Objektorientiertes Programmieren mit BYOB

von Eckart Modrow

Die auf SCRATCH basierende grafische Entwicklungsumgebung BYOB geht in einigen Bereichen weit über die Möglichkeiten von sonst in der Schule verwendeten Sprachen hinaus. Die Grundfunktionalität und die Eignung für die Oberstufeninformatik etwa im Bereich der Datenstrukturen wurden an anderer Stelle dargestellt (Modrow, 2011; Modrow/Mönig/Strecker, 2011). In diesem Beitrag geht es um die Möglichkeiten im Bereich der objektorientierten Programmierung (OOP).

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 ziemlich genaue 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 problematisch und wird entsprechend kritisiert. Geht es vielleicht auch 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. Dieses Vorgehen beruht auf Henry Liebermans „Delegations-Modell“ (vgl. Lieberman, 1986). Sehen wir es uns einmal genauer an einem Beispiel an!

Beispiel 1: Datenspeicher mit Zugriffsmethoden

Wir erzeugen als Objekt (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. Auf diese Weise können wir zwar beliebige Inhalte in die Liste schreiben und daraus entfernen, aber das genügt nicht, weil ein Datenspeicher sich ja nicht selbst „befüllt“. Dafür benötigen wir einen Zugriff auf die Methoden des Objekts von außen.

Um dies zu demonstrieren, erzeugen wir zwei Kopien unseres Speicher-Objekts, die wir *Akten* und *Andenken* nennen, sowie eine IT-Beauftragte *Anne*, und erhalten damit die Konfiguration von Bild 1. (Kopien enthalten schon anfangs ihre eigenen Attribute, hier die Liste. Erzeugten wir Klone, würden alle mit der gleichen Liste arbeiten, die wir erst überschreiben müssten, um unabhängige Speicher zu erhalten.)

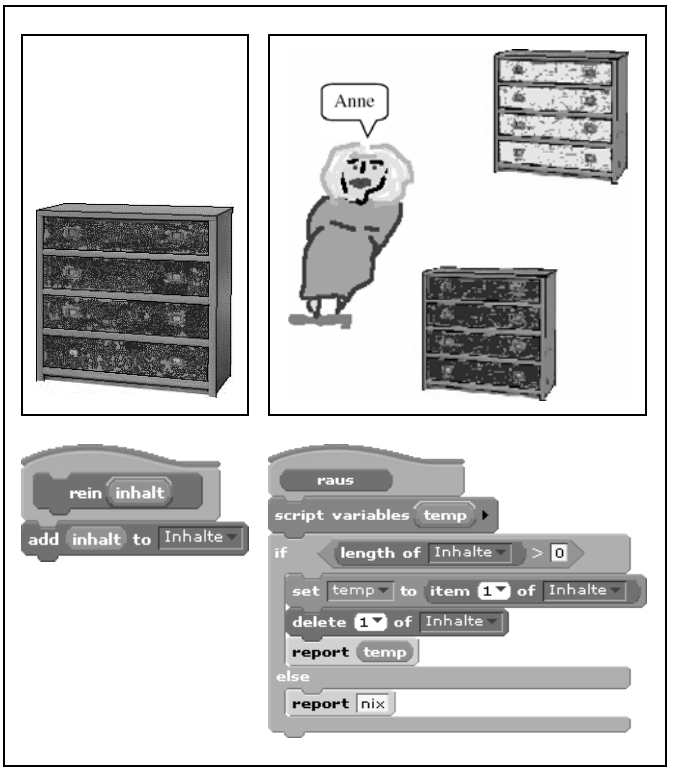


Bild 1: Datenspeicher mit Zugriffsmethoden.

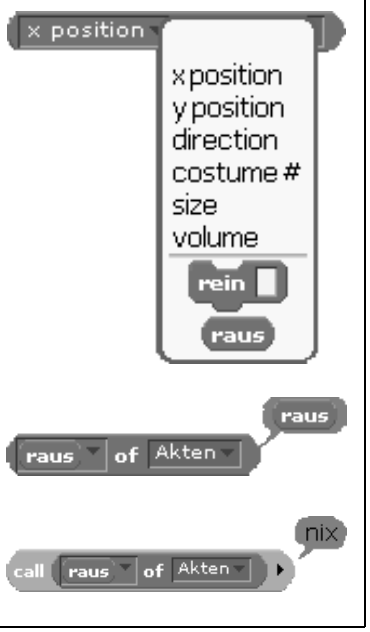


Bild 2:
Lokale Methoden eines Objekts erscheinen im *attribute*-Block des Objekts.
Der Aufruf liefert den Code der Methode, der mit *call* (bei Reporter-Blöcken) und *run* (bei Command-Blöcken) ausgeführt werden kann.

Wie kann nun *Anne* auf ihre Datenspeicher zugreifen? In BYOB sind die SCRATCH-Blöcke <attribute> of <sprite> und set <variable> to <value> stark modifiziert, und zwei neue, object und attribute, sind dazugekommen. Damit 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 (siehe Bild 2, links).

Wählen wir im Skriptbereich von *Anne* im Block <attribut> of <sprite> einen der Aktenschränke, so erscheinen dort zusätzlich zu den Standardattributen die lokalen Methoden des anderen Sprites: *Anne* vermag also zu „sehen", was ein Aktenschrank kann. Wählen

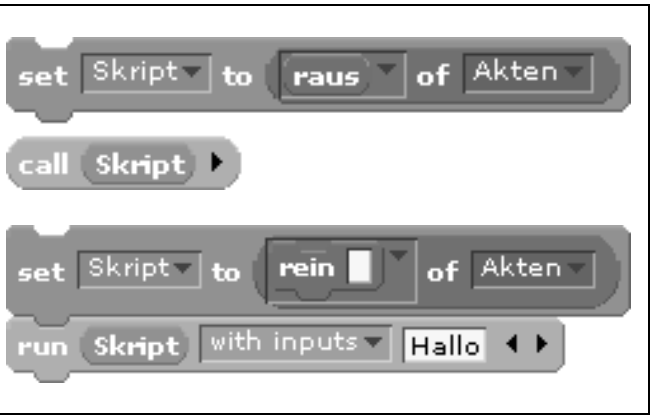


Bild 3: Schrittweises Vorgehen bei der Ausführung von Skripten fremder Objekte.

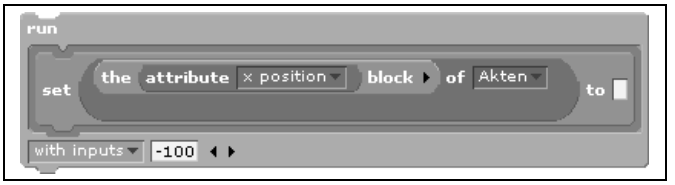


Bild 4:
Setzen eines Attributs eines fremden Objekts.

wir die Methode raus und klicken dann auf den Block, werden wir allerdings enttäuscht, denn 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 er noch leer, erhalten wir die entsprechende Antwort.

Auf eine ähnliche Weise schreiben wir Daten in den Aktenschrank, indem wir das richtige Skript im richtigen Kontext aufrufen. 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 (siehe Bild 2, unten).

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

1. *Anne* sieht bei einem Objekt nach, über welche lokalen Methoden es verfügt.
2. *Anne* wählt eine dieser Methoden aus und weist sie einer Variablen namens *Skript* zu. Das ist möglich, weil Skripte erstklassig sind.
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 weiterverarbeiten. Handelt es sich um eine Anweisung (command), verfährt sie entsprechend mithilfe des run-Blocks (siehe Bild 3).

Wollen wir darüber hinaus auch Attribute eines Sprites von außen verändern, kann der in BYOB erweiterte set-Block verwendet werden – aber wieder im richtigen Kontext: Wir rufen ihn mit run dort auf (siehe Bild 4).

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 (Bild 5, nächste Seite).

Auf ähnliche Weise organisiert sie den Zugriff auf gespeicherte Daten (siehe Bild 6, nächste Seite).

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

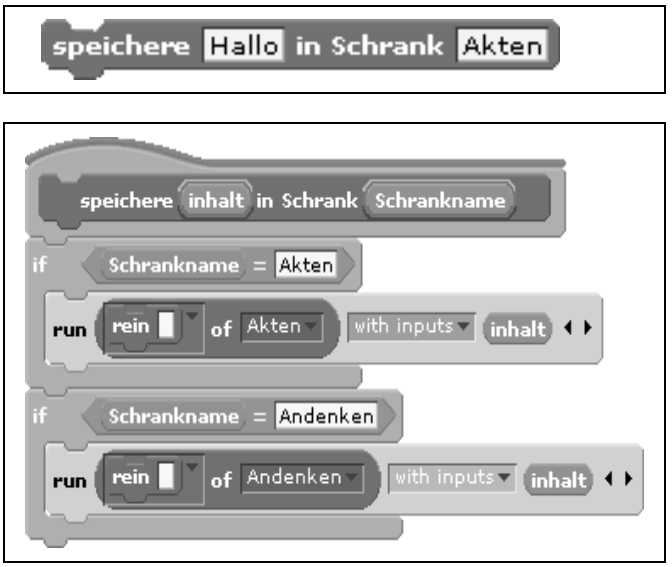


Bild 5: Vereinfachter Schreibzugriff.

1. Bei den Aktenschränken selbst oder auch bei der IT-Beauftragten kann eine Zugriffskontrolle implementiert werden: Ganz einfach durch eine Kennwortabfrage oder – komplexer – mit Benutzerlisten und zugeordneten Kennwörtern. Es ergibt sich eine Fülle kleiner Detailprobleme, die von den Lernenden bearbeitet und 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 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 können die Aufgaben auf Implementierungen von BYOB, die auf unterschiedlichen Rechnern laufen, verteilt werden. Eine echte Datenübermittlung lässt sich leicht realisieren und anschaulich repräsentieren.

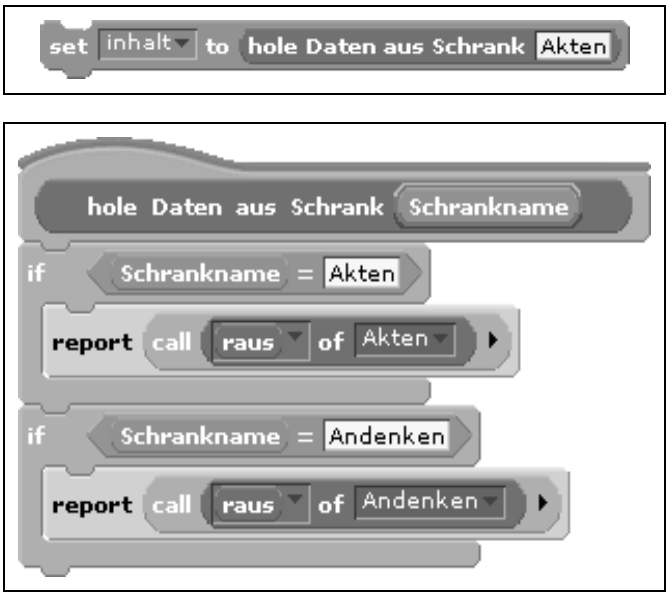


Bild 6: Vereinfachter Lesezugriff.

Vererbung durch Delegation

Bisher haben wir als Prototypen eines Datenspeichers unsere Kommode (Beispiel 1), von der wir Kopien ziehen und an die wir – wie in anderen OOP-Sprachen auch – Botschaften in Form von Methodenaufrufen senden. Zur OOP gehört aber zentral das Konzept der Vererbung.

In dem Artikel *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems* von

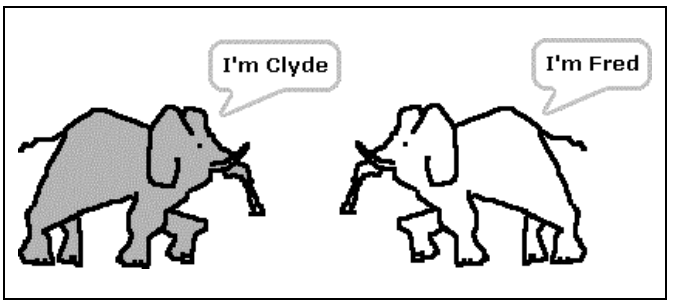


Bild 7a (oben): Die Objekte *Clyde* und *Fred*.

Bild 7b (unten): Direkter Zugriff auf die Vererbungsrelationen (*parent* und *children*).

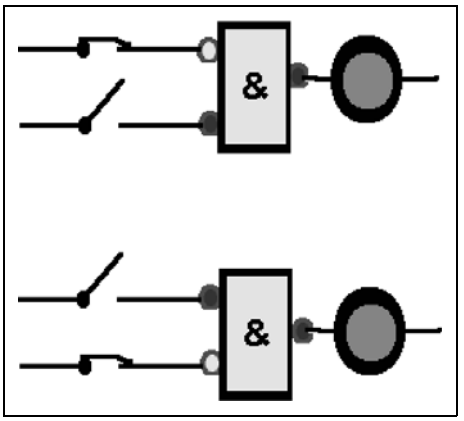


Bild 8: UND-Gatter.

Henry Lieberman werden Objekte als Verkörperung der Konzepte ihrer Klasse verstanden (vgl. Lieberman, 1986). 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 das individuelle Objekt *Clyde* (siehe Bild 7a, vorige Seite). 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 des Originals, ersetzt also einige Methoden durch andere, verändert Attribute und ergänzt sie um neue, dann entsteht allmählich 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 zunächst Prototypen erstellt werden.

In BYOB werden Sprites als Prototypen erzeugt und mit den gewünschten Attributen und Methoden ausgestattet. Ist deren Verhalten genügend erprobt, können Klone entweder im Sprite-Fenster oder mithilfe des clone-Blocks gewonnen werden (siehe Bild 7b, vorige Seite). Für jedes Sprite kann angezeigt werden, von welchem Sprite es abgeleitet wurde (*parent*) und über welche Kinder es verfügt (*children*). Die Eltern-Eigenschaft kann auch nachträglich gesetzt bzw. 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 Prototyps. Löscht man die Überschreibungen wieder (gegebenen-

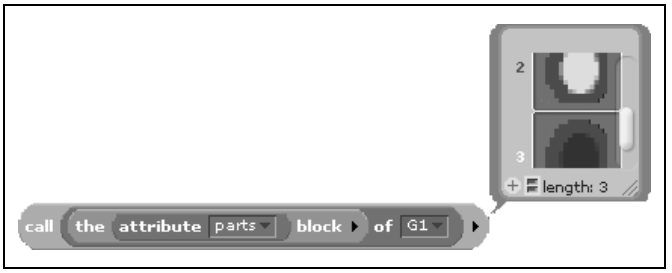


Bild 9: Attribut-Block des Gatters G1.

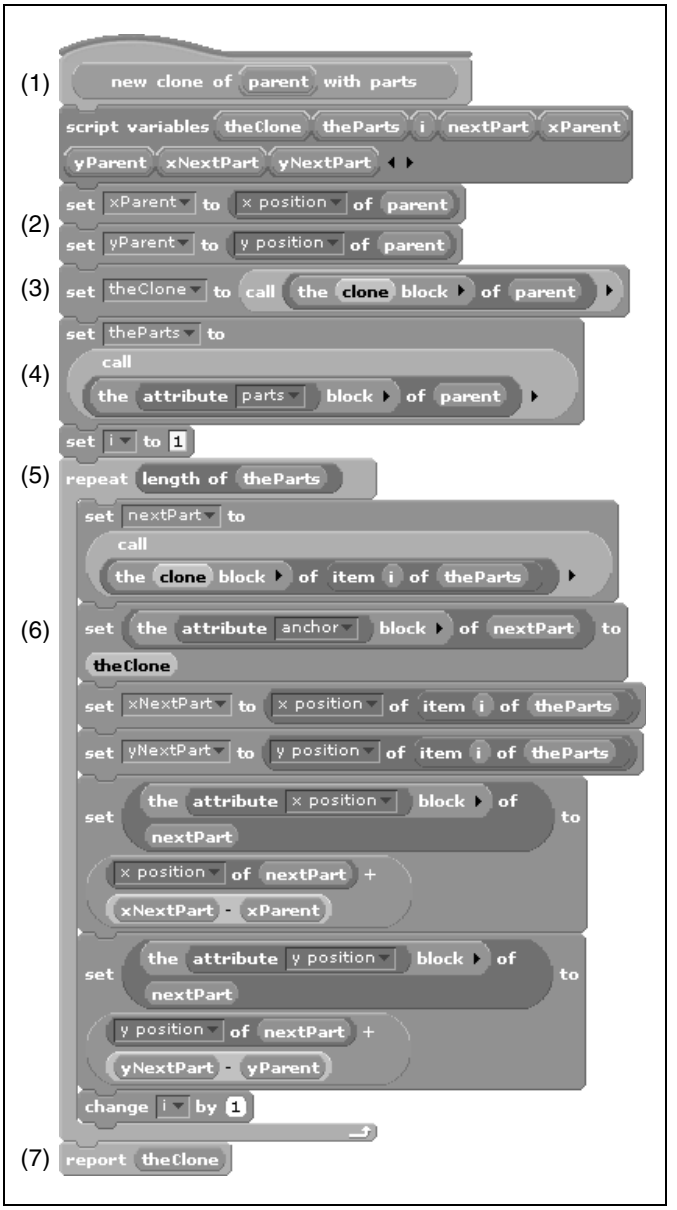


Bild 10: Erzeugung eines neuen Gatters durch Klonen.

Erläuterungen:
(1) Der Prototyp *parent* wird als Parameter übergeben.
(2) Die Koordinaten des *parent* werden bestimmt.
(3) Ein Klon des *parent* wird erzeugt.
(4) Die Teileliste des *parent* wird bestimmt.
(5) Die Liste wird durchlaufen, um die Klone der Teile zu erzeugen.
(6) Zu jedem dieser Klone wird die Position relativ zum Gatterklon festgelegt.
(7) Das neue Gatter wird zurückgegeben.

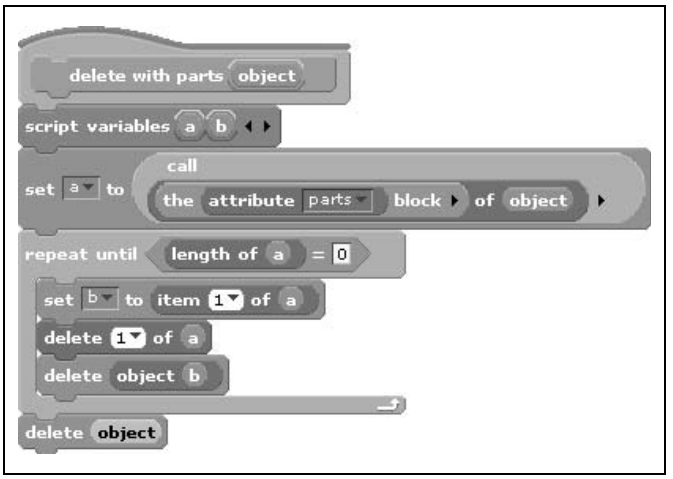


Bild 11: Block zum Löschen von Klonen.

falls auch dynamisch mithilfe des delete-Blocks), dann erscheinen erneut die geerbten. Wir wollen das Verfahren demonstrieren, indem wir ansatzweise einen Digital-Simulator schreiben.

Beispiel 2: Ein Digital-Simulator

Der Simulator 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 (siehe Bild 8, vorige Seite). Die „schlauen" Buchsen sollten also zusammen mit dem Rumpf eine Einheit bilden (Aggregation von Objekten).

In BYOB erreichen wir dies entweder interaktiv, indem wir ein Sprite aus dem Sprite-Fenster unten auf ein anderes Sprite auf der Arbeitsebene darüber ziehen. Letzteres 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 in Bild 9, vorige Seite, aussehen.

Unser Anliegen ist nun aber, 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 erzeugen wollen, übergeben wir den Prototypen als Muster und erstellen daraus ein neues Exemplar (siehe Bild 10, vorige Seite).

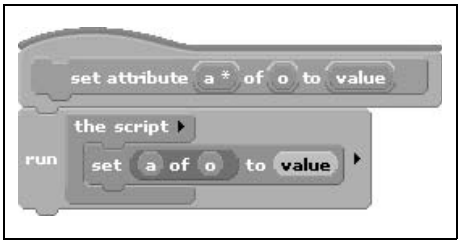


Bild 12: Methode zur Änderung dynamisch erzeugter Objekte.

Wollen wir Klone löschen, müssen wir die verknüpften Teile mit abräumen. Es empfiehlt sich, dafür frühzeitig eine eigene Methode zu schreiben, weil sonst zahlreiche Klone „per Hand" zu löschen sind. Das Beispiel demonstriert den dynamischen Einsatz des delete-Blocks. Inhaltlich unterscheidet es sich nicht von der Arbeitsweise mit anderen Sprachen (siehe Bild 11).

Schließlich möchten wir Attribute eines dynamisch erzeugten Objekts ändern können. Dazu schreiben wir die Methode von Bild 12. 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. Dies ermöglicht zugleich das Schreiben „smarter" Kontrollstrukturen.

Klassen in BYOB

Natürlich sind wir in BYOB nicht gezwungen, ausschließlich mit Delegation arbeiten; 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 freigestellt, sie ist also nicht vom Werkzeug bestimmt.

Eine *Klasse* ist eine Sammlung von Daten und Methoden, die als Konstruktionsplan für Exemplare dieser Klasse dient. Wird ein einzelnes Objekt erzeugt, dann alloziert das System den erforderlichen Speicherplatz und stellt die Operationen bereit, die mit dessen Daten arbeiten. Auf diese Weise kann man Klassen in BYOB implementieren. Als Beispiel wollen wir den Datentyp *Stapel* wählen.

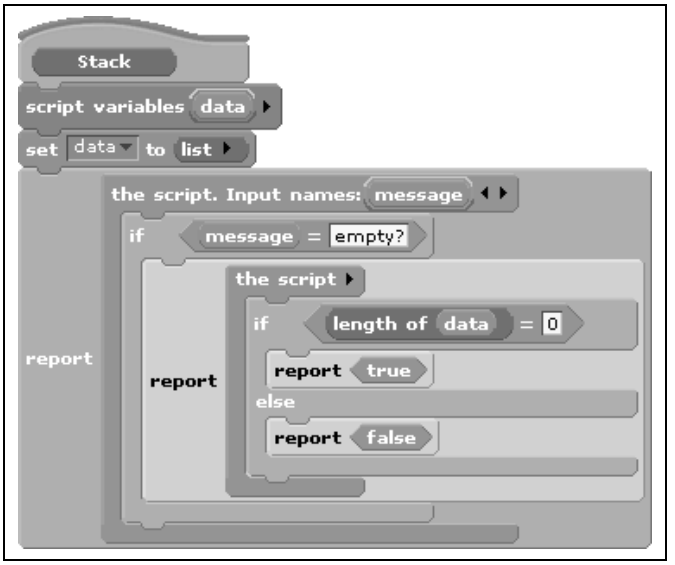


Bild 13: Stapel nur mit Methode *empty?*.

Beispiel 3: Stapel-Verwaltung

Es soll eine Liste verwaltet werden, 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, wie sie über Verweise 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 gegebenenfalls mit den entsprechenden Parametern aus. Dieser Mechanismus ist aus dem oben Gesagten schon hinlänglich bekannt.

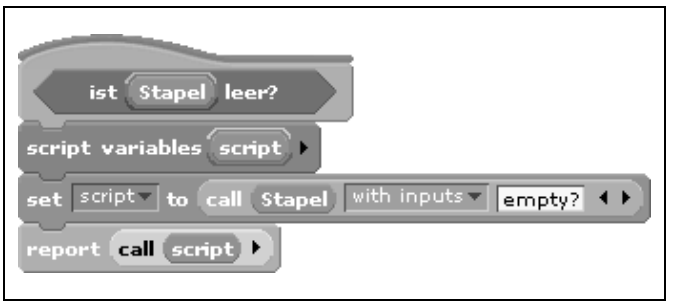
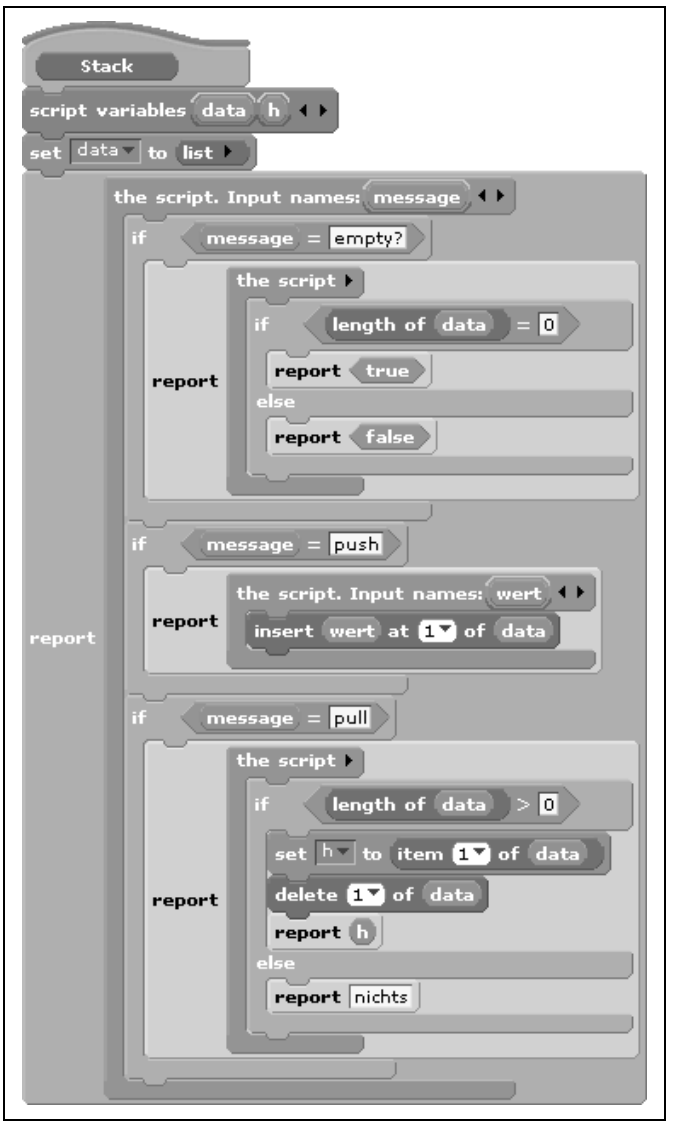


Bild 14 (oben): Abfrage *istLeer?*.

Bild 15 (unten): Die vollständige Stapelklasse.



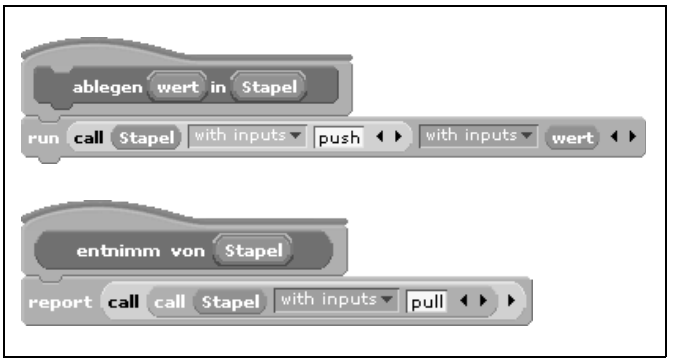


Bild 16: Anwendung der Methoden *push* und *pull*.

Probieren wir es also in möglichst knapper Form aus.

Unser Stapel enthält 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?*, antwortet er mit einem Skript, das diese Frage beantworten kann. Zunächst wird eine Variable *data* erzeugt und an eine anfangs leere Liste gebunden. Falls die Botschaft *empty?* zutrifft, wird das entsprechende Skript zurückgegeben (siehe Bild 13, vorige Seite).

Vereinbaren wir beim rufenden Objekt eine Variable *Teststapel* und binden diese an einen neuen Stapel, benötigen wir noch eine Möglichkeit, den Zustand des Stapels abzufragen: das Prädikat *istLeer?*. Wir verwenden hier die oben eingeführte „zweistufige“ Abfrage (siehe Bild 14).

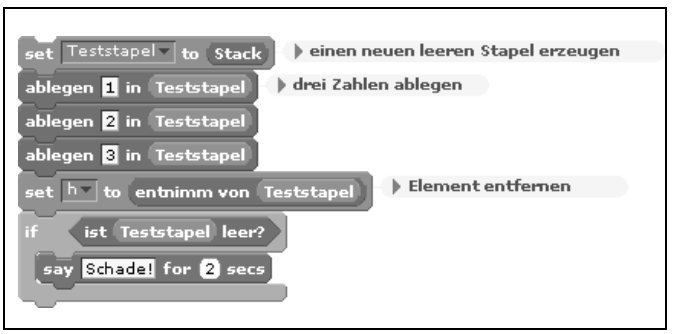


Bild 17: Hantieren mit Stapeln.

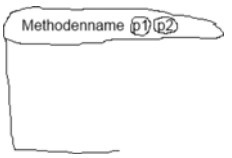
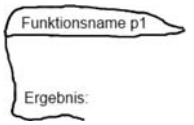
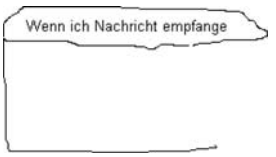
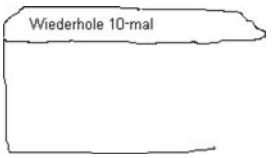
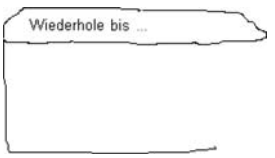
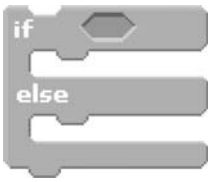
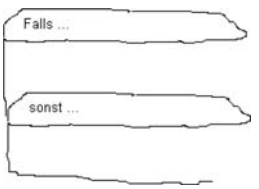
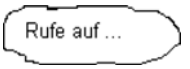
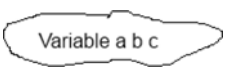
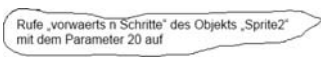
Element	BYOB-Symbol	Handschriftlich	Textuell
Methodenkopf			Methodenname p1 p2
Funktionskopf			Funktionsname p1: Ergebnis
Ereignis-behandlung (Beispiel)			Wenn ich Nachricht empfange
Zählschleife			Wiederhole 10-mal
Kopfgesteuerte Schleife			Wiederhole bis …
Einseitige Alternative			Falls …
Zweiseitige Alternative			Falls … sonst …
Evaluation eines Skripts			Führe aus …
Evaluation einer Funktion			Rufe auf …
Variablen-Vereinbarung			Variable a b c
Methodenaufruf eines anderen Objekts			Rufe „vorwaerts n Schritte" des Objekts „Sprite2" mit dem Parameter 20 auf

Tabelle: Grundelemente von BYOB.

Nachdem dies wunschgemäß funktioniert, erweitern wir die *Stack*-Klasse um zwei Skripte, um Daten abzulegen (push) und wieder vom Stapel zu nehmen (pull; siehe auch Bild 15, Seite 67).

Verwendet werden diese Skripte wie oben gezeigt in den Methoden des rufenden Objekts. Wir benutzen jetzt die verkürzte Schreibweise (siehe Bild 16, Seite 67).

Mit diesen Operationen können wird dann recht einfach mit Stapeln hantieren (siehe Bild 17, Seite 67).

Es ist noch einmal zu betonen, dass diese Implementierung von Klassen nur eine von vielen Möglichkeiten darstellt. Sie wird im Artikel *Bringing „No Ceiling" to Scratch* von Brian Harvey und Jens Mönig eingeführt (Harvey/Mönig, 2010). Übergeben wir aber einem Objekt beispielsweise die Botschaft gleich mit allen erforderlichen Parametern in Listenform, dann kann das Objekt auch selbst die Ergebnisse ermitteln und zurückgeben.

Zur Notation von BYOB-Programmen

Immer wieder werden Einwände der Art geäußert, dass BYOB-Programme auf Papier kaum zu notieren und Klausuren deshalb schwer zu stellen seien, da von den Schülerinnen und Schülern nicht gut verlangt werden könne, mit Buntstiften zu arbeiten. Alternativ finden sich im Internet ausgefeilte Syntaxvorschläge auf diesem Gebiet. Wenn es mir auch nicht besonders sinnvoll zu sein scheint, eine weitgehend syntaxfreie Sprache auf diesem Weg wieder mit Syntax auszustatten, und die Algorithmen eigentlich in den dafür vorgesehenen Formen (Struktogramme, UML usw.) zu notieren wären, folgen jetzt doch noch zwei Vorschläge zu diesem Thema.

Zu zeigen ist also, dass in BYOB grafisch formulierte Algorithmen sich auch auf Papier wiedergeben lassen. Dazu müssen Methodenköpfe und algorithmische Grundstrukturen darstellbar sein. Die Schachtelung ergibt sich wie bei anderen Systemen durch Einrückungen und grafische Hilfsmittel (siehe Tabelle, vorige Seite).

Beispiel 4: Sortieren einer Liste

Der Sortieralgorithmus wird mit Einrückungen wie folgt geschrieben:

```
sortiere die Liste l
  Variablen 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
```

In BYOB sieht der Algorithmus wie in Bild 18 aus und „von Hand" gezeichnet wie in Bild 19.

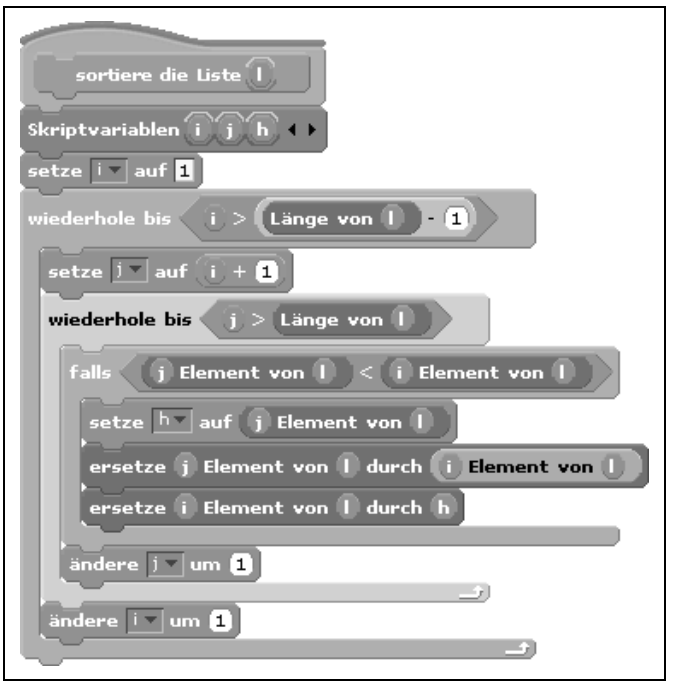
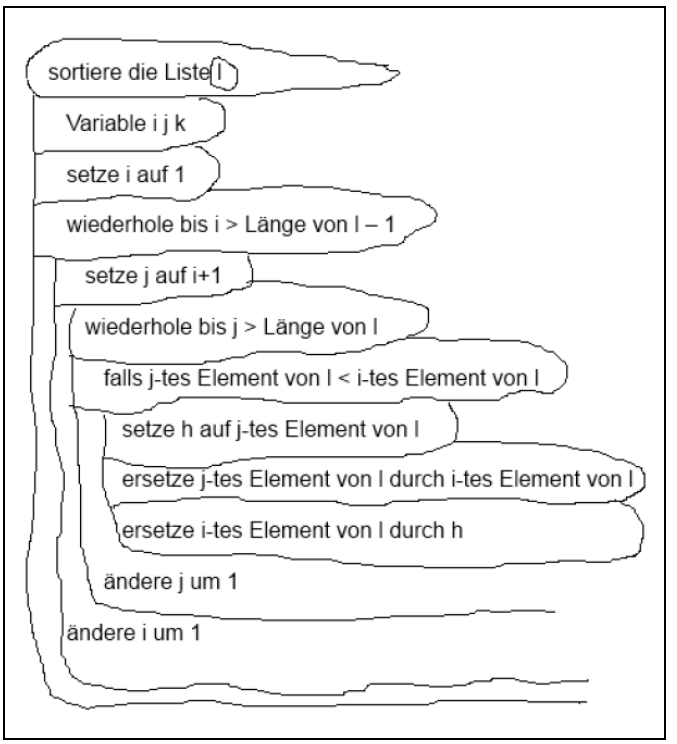


Bild 18 (oben):
Sortieralgorithmus *Sortieren einer Liste* in BYOB.

Bild 19 (unten):
Der Sortieralgorithmus *Sortieren einer Liste* „von Hand".



Fazit

Ein LOG-IN-Autor schreibt am 9.11.2011 in einem Forum (Baumann, 2011):

> […] 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 zeitlebens 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 über die weitreichenden Erkenntnisse des Autors verblüfft. Anscheinend hört das Lernen nach der Schule auf, und dass Programme nur in Textform vorliegen können, wird die Entwickler und Nutzer z. B. von LabView, AppInventer oder den UML-Werkzeugen erstaunen. Will ich also bis zum Abitur Blöcke stapeln? Möglich ist es jedenfalls, wie oben gezeigt.

Prof. Dr. Eckart Modrow
Didaktik der Informatik
Universität Göttingen
Goldschmidtstrasse 7
37077 Göttingen

E-Mail: emodrow@informatik.uni-goettingen.de

Literatur und Internetquellen

Baumann, R.: Squeak-Etoys-Homepage aus dem Dornröschenschlaf wecken – Forumsbeitrag (2011).
http://forum.world.st/Squeak-Etoys-Homepage-aus-dem-Dornroschen schlaf-wecken-tp3991829p4021485.html

BYOB – Build Your Own Blocks:
http://byob.berkeley.edu/

Harvey, B.; Mönig, J.: Bringing „No Ceiling“ to Scratch – Can One Language Serve Kids and Computer Scientists? In: Constructionism 2010 conference, Paris, 2010, S. 1–10.
http://byob.berkeley.edu/BYOB.pdf

Lieberman, H.: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In: ACM Sigplan Notices, Band 21 (1986), H. 11, S. 214–223.
http://web.media.mit.edu/~lieber/Lieberary/OOP/Delegation/Delegation.html

Modrow, E.: Visuelle Programmierung – oder: Was lernt man aus Syntaxfehlern? In: M. Thomas (Hrsg.): Informatik in Bildung und Beruf. INFOS 2011 – 14. GI-Fachtagung Informatik und Schule, 12.–15. September 2011 in Münster. Reihe „GI-Edition Lecture Notes in Informatics“, Band P-189. Bonn: Köllen Verlag, 2011, S. 27–36.

Modrow, E.; Mönig, J.; Strecker, K.: Wozu Java? – Plädoyer für grafisches Programmieren. In: LOG IN, 31. Jg. (2011), H. 168, S. 35–41.

Alle Internetquellen wurden zuletzt am 16. Januar 2012 geprüft.