesetzt. Diese Möglichkeit ermöglicht gleichzeitig das Schreiben „smarter“ Kontrollstrukturen.

3. Klassen in BYOB

Natürlich müssen wir in BYOB nicht mit Delegation arbeiten. Wir können es, wenn wir es für nötig halten, aber wir können auch den traditionellen Weg über Klassen gehen, wenn es zur Lerngruppe und zum Problem passt. Anders als in vielen anderen Sprachen wird uns die pädagogische Entscheidung frei gestellt, sie ist nicht vom Werkzeug bestimmt.

Eine Klasse ist eine Sammlung von Daten und Methoden, die als Konstruktionsplan für Exemplare dieser Klasse dient. Wird ein solches Objekt erzeugt, dann allokiert das System den erforderlichen Speicherplatz und stellt die Operationen bereit, die mit diesen Daten arbeiten. Genau auf diese Art kann man Klassen in BYOB implementieren. Als Beispiel wollen wir den Datentyp Stapel wählen. Dieser soll eine Liste verwalten, auf die nach dem LIFO-Prinzip zugegriffen wird.

Jeder Stapel muss seinen eigenen Datenbereich haben, der bei Bedarf erzeugt wird. Daten (hier: eine Liste) „leben“ so lange im Speicher eines Computers, wie sie über Referenzen erreicht werden können. Existiert also ein Skript, das diese Daten adressiert, dann werden sie nicht gelöscht. Diesen Mechanismus nutzen wir aus: da in den Skripten einer Klasse auf die Attribute des Objekts Bezug genommen wird, betrachten wir als Objekt eine Sammlung aus Daten und Skripten, die hier ebenfalls als Daten interpretiert werden. Erhält ein Objekt die Botschaft, dass eines seiner Skripte ausgeführt werden soll, dann gibt es das entsprechende Skript zurück. Das rufende Objekt führt dieses dann ggf. mit den entsprechenden Parametern aus. Dieser Mechanismus ist aus dem oben Gesagten schon hinlänglich bekannt.

Probieren wir es also in möglichst knapper Form aus.

Ein Stapel enthalte eine anfangs leere Liste. Als Funktionalität wollen wir zuerst nur die Möglichkeit implementieren, anzufragen, ob der Stapel leer ist. Erhält der Stapel die Botschaft empty?, dann antwortet er mit einem Skript, das diese Frage beantworten kann.

Eine Variable data erzeugen und an eine anfangs leere Liste binden.

Falls die Botschaft empty? ist, das entsprechende Skript zurück geben.

Vereinbaren wir beim rufenden Objekt eine Variable Teststapel und binden diese an einen neuen Stapel, dann brauchen wir noch eine einfache Möglichkeit, den Zustand des Stapels abzufragen: das Prädikat istLeer?. Wir verwenden hier die oben eingeführte „zweistufige" Abfrage.

Nachdem das klappt, erweitern wir die Stack-Klasse um zwei Skripte, um Daten abzulegen (push) und wieder vom Stapel zu nehmen (pull). Benutzt werden diese Skripte wie oben gezeigt in den Methoden des rufenden Objekts. Wir benutzen jetzt die verkürzte Schreibweise:

Mit diesen Operationen können wird dann recht einfach mit Stapeln hantieren.

Es ist noch einmal zu betonen, dass diese Implementierung von Klassen nur eine von vielen Möglichkeiten darstellt. Sie wird im Artikel „*No Ceiling to Scratch*"[6] von Brian Harvey und Jens Mönig eingeführt. Übergeben aber wir einem Objekt z. B. die Botschaft gleich mit allen

[6] [HM10]

erforderlichen Parametern in Listenform, dann kann das Objekt auch selbst die Ergebnisse ermitteln und zurück geben.

4. Zur Notation von BYOB-Programmen

Es kommen immer wieder Einwände, dass BYOB-Programme auf Papier kaum zu notieren und Klausuren deshalb schwer zu stellen wären, denn es könnte ja wohl nicht verlangt werden, dass die Schülerinnen und Schüler dort mit Buntstiften arbeiten. Alternativ finden sich im Internet ausgefeilte Syntaxvorschläge auf diesem Gebiet. Wenn sich mir die Sinnhaftigkeit auch nicht erschließt, eine weitgehend syntaxfreie Sprache auf diesem Weg wieder mit Syntax zu behaften, und die Algorithmen eigentlich in den dafür vorgesehenen Formen (Struktogramme, UML, …) zu notieren wären, folgen jetzt doch noch zwei Vorschläge zu diesem Thema.

Zu zeigen ist also, dass in BYOB grafisch formulierte Algorithmen auch auf Papier aufschreibbar sind. Dazu müssen Methodenköpfe und algorithmische Grundstrukturen darstellbar sein. Die Schachtelung ergibt sich wie bei anderen Systemen auch durch Einrückungen und grafische Hilfsmittel.

Element	BYOB-Symbol	handschriftlich	textuell
Methodenkopf	Methodenname p1 p2	Methodenname p1 p2	Methodenname p1 p2
Funktionskopf	Funktionsname p1 report	Funktionsname p1 Ergebnis:	Funktionsname p1: Ergebnis
Ereignis-behandlung (Beispiel)	when I receive Nachricht	Wenn ich Nachricht empfange	Wenn ich Nachricht empfange
Zählschleife	repeat 10	Wiederhole 10 mal	Wiederhole 10 mal
kopfgesteuerte Schleife	repeat until	Wiederhole bis	Wiederhole bis …
einseitige Alternative	if	Falls ...	Falls …

zweiseitige Alternative	if / else	Falls … / sonst …	Falls … sonst …
Evaluation eines Skripts	run	Führe aus ……..	Führe aus …
Evaluation einer Funktion	call	Rufe auf …..	Rufe auf …
Variablen-vereinbarung	script variables a b c	Variable a b c	Variable a b c
Methodenaufruf eines anderen Objekts	run the script vorwaerts Schritte of object Sprite2 with inputs 20	Rufe „vorwaerts n Schritte“ des Objekts „Sprite2“ mit dem Parameter 20 auf	Rufe „vorwaerts n Schritte“ des Objekts „Sprite2“ mit dem Parameter 20 auf

Beispiel: Sortieren einer Liste in BYOB, formal mit Einrückungen und „per Hand“ geschrieben

```
sortiere die Liste l
  Skriptvariablen i j h
  setze i auf 1
  wiederhole bis i > Länge von l - 1
    setze j auf i + 1
    wiederhole bis j > Länge von l
      falls j Element von l < i Element von l
        setze h auf j Element von l
        ersetze j Element von l durch i Element von l
        ersetze i Element von l durch h
      ändere j um 1
    ändere i um 1
```

```
sortiere die Liste l
  Variable i j k
  setze i auf 1
  wiederhole bis i > Länge von l – 1
    setze j auf i+1
    wiederhole bis j > Länge von l
      falls j-tes Element von l < i-tes Element von l
        setze h auf j-tes Element von l
        ersetze j-tes Element von l durch
                                    i-tes Element von l
        ersetze i-tes Element von l durch h
      ändere j um 1
  ändere i um 1
```

```
sortiere die Liste l
  Variable i j k
  setze i auf 1
  wiederhole bis i > Länge von l – 1
    setze j auf i+1
    wiederhole bis j > Länge von l
      falls j-tes Element von l < i-tes Element von l
        setze h auf j-tes Element von l
        ersetze j-tes Element von l durch i-tes Element von l
        ersetze i-tes Element von l durch h
      ändere j um 1
  ändere i um 1
```

Fazit

Ein namhafter Autor von LOG IN schreibt am 9.11.2011 in einem Forum[7]: *„…auch Byob ist zweifellos ein starkes Produkt. Nur hat es derzeit für die Schule keinerlei Relevanz. Seine entscheidende Schwäche ist insbesondere, dass man nicht zur textuellen Programmierung übergehen kann, also zeit-*

[7] [Ba11]

lebens an die Kacheln oder Blöcke gebunden ist. Ersteres aber ist für die Schule unerlässlich, mindestens ab Klasse 10. Trotz Meister Modrow, der bis zum Abitur Blöcke stapeln will - das geht nicht! Programme sind Texte…“ Trotz der für mich ehrenvollen Erwähnung bin ich verblüfft über die weitreichenden Erkenntnisse des Autors. Anscheinend endet das Lernen nach der Schule, und dass Programme nur in Textform vorliegen können, wird die Entwickler und Nutzer z. B. von LabView, AppInventer oder den UML-Tools erstaunen. Will ich also bis zum Abitur Blöcke stapeln? Zumindest kann man es.

Literatur

[Ba11] Baumann, R.: Forumsbeitrag auf http://forum.world.st/Squeak-Etoys-Homepage-aus-dem-Dornroschenschlaf-wecken-tp3991829p4021485.html

[HM10] Harvey, B.; Moenig, J.: Bringing “No Ceiling” to Scratch: Can One Language Serve Kids and Computer Scientists?, Constructionism 2010, Paris

[Li86] Lieberman, Henry: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, ACM SIGPLAN Notices, Volume 21 Issue 11, Nov. 1986

[Mo11] Modrow, E.: Visuelle Programmierung – oder: Was lernt man aus Syntaxfehlern?, Lecture Notes in Informatics 189, GI-Edition 2011

[MSM11] Modrow, E.: Strecker, K.; Mönig, J.: Wozu Java?, LOG IN 168, 2011