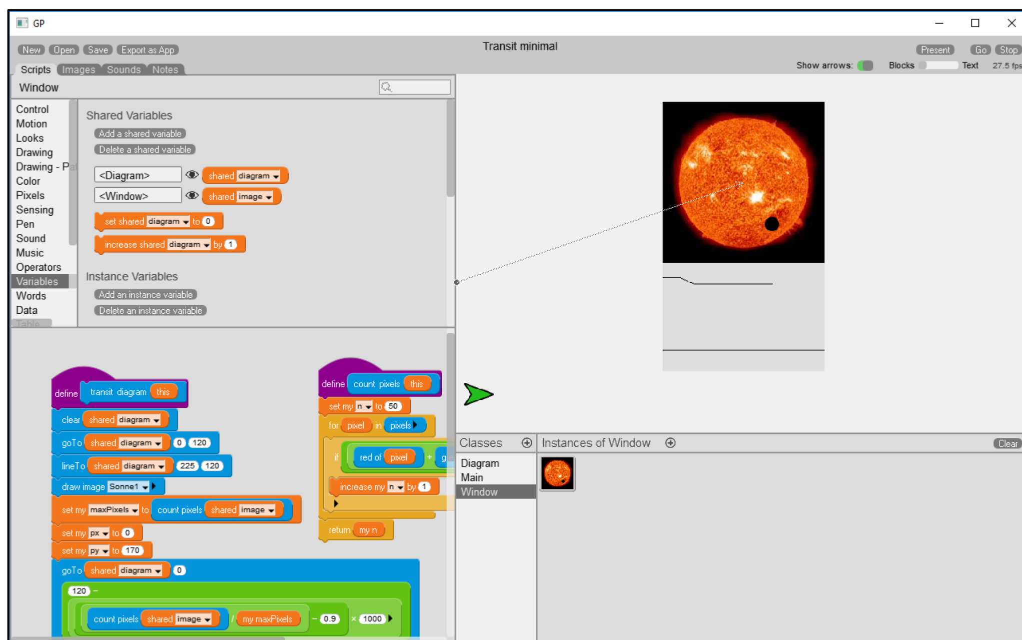
Eckart Modrow

Informatik mit GP

- GP in Beispielen -



© Eckart Modrow 2017

emodrow@informatik.uni-goettingen.de

Vorwort

Das vorliegende Skript stellt, ähnlich wie der Vorgänger *„Informatik mit BYOB“*[1], anhand einer Sammlung von Programmierbeispielen einige – aber bei Weitem nicht alle – Möglichkeiten der grafischen Sprache GP dar. GP bildet nach Scratch und BYOB/snap! den nächsten Schritt in der Entwicklung der grafischen Werkzeuge. Das System überwindet eine Reihe von Einschränkungen, die bei ihren Vorläufern noch vorhanden waren, und damit auch eine Reihe von Argumenten, die gegen grafische Sprachen vorgebracht wurden (und werden). GP ähnelt seinen Vorläufern äußerlich stark, und das ist auch kein Wunder, denn deren Schöpfer entwickeln auch GP. Damit ist das System für Nutzer von Scratch oder snap! sofort einsetzbar. Es enthält intern aber auch starke Unterschiede etwa beim Objektmodell, die bei etwas fortgeschrittenen Anwendungen entsprechend stark zu berücksichtigen sind.

Die Auswahl der folgenden Beispiele ist relativ konservativ, lehnt sich teilweise noch eng an bestehenden Informatikunterricht an. Das ist Absicht. Ich hoffe, damit die unterrichtenden Kolleginnen und Kollegen vom traditionellen Unterricht „abzuholen“ und auf den Weg zu einem sehr an Kreativität, aber auch an der Vermittlung informatischer Konzepte ausgerichteten Unterricht mitzunehmen. Die ersten Beispiele sind sehr ausführlich und beschreiben detailliert den Umgang mit GP. In den späteren Kapiteln werden eher die Möglichkeiten der Sprache illustriert, am Ende ohne direkten Anwendungsbezug. Dieser Kompromiss ist dem Platzbedarf geschuldet, weil erweiterte Konzepte eigentlich auch erweiterte Problemstellungen erfordern.

Die Beispiele wurden durchgehend in der Betaversion GP 075 realisiert.

Ich bedanke mich sehr bei John Maloney und Jens Mönig für ihre Unterstützung – und für die Ergebnisse ihrer Arbeit. Die Lernenden werden es ihnen danken!

Ansonsten wünsche ich viel Freude bei der Arbeit mit GP!

Göttingen am 2.10.2017

[1] E. Modrow, Informatik mit BYOB, https://www.uni-goettingen.de/de/informatik-mit-byob/423680.html

Inhalt

Vorwort 2
Inhalt 3

1 Zu GP 5
- 1.1 Blockorientierte Sprachen 5
- 1.2 Objektorientierte Sprachen 5
- 1.3 Was ist GP? 6
- 1.4 Was ist GP nicht? 8
- 1.5 Der GP-Bildschirm 8
- 1.6 Mit GP arbeiten: Planetensysteme 9
  - 1.6.1 Erster Ansatz: Der Planet kennt alles 9
  - 1.6.2 Zweiter Ansatz: Die Sonne lernt dazu 11
  - 1.6.3 Dritter Ansatz: Jeder gegen Jeden 13

2 Ein Zeichenprogramm 17
- 2.1 Mit Knöpfen arbeiten 17
- 2.2 Klassen exportieren und importieren 21
- 2.3 Aufgaben 22
- 2.4 Einfache Algorithmik und Fehlersuche 23
  - 2.4.1 Rechtecke zeichnen 23
  - 2.2.2 Fehler finden 27
  - 2.2.3 Aufgaben 29
- 2.5 Rechtecke oder Kreise zeichnen 30
- 2.6 Arbeitsteilig vorgehen 33
- 2.7 Aufgaben 34

3. Simulation eines Federpendels 35
- 3.1 Die Uhr 36
- 3.2 Der Erreger 36
- 3.3 Der Faden 37
- 3.4 Die Kugel 38
- 3.5 Der Stift 39
- 3.6 Das Zusammenspiel der Komponenten 39
- 3.7 Weshalb handelt es sich um eine Simulation? 40

4. Ein Barcodescanner 41
- 4.1 Der EAN8-Code 41
- 4.2 Blöcke als Strukturierungshilfe 41
- 4.3 Aufgaben 45

5. Planeten-Transits 46
- 5.1 Das Szenario 46
- 5.2 Eine Diagramm-Klasse 46
- 5.2 Eine Window-Klasse 47

6. Computeralgebra: funktional programmieren 48
6.1 Funktionsterme 48
6.2 Funktionsterme mit einer Helper-Klasse parsen 49
6.3 Funktionsterme ableiten 53
6.4 Funktionsterme berechnen und Graphen zeichnen 55
6.5 Aufgaben 58

7. Rekursive Kurven 59
7.1 Die Schneeflockenkurve 59
7.2 Die Hilbertkurve 60
7.3 Aufgaben 61

8. Listen und verwandte Strukturen 62
8.1 Sortieren mit Listen – durch Auswahl 62
8.2 Sortieren mit Listen – Quicksort 64
8.3 Kürzeste Wege mit dem Dijkstra-Verfahren 65
8.4 Matrizen und Tabellen 70
8.5 Aufgaben 72

9. Im Netz arbeiten 73
9.1 SQL-Datenbanken verwenden 73
9.2 JSON-Datenbanken 79
9.3 Chatten 82
9.4 Aufgaben 85
9.5 Ein Sensorboard benutzen 86

10. Objektorientierte Programmierung 88
10.1 Die Kommunikation zwischen Objekten 89
10.2 Aufgaben 91
10.3 Programmierung mit Prototypen 92
10.4 Magnete 93
10.5 Ein lernender Roboter 95
10.6 Aufgaben 99

1 Zu GP

1.1 Blockorientierte Sprachen

GP *(http://gpblocks.org/versions/*) ist eine Nachfolgerin u.a. von BYOB (*Build Your Own Blocks),* deren Name schon einen Teil des Programms beschreibt: die Nutzer, bei denen es sich z. B. um Lernende an Schulen und Universitäten handelt, benutzen vorhandene Befehle in Form von Blöcken und werden in die Lage versetzt, eigene neue *Blöcke* zu entwickeln. Aus Kombinationen von beiden bestehen dann die entwickelten Programme (*Skripte*). Man muss wissen, dass praktisch alle Programmiersprachen *block-orientiert* sind: Befehlsfolgen lassen sich unter einem neuen Namen zusammenfassen. Den so entstehenden neuen *Befehlen* können bei Bedarf Werte (*Parameter*) übergeben werden, mit denen sie dann arbeiten, und sie können auch Ergebnisse zurückliefern. Damit gewinnen wir mehrere Vorteile:

Vorteile block-orientierter Sprachen

- **Programme werden kürzer**, weil Programmteile in die Blöcke ausgelagert und mehrfach verwendete Befehlsfolgen nur einmal geschrieben und dann unter dem neuen Namen mehrfach benutzt werden.
- **Programme enthalten weniger Fehler**, weil die Blöcke weitgehend unabhängig voneinander entwickelt und getestet werden und die aktuell entwickelte Befehlsfolge somit kurz und übersichtlich bleibt. „Lange“ Programmteile sind nur sehr selten notwendig und meist ein Zeichen für einen schlechten Programmierstil.
- **Programme erhalten einen eigenen Stil**, weil die neuen Befehle die Art spiegeln, in der die Programmierenden Probleme lösen.
- **Die Programmiersprache wird erweitert**, weil die erstellten Blöcke neue Befehle und somit auch neue Möglichkeiten repräsentieren.

1.2 Objektorientierte Sprachen

Werden etwas umfangreichere Probleme bearbeitet, dann wächst auch die Zahl der zu lösenden Teilprobleme. Oft lassen sich diese zu Gruppen zusammenfassen, die konkreten *Objekten* zuzuordnen sind. Oft tauchen diese Teilprobleme auch immer wieder auf, sodass sie sich lösen lassen, wenn entsprechende Objekte bereitgestellt werden, z. B. in *Bibliotheken*. Ein wichtiger Aspekt dieser Arbeitsweise ist, dass sich so arbeitsteilige Teamarbeit gut realisieren lässt, bei der die unterschiedlichen Teams Objekte erzeugen, die Teilaufgaben lösen. Natürlich müssen die Arbeitsergebnisse dann auch zusammengefasst werden können. Die objektorientierte Arbeitsweise wird oft realisiert, indem *Klassen* erzeugt werden, die das Verhalten einer Gruppe ähnlicher Objekte beschreiben. Von diesen Klassen werden dann *Instanzen* (Exemplare) erzeugt, die die Probleme lösen sollen. Für Anfänger geeigneter ist das *Prototypen*-basierte Vorgehen, bei dem für jede Objektgruppe ein Beispiel, der Prototyp, erzeugt wird, der schrittweise entwickelt und getestet wird. Ist man mit dem Ergebnis zufrieden, dann werden weitere Objekte dieser Klasse durch Vervielfältigung (*Klonen*) des Prototyps abgeleitet.

Das objektorientierte Vorgehen hat die folgenden Vorteile:

Vorteile objektorientierter Sprachen

- **Probleme werden übersichtlicher**, weil Teilprobleme Objekten zugeordnet und (weitgehend) unabhängig gelöst werden können.
- **Probleme werden anschaulicher**, weil die Aufteilung in Objekte oft der intuitiven Anschauung entspricht, sodass „Alltagswissen“ in die Lösungen einfließen kann.
- **Problemangepasste Werkzeuge lassen sich bereitstellen,** weil entsprechende Bibliotheken existieren oder erzeugt werden.
- **Die Zusammenarbeit wird erleichtert**, weil objektorientiertes Arbeiten die weitgehende Isolierung der Problemlösungen voneinander nahelegt, sodass sich die unterschiedlichen Gruppen wenig stören.

1.3 Was ist GP?

die Entwickler

GP (*General-Purpose Blocks Programming Language*) wurde (und wird) von *John Maloney*, *Jens Mönig* und anderen entwickelt und im Internet frei zur Verfügung gestellt[2]. Sie ähnelt von der Oberfläche und dem Verhalten her *Scratch*[3], einer ebenfalls freien Programmierumgebung für Kinder, die am *MIT*[4] entwickelt wurde. Die umgesetzten Konzepte gehen allerdings weit darüber hinaus. GP ist eine voll entwickelte Programmiersprache, die folgerichtig auch in (fast) allen Problembereichen eingesetzt werden kann. Das betrifft auch rechenintensive Bereiche, z. B. in der Bildverarbeitung, denn GP ist sehr schnell. Das ist nicht selbstverständlich und war ein Manko ihrer Vorgänger. Scratch und BYOB/snap! sind weitgehend damit beschäftigt, den Systemzustand zu kontrollieren und es so z. B. zu gestatten, Endlosschleifen zu unterbrechen oder Zugriffsfehler auf Datenstrukturen zu „tolerieren“. Für die eigentliche Programmausführung bleibt dann wenig Zeit. GP leistet das Gleiche, ist aber trotzdem schnell.

GP ist schnell

kaum Syntaxfehler

GP ist eine grafische Programmiersprache: Programme (*Skripte*) werden nicht als Text eingegeben, sondern aus *Kacheln* zusammengesetzt. Da sich diese Kacheln nur zusammenfügen lassen, wenn dieses einen Sinn ergibt, werden „falsch geschriebene“ Programme weitgehend verhindert. GP ist deshalb weitgehend *syntaxfrei*. Völlig frei von Syntax ist sie trotzdem nicht, weil manche Blöcke unterschiedliche Kombinationen von Eingaben verarbeiten können: stellt man diese falsch zusammen, dann können durchaus Fehler auftreten. Allerdings passiert das eher bei fortgeschrittenen Konzepten. Wendet man diese an, dann sollte man auch wissen, was man tut.

zwei Programmierstile

GP ist außerordentlich „friedlich“: Fehler führen nicht zu Programmabstürzen, sondern werden durch den Aufruf eines *Debugger*-Fensters angezeigt, das beschreibt, wodurch der Fehler verursacht wurde – ohne dramatische Folgen. Die benutzten Kacheln, zu denen auch die entwickelten Blöcke gehören, „leben“ immer. Sie lassen sich durch Mausklicks ausführen, sodass ihre Wirkung direkt beobachtet werden kann. Damit wird es leicht, mit den Skripten zu experimentieren. Sie lassen sich testen, verändern,

[2] https://gpblocks.org/
[3] http://scratch.mit.edu/
[4] Massachusetts Institute of Technology, Boston

in Teile zerlegen und wieder gleich oder anders zusammensetzen. Wir erhalten damit einen zweiten Zugang zum Programmieren: neben der Problemanalyse und dem damit verbundenen *top-down*-Vorgehen tritt die experimentelle *bottom-up*-Konstruktion von Teilprogrammen, die zu einer Gesamtlösung zusammengesetzt werden können.

Unterstützung verschiedener Programmier-paradigmen

GP unterstützt neben der üblichen imperativen und funktionalen Programmierung inklusive anonymer Funktionen (*Lambda*) ein klassenbasiertes Objektmodell, wobei zu jeder Klasse mindestens eine Instanz gehört, deren Eigenschaften direkt erprobt werden können. Fasst man diese Instanz als einen Prototyp auf, dann ist GP Prototypen-orientiert.

anschaulich und erweiterbar

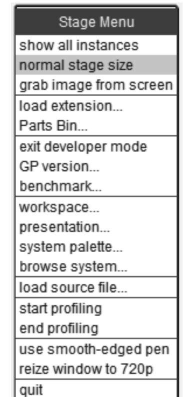
GP ist anschaulich: sowohl die Programmabläufe wie die Belegungen der Variablen lassen sich bei Bedarf am Bildschirm anzeigen und verfolgen.

GP ist erweiterbar: neue Funktionalitäten können exportiert und importiert werden.

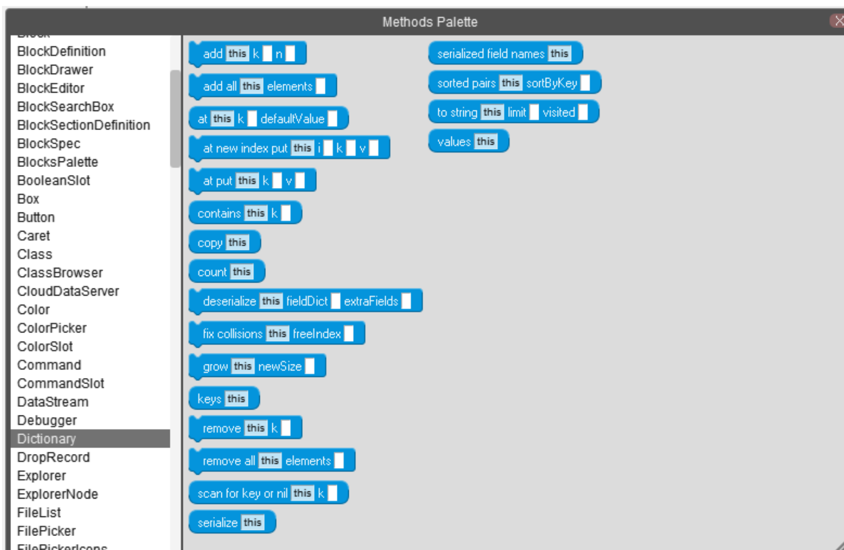
objektorientiert

GP ist objektorientiert, natürlich. Methoden und Attribute werden explizit global oder lokal vereinbart. Soll Funktionalität über die eigene Klasse hinaus bereitgestellt werden, dann kann das einfach und übersichtlich z. B. in einer neuen Befehlspalette geschehen. GP unterstützt allerdings keine Vererbung – und das mit Absicht. Die Entwickler sind der Ansicht, dass für den ins Auge gefassten Nutzerkreis Vererbung nicht notwendig ist und viele Dinge nur verkompliziert.

GP gestattet den vollen Zugriff auf das System: sowohl das Netzwerk wie externe Hardware kann direkt angesteuert werden, Cloud-Ressourcen sind verfügbar, Grafik- und Musikanwendungen lassen sich sehr gut realisieren – und die Ergebnisse lassen sich als Stand-Alone-Anwendungen exportieren. GP läuft auf der meisten Hardware und unter fast allen Betriebssystemen, u. a. im Browser auch auf Tablets.



Findet man unter den Blöcken der Befehlsregister nicht das Geeignete, dann gestattet der *Entwickler-Modus* im *Stage-Menu* den Zugriff auf alle im System vorhandene Befehle. Da GP selbst in dieser Sprache geschrieben wurde, stehen ungleich mehr Blöcke als in den Registern dargestellt zur Verfügung. Mithilfe der *Methoden-Tabelle* lässt sich das System durchsuchen. Findet man geeignete Werkzeuge, dann lassen sich diese nutzen wie andere auch. Hier dargestellt sind die Methoden der *Dictionary*-Klasse.

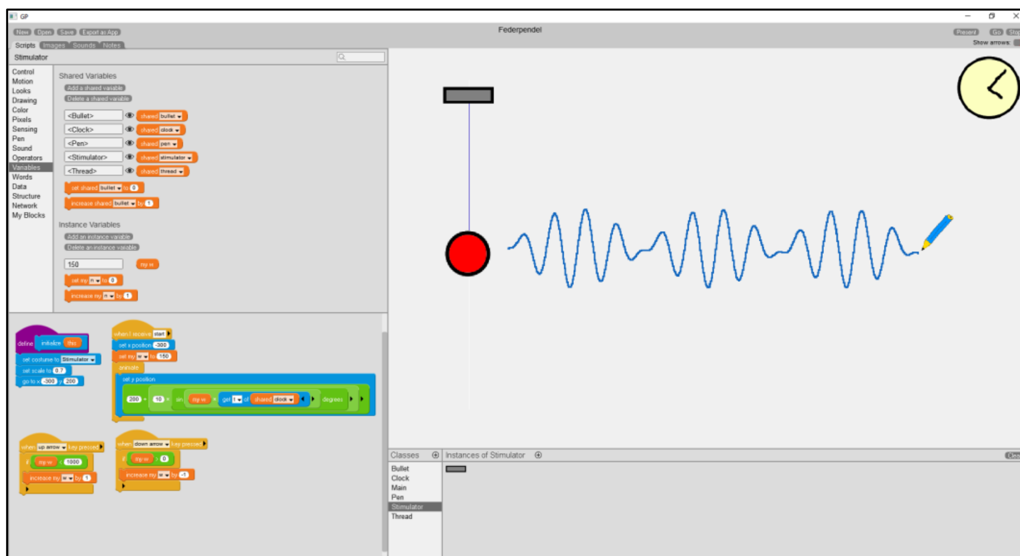


1.4 Was ist GP nicht?

Zielgruppen und Grenzen

Die Zielgruppen von GP sind einerseits Lernende an Schulen und Universitäten, andererseits „Casual Programmers“, also Nutzer, die programmieren können, dieses aber nur gelegentlich bei Bedarf tun. Als Beispiel mag ein Biologe dienen, der irgendwann in der Vergangenheit eine textbasierte Programmiersprache erlernte, also die grundlegenden Konzepte kennt, sich aber nicht mit der aktuellen Version dieses Werkzeugs auseinandersetzen mag. Um ein anstehendes Problem zu lösen, etwa die Auswertung von Mikroskopbildern zu automatisieren, benutzt er dann GP. Aus dieser Beschreibung ergeben sich die Grenzen von GP: die Sprache ist nicht für professionelle Programmierer gedacht.

1.5 Der GP-Bildschirm



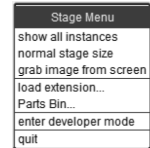
Die Oberfläche von GP.

Der GP –Bildschirm besteht unterhalb der Menüleiste aus sechs Bereichen. (Sollten Sie weniger sehen, dann verschieben Sie Begrenzung zwischen Skript- und Ausgabefenster etwas. Die unteren Bereiche werden dann auch sichtbar.)

die Bildschirmbereiche

- Ganz links befinden sich die Befehlsregister, die in die Rubriken *Control*, *Motion*, *Looks*, *Sound* usw. gegliedert sind. Klickt man auf den entsprechenden Knopf, dann werden rechts davon die Inhalte dieser Rubrik angezeigt. Passen sie nicht alle auf den Bildschirm, dann kann man in der üblichen Art den Bildschirmbereich scrollen. Wählt man oberhalb dieses Bereichs eine der Kategorien *Images*, *Sounds* und *Notes* aus, dann können Bilder oder Klänge geladen, gespeichert oder bearbeitet sowie Anmerkungen zum Projekt eingegeben werden.
- Darunter befindet sich der Skript-Bereich, in dem sich bei Anwahl der *Script*-Paletten Skripte erzeugen, testen und verändern lassen.
- Rechts-oben befindet sich das Ausgabefenster, in dem sich die Objekte, die *Sprites*, bewegen. Das aktuell ausgewählte Sprite wird durch einen Pfeil angezeigt. Stört dieser, dann lässt er sich auch abschalten (mit „*show arrows*“ über dem Fenster).

Wechsel des Arbeitsmodus



- Klickt man (in Windows) rechts in das Ausgabefenster, dann erhält man weitere Optionen z. B. zum Wechsel in den *„developer mode"*, zum Laden von Bibliotheken, zum Erzeugen von *Screenshots* usw.
- In der unteren Mitte befindet sich der Klassenbereich, in dem die zur Verfügung stehenden Klassen angezeigt werden. Neue Klassen werden hier erzeugt, bestehende gelöscht oder umbenannt. Insbesondere können Klassen exportiert und in neue Projekte geladen werden.

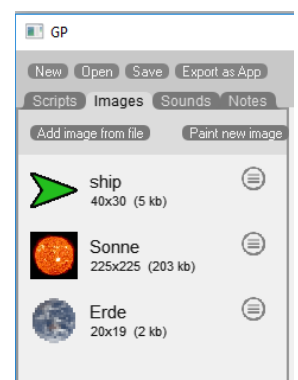
Klassen und Instanzen

- Rechts davon werden die Instanzen der ausgewählten Klasse angezeigt. Sie können dort selektiert, im Aussehen bearbeitet, erzeugt und gelöscht werden. Von Ausnahmen abgesehen gehört zu jeder Klasse immer mindestens eine Instanz als Prototyp.
- Die Menüleiste selbst bietet links die üblichen Menüs zum Laden und Speichern des Projekts.
- Ganz rechts finden wir drei Buttons *Present*, *Go* und *Stop*. Der erste schaltet in den Präsentationsmodus, in dem nur der Arbeitsbereich sichtbar ist – und der durch Drücken der *ESC*-Taste verlassen wird. Der zweite schickt die Botschaft „*Go*" durch das System, mit deren Hilfe Skripte gestartet werden können, der dritte stoppt die Programmausführung.

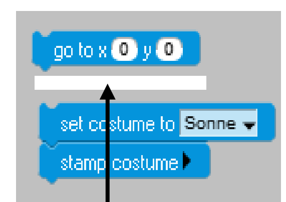
1.6 Mit GP arbeiten: Planetensysteme

In diesem kleinen Projekt soll ein erster Überblick über die Arbeit mit GP gegeben werden, und zwar von ganz einfachen Lösungen bis zu etwas komplexeren.

1.6.1 Erster Ansatz: Der Planet kennt alles (z. B. die Sonnenmasse)!

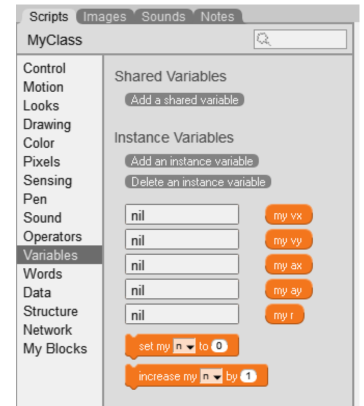


1. Wir bauen unser Sonnensystem „per Hand" zusammen und lassen dann den Planeten laufen. Dafür suchen wir Bilder der Sonne und eines Planeten im Internet, verkleinern sie in einem gängigen Grafikprogramm und speichern sie als *png*-Dateien. Die beiden Bilder laden wir in GP im Register *Images* und speichern unser Projekt unter einem geeigneten Namen – sicherheitshalber.
2. Wir zeichnen die Sonne in die Mitte der Arbeitsfläche (Koordinaten (0|0)). Dazu wechseln zum Register *Scripts*, wo wir Programmbefehle eingeben und ausführen können, wählen als Kostüm der (einzigen) Instanz der vorgegebenen Klasse *MyClass* das Sonnenbild, schicken sie zur Mitte und drucken das Kostüm auf die Arbeitsfläche. Da bleibt es jetzt. Die Blöcke haben wir dazu in den Skriptbereich verschoben und „zusammengesteckt". Ein Klick auf den Blockstapel, das *Skript*, führt diesen aus. Dabei erhält er einen grünen Rand. Der *goTo*-Block befindet sich in der *Motion*-Palette, *set costume to* in der *Looks*- und *stamp costume* in der *Pen*-Palette.

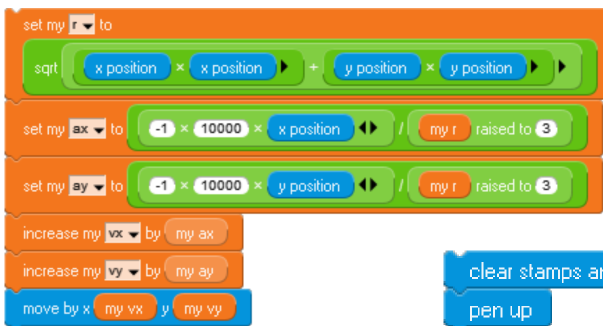


Dort, wo Blöcke „einrasten" können, wird bei Annäherung ein weißer Balken sichtbar.

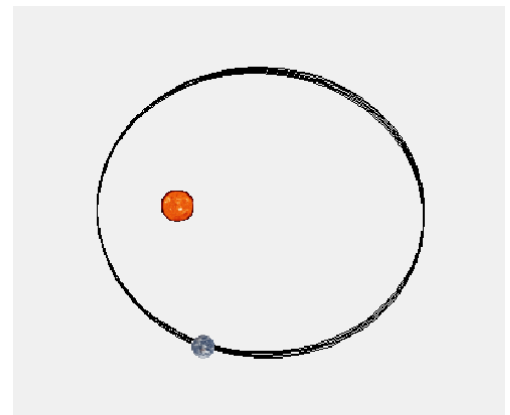
3. Wir schicken den Planeten irgendwohin auf der Arbeitsfläche, wechseln sein Kostüm und verpassen ihm die lokalen Variablen (Instanzvariablen) *vx* und *vy* als Geschwindigkeitskomponenten, *ax*, *ay* für die Beschleunigung und *r* für den Abstand zur Sonne. Dazu kopieren wir die beiden ersten Blöcke aus dem oberen Bild (Rechtsklick darauf) und ändern die Parameter. Dann wechseln wir ins *Variables*-Register und fügen fünf *Instance-Variables* hinzu. Man erkennt Instanzvariable am vorangestellten *„my“*. Deren Werte können wir mit dem danach auftauchenden Block *set my <variable> to <value>* bestimmen bzw. mit *increase my <variable> by <value>* ändern.



4. Befindet sich die Sonne im Ursprung des Koordinatensystems, dann erhält man die Gravitationskraft auf den Planeten zu $\boldsymbol{F} = -G * \frac{m*M}{r^3}*\boldsymbol{r}$ (Vektoren fett), also $\boldsymbol{a} = -G * \frac{M}{r^3}*\boldsymbol{r}$. Aus den beiden Kraftkomponenten berechnen wir die Beschleunigungskomponenten und aus diesen die Änderungen der Geschwindigkeitskomponenten und des Ortes. Alle Werte wurden so gewählt, dass die Bahnkurve auf den Bildschirm passt. Die mathematischen Ausdrücke werden mithilfe der Blöcke aus der *Operators*-Palette zusammengestellt.



5. Diese Anweisungsfolge muss jetzt immer wieder ausgeführt werden. Dazu verpacken wir sie in eine *animate*-(unendliche)-Schleife aus der *Control*-Palette. Der Schönheit halber löschen wir vorher alle eventuell vorhandenen Spuren auf der Arbeitsfläche und senken den eingebauten Stift des Planeten ab, damit wir seine neue Spur sehen. Den Pfeil (*arrow*) zum Planeten stellen wir mit dem Schalter oben-rechts auch ab. Dann stellen wir fest, dass wir nicht sonderlich genau gerechnet haben.

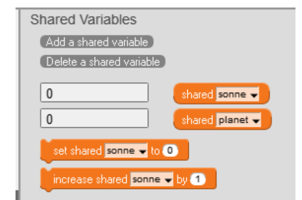


1.6.2 Zweiter Ansatz: Die Sonne lernt dazu

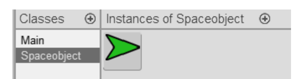
Wir benutzen zwei Klassen (*Main* und *Spaceobject*) und zeigen daran, wie auf *Attribute* und *Methoden* der Objekte auf traditionelle Weise zugegriffen wird[5] - also ähnlich wie in anderen objektorientierten Sprachen auch. Außerdem werden die üblichen *algorithmischen Grundstrukturen* sowie *globale* und *lokale Variable* sowie eine *Liste* verwandt. Es bleibt aber bei zwei Objekten, der Sonne und dem Planeten.

1. Wir gehen wie unter 1.6.1 – Punkt 1 vor und verfügen dann über zwei Bilder von Sonne und Planet.

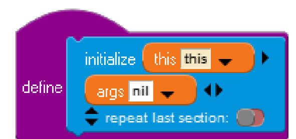
2. Wir erzeugen wie unter 1.6.1 – Punkt 3 geschildert zwei Variable namens *sonne* und *planet*, diesmal aber als *globale Variable*. Diese werden als *Shared Variables* bezeichnet und haben eigene Blöcke für Wertzuweisung und Veränderung.

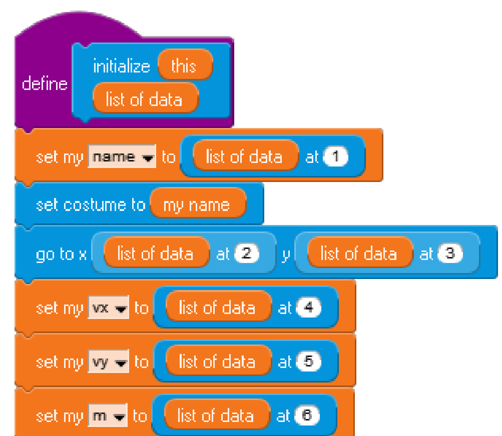


3. Unten in der Mitte des GP-Fensters befindet sich der Klassen-Bereich, in dem Klassen umbenannt, erzeugt, exportiert und gespeichert werden können. Anfangs gibt es nur eine Klasse namens *MyClass*. Zu dieser und anderen neu erzeugten Klassen gehört jeweils eine Instanz dieser Klasse, die im Arbeitsbereich als *ship* (grüner Pfeil) dargestellt wird. In unserem Fall taufen wir die MyClass-Klasse um in *Main* (*Rechtsklick auf den Klassennamen und das Kontextmenü benutzen*) und erzeugen eine neue Klasse *Spaceobjects*, indem wir auf den +-Button hinter *Classes* klicken. Wir wechseln zwischen den Klassen, indem wir deren Namen anklicken.



4. Im Skriptbereich der Spaceobject-Klasse wechseln wir zur *My Blocks* -Palette und erzeugen einen *Konstruktor* (*initialize method*). Das ist ein Skript, das immer dann ausgeführt wird, wenn ein Objekt (eine Instanz) einer Klasse neu erzeugt wird. Man benutzt Konstruktoren z. B. dazu, einem Objekt Anfangswerte zu verpassen: hier: ein Bild, Position, Anfangsgeschwindigkeiten, … Wir klicken also auf den Button *Make a initialize method* und erhalten den nebenstehenden Block *initialize*: einen *Hat*-Block einer Methode, der später eine Referenz auf das Objekt selbst (*this*) sowie eventuelle Parameter übergeben werden. Da wir eine ganze Menge Parameter übergeben wollen, führen wir die nicht alle einzeln auf, sondern geben sie in Form einer *Liste* an, z. B. als *[Name, x, y, vx, vy, Masse]*. Den Parameter taufen wir nach Anklicken um in *list of data*. Wenn entsprechende lokale Variable für die Klasse eingeführt wurden, werden im Konstruktor die Anfangswerte des Parameters in diese kopiert. Zugegriffen wird auf Listenelemente z. B. durch den Block *<list> at <index>*. Aus der *Data*-Palette. Dort finden sich auch reichlich weitere Listenoperationen.

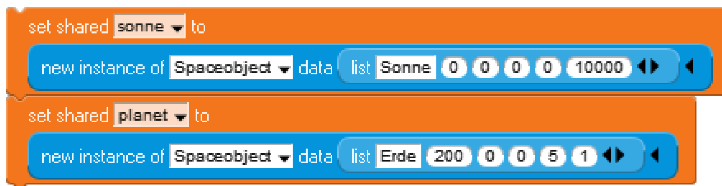




[5] Mit Botschaften arbeiten wir später.

5. Da unsere *Main*-Klasse die Abläufe steuern soll, schreiben wir dort zwei Blöcke, die Sonne und Planet durch Konstruktoraufrufe mit den entsprechenden Parameterlisten erzeugen. Das geschieht mithilfe des *new instance of <class>*-Blocks aus der *Structure*-Palette.

new instance of MyClass

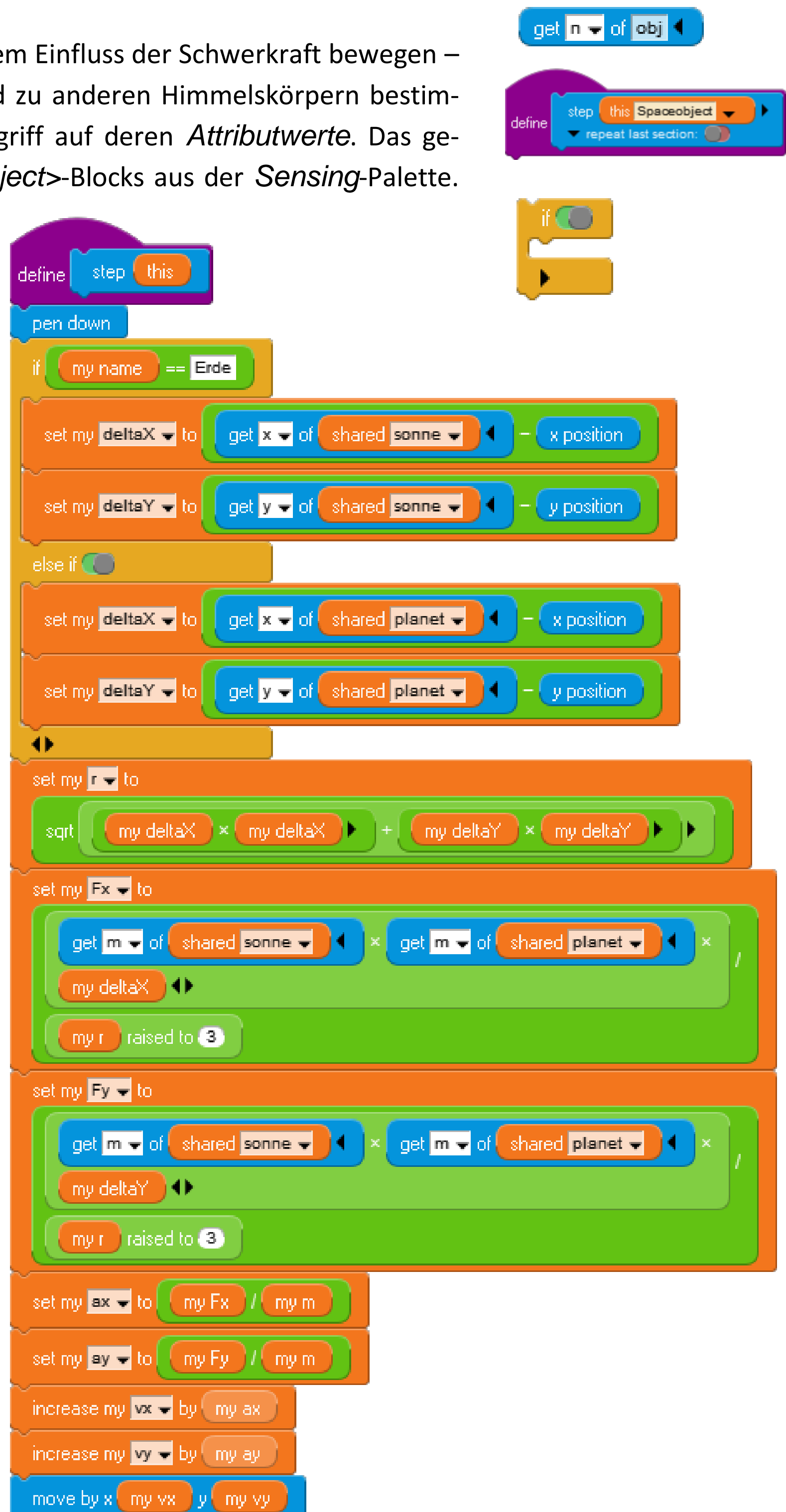


get n of obj

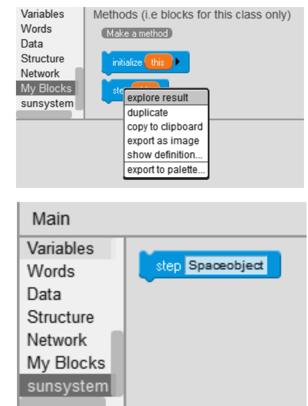
define step this Spaceobject repeat last section:

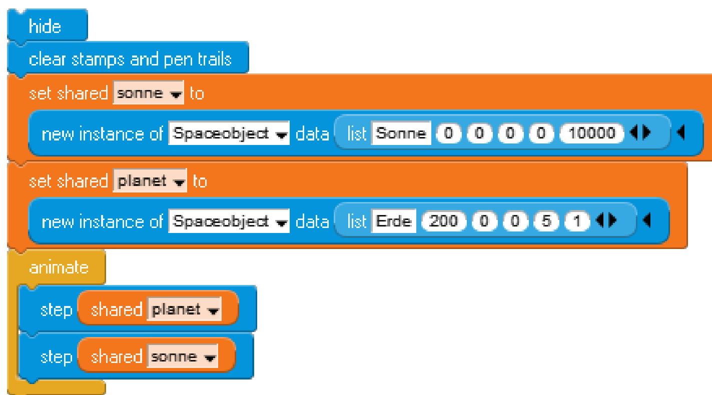
if

6. Unsere Himmelsobjekte sollen sich unter dem Einfluss der Schwerkraft bewegen – schrittweise. Dazu müssen sie den Abstand zu anderen Himmelskörpern bestimmen können. Sie benötigen also einen Zugriff auf deren *Attributwerte*. Das geschieht mithilfe des *get <attribut> of <object>*-Blocks aus der *Sensing*-Palette. Dort sind Attribute, über die alle Instanzen verfügen (*x* und *y*) aufgeführt. Andere Attribute (z. B. die Masse *m*) müssen „per Hand“ eingegeben werden. Wir statten unsere Spaceobject-Klasse mit einer Methode *step* aus, indem wir in der *My Blocks*-Palette der Klasse den Button *Make a method* betätigen – und erhalten den entsprechenden Methodenkopf, jetzt aber für eine „normale“ Methode. An diese docken wir die Blöcke zur Berechnung der beiden Kraftkomponenten an (siehe 1.6.1 – Punkt 4) – und entscheiden jeweils in einer Alternative (*if …*), welches der „andere“ Himmelskörper ist.



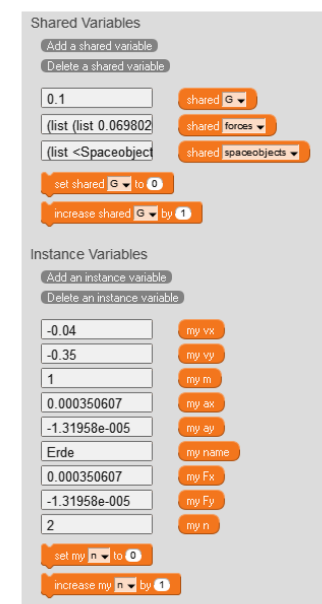
7. Die Steuerung aus der Main-Klasse ist jetzt einfach: hier werden einfach immer wieder die *step*-Methoden der beiden Himmelsobjekte aufgerufen. Die kann man aber „von außen“, also von einer anderen Klasse aus, gar nicht sehen. Ebenso wenig wie die lokalen Variablen. Diese Größen sind alle „*privat*“. Soll eine Methode in anderen Klassen sichtbar sein, dann müssen wir sie exportieren. Wir klicken mit der rechten Maustaste in der *My Blocks*-Palette oben auf den Methodenkopf und wählen aus dem aufklappenden Kontextmenü den Punkt *export to palette …* . Dann geben wir einen Namen an, z. B. „Solar System“. Ab jetzt ist diese Methode in allen Klassen verfügbar. Wir müssen dann natürlich immer angeben, für welches Objekt dieser Klasse sie aufgerufen werden soll: hier also entweder für die Sonne oder den Planeten. Das gesamte Skript unserer Main-Klasse lautet dann:





Wir erhalten dasselbe Bild wie im letzten Abschnitt – mit einem kleinen Unterschied: zieht man die Sonne von ihrem alten Platz, dann sieht man, dass sie sich auch minimal bewegt hat.

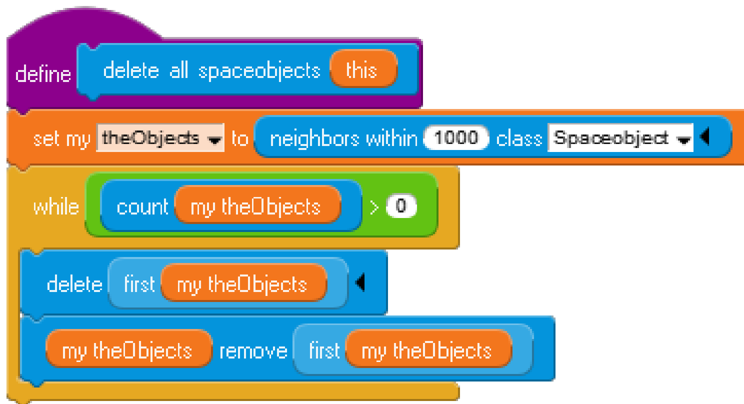
1.6.3 Dritter Ansatz: Jeder gegen Jeden



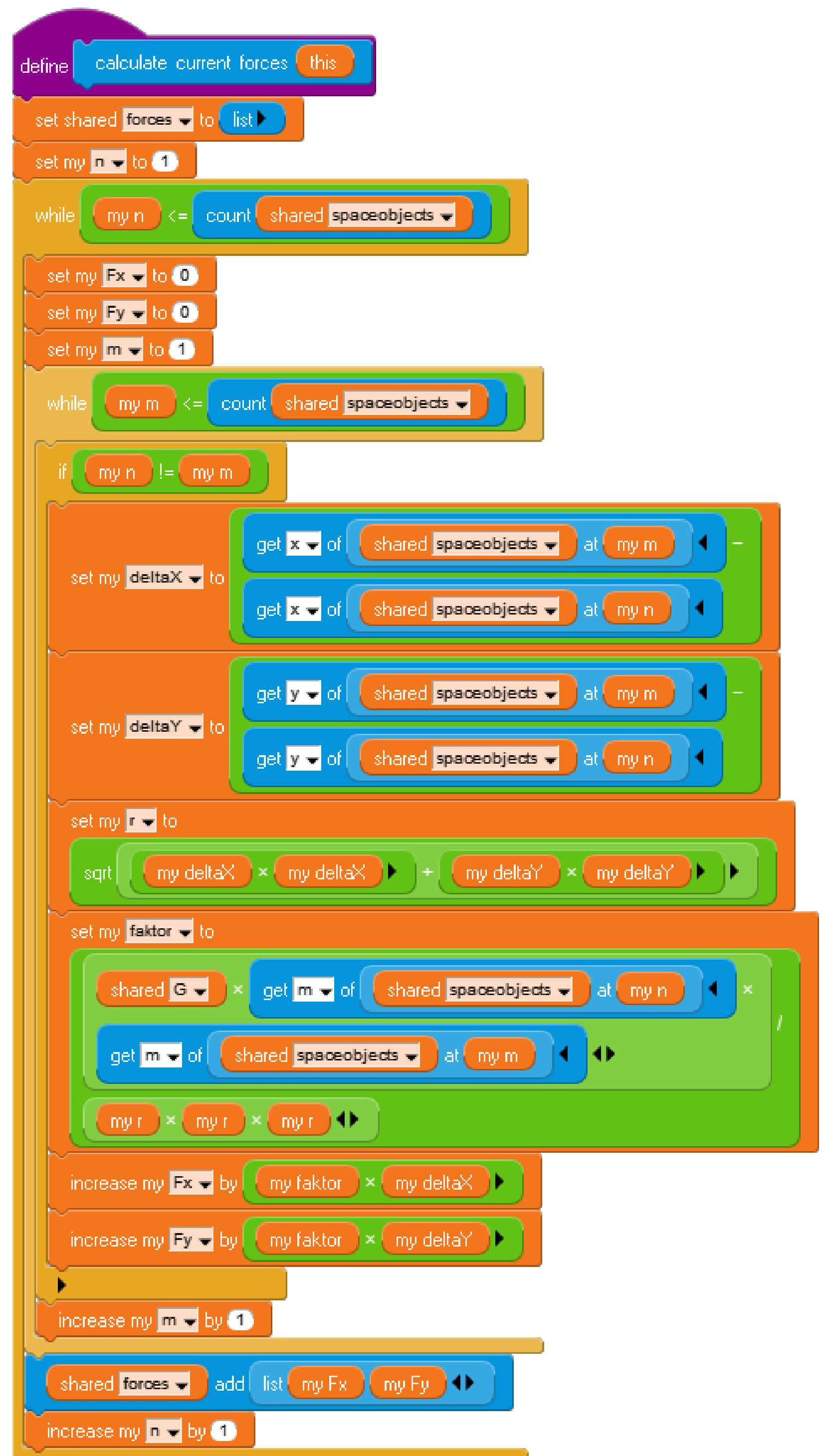
Zuletzt wollen wir mithilfe der jetzt bekannten Verfahren ein volles Sonnensystem mit beliebig vielen Himmelskörpern simulieren. Wir stecken diese in eine Liste namens *spaceobjects* und verpassen jedem Objekt eine Nummer, die seinem Index in dieser Liste entspricht. Auch die Berechnung der Kräfte soll jetzt „zum gleichen Zeitpunkt“ innerhalb der *Main*-Klasse erfolgen, damit sich nicht Objekte schon bewegt haben, bevor die anderen ihre Ortsänderungen berechnen konnten.

Zuerst einmal schaffen wir aber eine Möglichkeit, alle schon vorhandenen Himmelsobjekte zu löschen, bevor wir neue erschaffen. Die Reste aus vorherigen Programmläufen haben uns nämlich geärgert.

Wir vereinbaren eine Instanzvariable *theObjects* (s.o.) und erzeugen eine Klassenmethode in der *My Blocks*-Palette. Dann weisen wir *theObjects* eine Liste aller Objekte der Klasse *Spaceobject*, also der vorhandenen Himmelskörper, im Umkreis von 1000 Pixeln zu. Solange in dieser Liste noch Elemente vorhanden sind, löschen wir das erste und danach auch die Referenz auf dieses aus der Liste. Die erforderliche Kontrollstruktur *while*-Schleife finden wir in der *Control*-, die Listenoperationen *count*, *first* und *remove* in der *Data*-Palette. *Delete <object>* stammt aus der *Structure*-Palette.

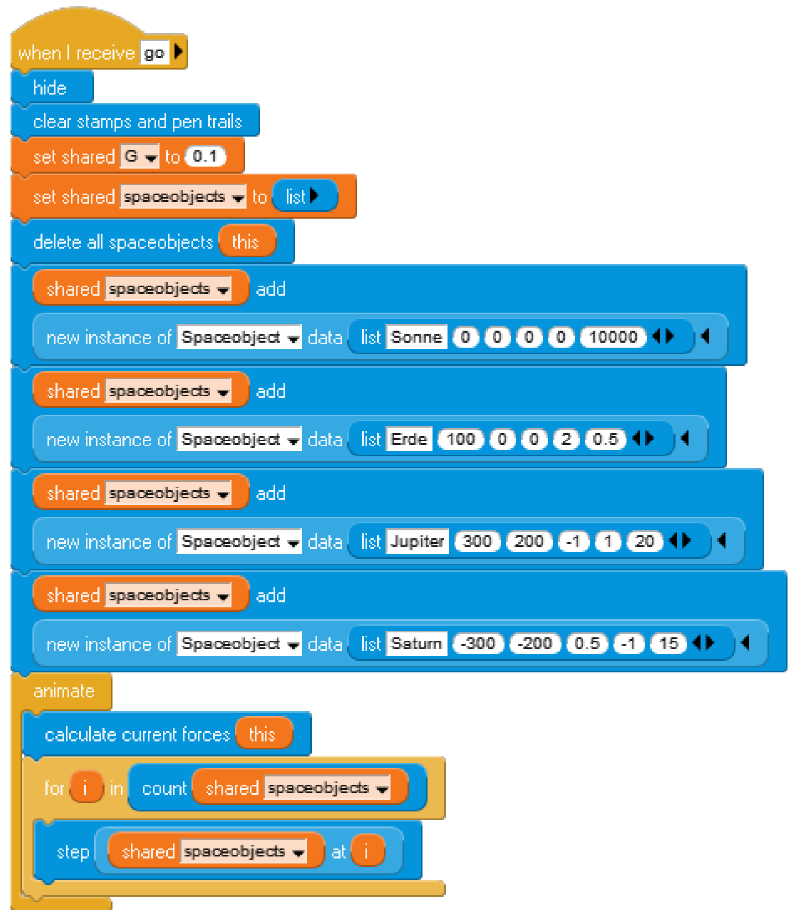


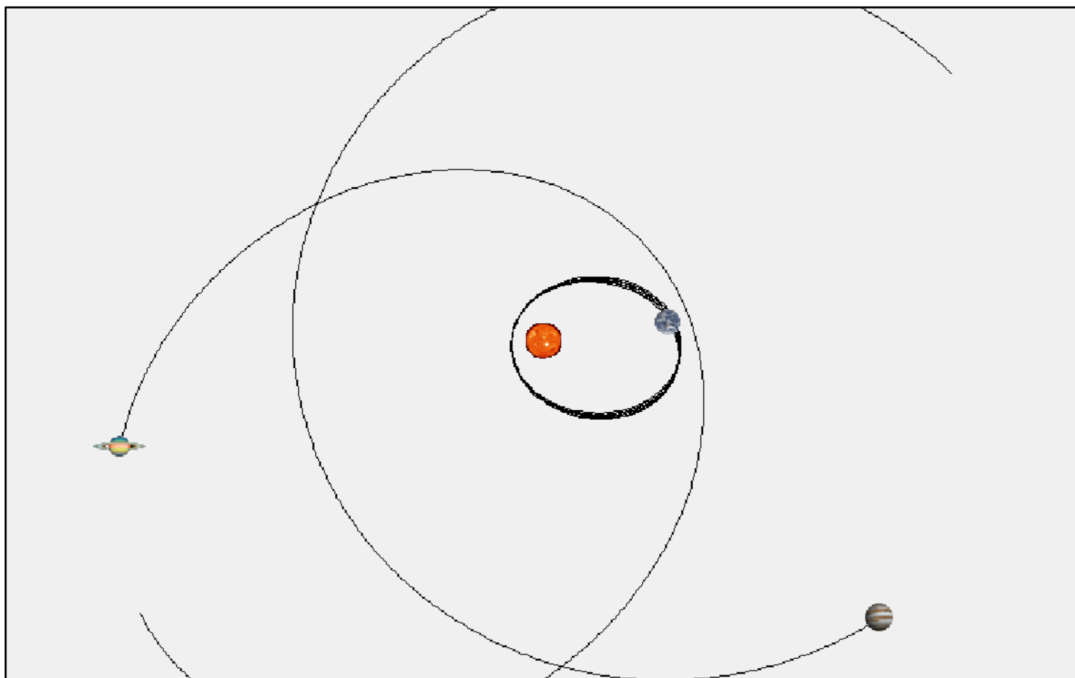
Die zweite Methode der Main-Klasse dient zur Berechnung der Kräfte. Sie löscht zuerst die vorhandenen Werte. Danach berechnet sie die Kräfte zwischen allen Himmelskörpern (*while n …*) und allen anderen (*while m …*). Nur wenn die Körper verschieden sind greift sie auf die Attribute der Objekte mit *get <attribut> of <object>* zu und berechnet daraus die Kraftkomponenten. Diese werden summiert. Nach jedem Durchlauf werden die Ergebnisse in die Liste *forces* eingetragen.



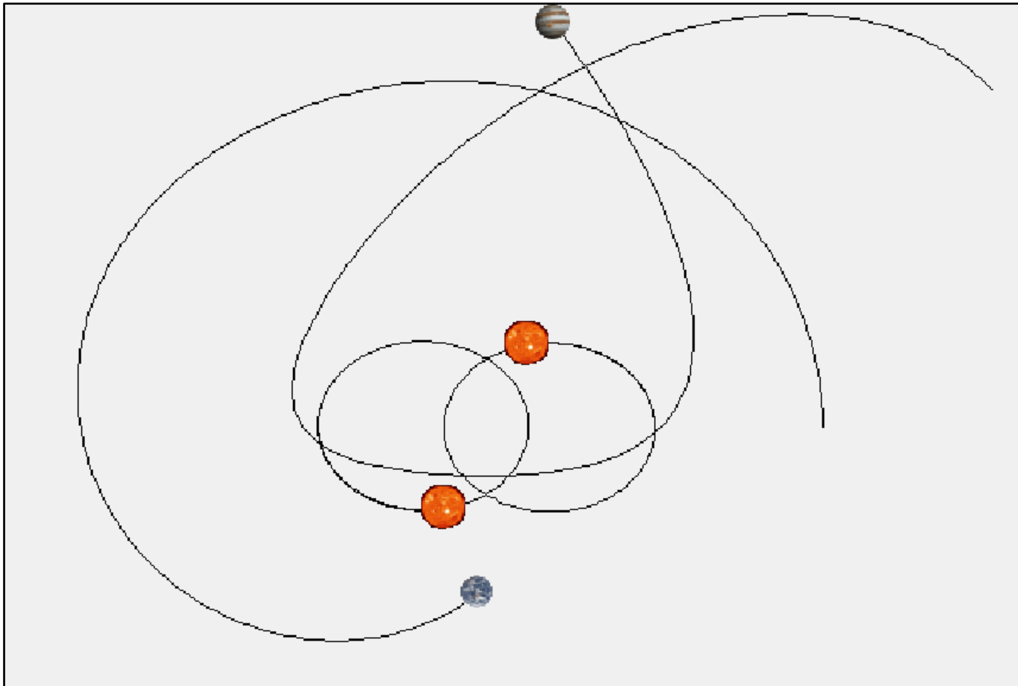
Die eigentliche Arbeit erfolgt mithilfe dieser Methoden in einer *Ereignisbehandlungsmethode*, die auf das Senden der Nachricht *go* reagiert. In dieser wird nach Löschen des Bildschirms die Gravitationskonstante auf 0.1 gesetzt. Danach werden alle eventuell vorhandenen Himmelskörper gelöscht und eine neue Sonne sowie drei Planeten erzeugt. Das geschieht mithilfe des Blocks *new instance of <class> data …* aus der *Structure*-Palette. Dieser ruft, wenn vorhanden, den entsprechenden Konstruktor auf.

Anschließend werden in einer Endlosschleife (*animate*) jeweils die neuen Kräfte zwischen den Objekten berechnet. Danach wird für alle vorhandenen Objekte die *step*-Methode aufgerufen (und so ausgeführt). Das Ergebnis sind die (nicht sehr genauen) Bahnbewegungen.

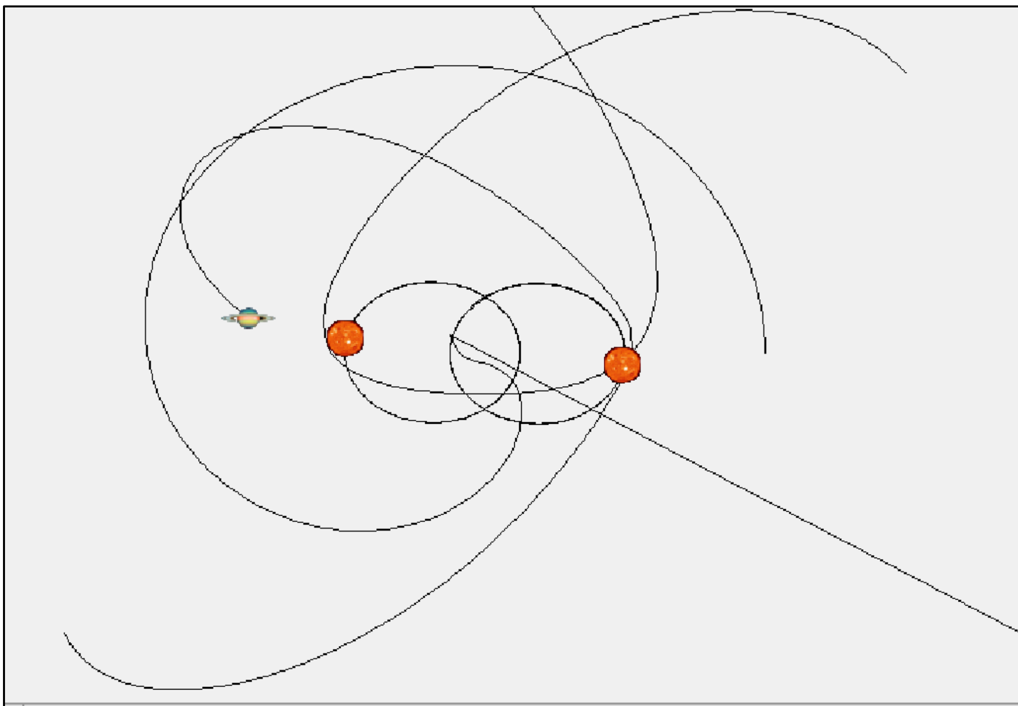




Man kann aber auch z. B. mit Doppelsternsystemen experimentieren und erfahren, dass es die Planeten in solchen Systemen nicht ganz leicht haben.



Man kann verstehen, wie „querschießende“ Planeten in ziemlich ungewöhnliche Situationen geraten können.



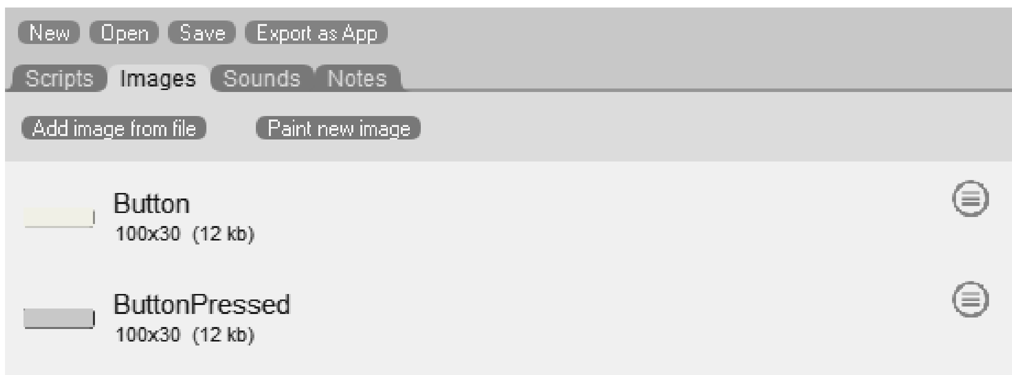
2 Ein Zeichenprogramm

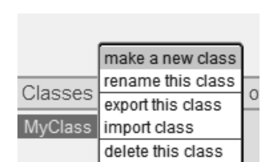
Wir beginnen mit einem kleinen Beispiel, das die Ereignisverarbeitung und den direkten Umgang mit einfachen Skripten demonstrieren soll. Mithilfe von Knöpfen sollen verschiedene Funktionen eingestellt werden können, die von verschiedenen Arbeitsgruppen realisiert werden. Das Beispiel stellt Möglichkeiten zu arbeitsteiligem Vorgehen sowie zur Fehlersuche vor, an die sich die Lernenden früh gewöhnen sollten.

2.1 Mit Knöpfen arbeiten

Buttons erzeugen

Als erstes zeichnen wir in einem gängigen Zeichenprogramm einen Button, schneiden ihn aus und fügen ihn in ein neues Bild genau seiner Größe ein. Dann füllen wir ihn mit einer Farbe, die den *Gedrückt*-Zustand anzeigen soll, speichern das Bild als PNG-Datei *ButtonPressed.png*, färben den Button etwas anders ein und speichern das geänderte Bild des nun symbolisierten *Nichtgedrückt*-Zustands als *Button.png*. Diese beiden Bilder laden wir als Image-Dateien in GP.





Danach ändern wir den Namen der Klasse *MyClass* in *Button* (nach Rechtsklick auf den Klassennamen). Das standardmäßig anfangs zugewiesene grüne Pfeilkostüm der noch einzigen Instanz soll zu einem Button-Kostüm werden. Also weisen wir ihm die richtige Größe zu, in unserem Fall 100 x 30 Pixel, zeichnen das Button-Bild und verkleinern es gleich wieder, weil uns diese Buttongröße denn doch etwas übertrieben erscheint. Das alles geschieht durch eine Sequenz von drei Blöcken, die wir in den *Looks*- und *Drawing*-Paletten finden.

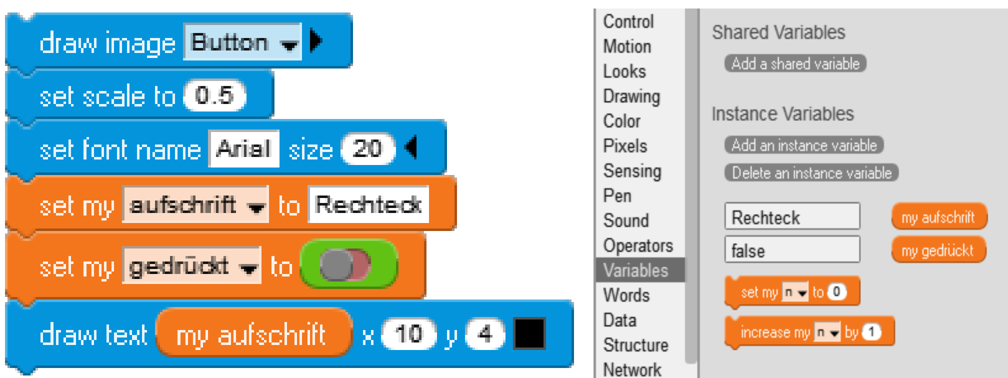
Diese Blocksequenz müssen wir nur anklicken, sie wird dann ausgeführt. (Dauert das lange genug, dann sehen wir den grünen Rand um laufende Skripte.)

Unser Block benötigt jetzt noch eine schöne Aufschrift, z. B. „*Rechteck*“. Das können wir durch Ausführen des *draw-text*-Blocks erreichen, der den angegebenen Text ab

den angegebenen Koordinaten *x* und *y*, die der linken-oberen Ecke des Textes entsprechen, in der aktuellen Schriftart auf das Sprite-Kostüm schreibt. Durch Ausprobieren finden wir die richtige Einstellung im *set-font-name*-Block, den wir durch Anklicken des kleinen schwarzen Pfeils rechts so erweitern, dass die Schriftgröße eingegeben werden kann. Beide Blöcke finden wir in der *Drawing*-Palette.

Wahrheitswerte als „Schalter"

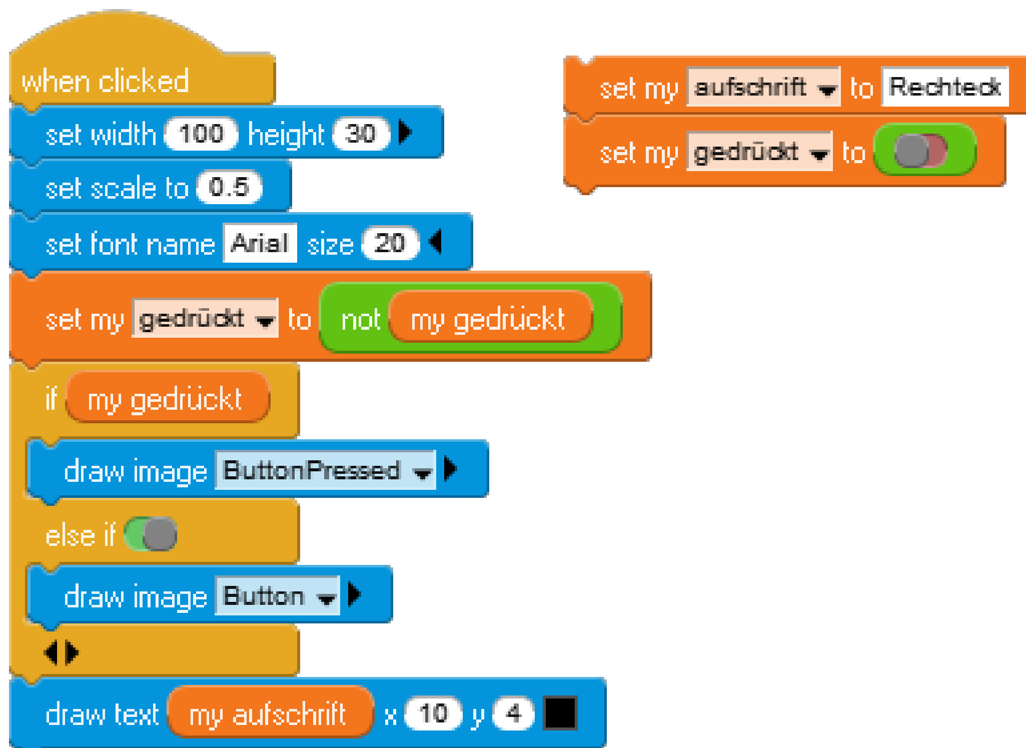
Unser Knopf soll sein Erscheinungsbild ändern, wenn er angeklickt wird. Dazu muss man wissen, ob er schon angeklickt wurde. Wir speichern diese Information in der Variablen *gedrückt*. Da wir später mehrere Knöpfe verwenden wollen, die alle einzeln anklickbar sind, wählen wir eine *Instanzvariable (Instance Variable*), die für jede Instanz anders aussehen kann und so deren spezifischen Zustand beschreibt. Als Inhalt wählen wir einen *Wahrheitswert* (*true* oder *false*), der in GP durch einen Schalter symbolisiert wird. Da sich auch die Aufschriften der Knöpfe unterscheiden werden, führen wir dafür auch gleich eine Instanzvariable *aufschrift* ein. Beides erreichen wir in der Palette *Variables*. Bei der Darstellung werden Instanzvariable durch ein vorangestelltes „*my*" gekennzeichnet.



auf Mausklicks reagieren

In der Control-Palette finden wir den *Hat-Block* („Hut" über einem Skript) *when-clicked*, der aufgerufen wird, wenn das entsprechende Element angeklickt wird. Das Ereignis „angeklickt" wird dadurch „behandelt". Docken wir an den Hat-Block die schon entwickelte Befehlsfolge an, dann wird diese immer ausgeführt, sobald der Button angeklickt wurde – immer gleich. Wir wollen aber einen Kostümwechsel! Den erreichen wir durch Auswerten der *gedrückt*-Variablen. Ist diese *false*, dann ist der Button noch nicht gedrückt und sollte also (ab jetzt) als gedrückt dargestellt werden; ist er *true*, dann wird das andere Kostüm gewählt. Nach jedem Klick wird der Wahrheitswert „umgeschaltet".

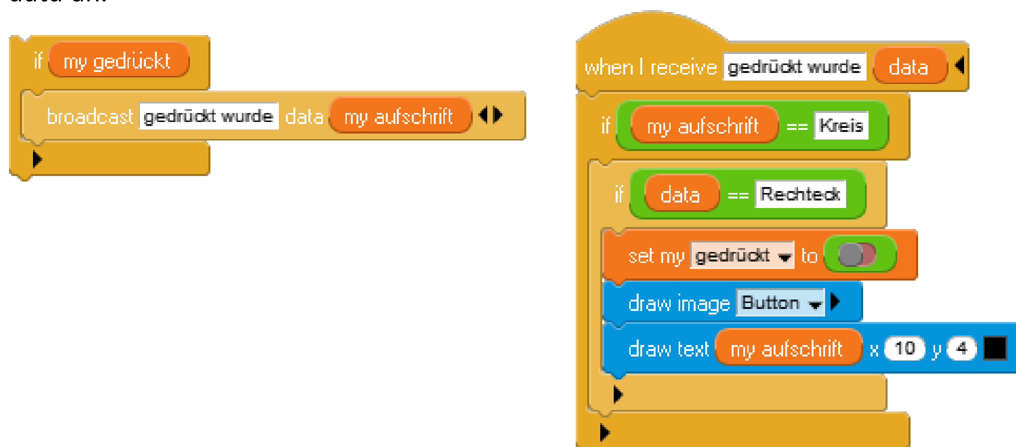
wenn der Knopf angeklickt wurde	
Ändere den Zustand der „gedrückt"-Variablen	
gedrückt?	
true	false
Wähle das Kostüm „Button"	Wähle das Kostüm „ButtonPressed"



Das funktioniert ganz prima. Deshalb erzeugen wir drei Klone unseres Buttons, indem wir dreimal den +-Button über dem Instanzenbereich anklicken. Danach wählen wir die drei neuen Instanzen aus und ändern jeweils die Aufschrift. Nach ihrem Anklicken haben sie das richtige Aussehen mit der gewählten Aufschrift: **Sie führen alle dasselbe Skript ihrer Klasse aus – allerdings mit unterschiedlichen Werten der lokalen Instanzvariablen.**

Botschaften einsetzen

Leider ergibt sich das Problem, dass eigentlich immer nur ein Knopf einer Gruppe ähnlicher Funktionalität gedrückt sein sollte. Wir können das z. B. erreichen, indem die Knöpfe beim Drücken *Botschaften* aussenden, auf die andere Elemente von GP reagieren. (Auf diese Weise werden wir weiter unten auch das Zeichnen realisieren.) An die Botschaft *„gedrückt wurde“* hängen wir zusätzlich die Aufschrift des Knopfes als *data* an.



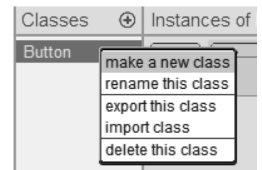
Hier sollen erstmal eventuell aktive Knöpfe „abgeschaltet“ werden, wenn ein ähnlicher Knopf gedrückt wird. Erhält z. B. der „Kreis“-Knopf die Nachricht, dass der „Rechteck“-Knopf gerade aktiviert wurde, dann geht er in den „Ungedrückt“-Zustand über. Die vollständigen Skripte, die alle vier Knöpfe berücksichtigen, lauten:

```
when clicked
set width 100 height 30
set scale to 0.5
set font name Arial size 20
set my gedrückt to not my gedrückt
if my gedrückt
  draw image ButtonPressed
else if
  draw image Button
draw text my aufschrift x 10 y 4
if my gedrückt
  broadcast gedrückt wurde data my aufschrift
```

```
when I receive gedrückt wurde data
if my aufschrift == Kreis
  if data == Rechteck
    set my gedrückt to
if my aufschrift == Rechteck
  if data == Kreis
    set my gedrückt to
if my aufschrift == schwarz
  if data == rot
    set my gedrückt to
if my aufschrift == rot
  if data == schwarz
    set my gedrückt to
draw image Button
draw text my aufschrift x 10 y 4
```

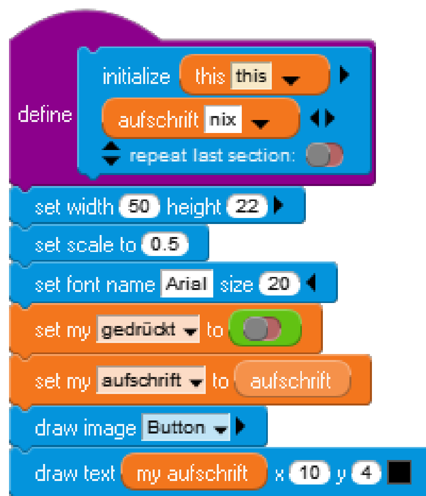
2.2 Klassen exportieren und importieren

Da wir jetzt so wunderbare Knopfgruppen erzeugen können, wollen wir diese Funktionalität auch anderen zur Verfügung stellen. Im einfachsten Fall exportieren wir einfach die Klasse *Button*, indem wir rechts auf die Klasse klicken und den dritten Eintrag im Kontextmenü wählen. Andere können diese Klasse dann auf diesem Wege importieren und nutzen. Skriptblöcke kann man auch einfach über die Zwischenablage transportieren.

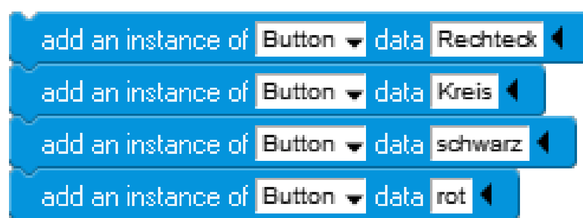


Einen Konstruktor schreiben

Etwas komfortabler ist es, der Klasse eine *Initialisierungsmethode* (einen *Konstruktor*) zu spendieren, die ausgeführt wird, wenn eine neue Instanz der Klasse erzeugt wird. Wir finden diese Möglichkeit unter *My Blocks*. So einem Konstruktor können wir Daten als *Parameter* übergeben, die in die Instanzvariablen eingetragen werden können. In unserem Fall wollen wir als Vorgabewert die Aufschrift des Buttons als „nix" setzen. Diese kann dann als Übergabeparameter beim Aufruf des Konstruktors geändert werden.



Wir können diese Funktionalität wieder über die Klasse Button exportieren und in ein anderes Projekt importieren. Dort lassen sich die angegebenen vier Knöpfe dann schnell durch vier Anweisungen aus der *Structure*-Palette erzeugen.



Kommentare einfügen

Ein gutes Zeichenprogramm ist ziemlich umfangreich, wenn wir uns nicht nur auf zwei Farben und zwei Formen beschränken. Wir sollten deshalb arbeitsteilig in verschiedenen Gruppen arbeiten und die Ergebnisse später zusammenstellen. Da die Skripte später von anderen gelesen und verstanden werden müssen, ist es gute Praxis, *Kommentare* in die Skripte einzufügen und, hier unter *Notes*, Anmerkungen zum Programm zu machen. Wir finden den Kommentar-Block in der *Control*-Palette.

2.3 Aufgaben

Um die folgenden Aufgabe zu bearbeiten, müssen Sie sich etwas in den Befehlsregistern umsehen. Experimentieren Sie mit den Blöcken!

1. Es wäre doch ganz nett, wenn die Knöpfe gleich an der richtigen Stelle erscheinen.
   a: Führen Sie zwei Instanzvariable *x* und *y* ein, deren Werte Sie „per Hand“ setzen können. Der Button soll dann natürlich auch zur richtigen Stelle „springen“.
   b: Initialisieren Sie *x* und *y* gleich im Konstruktor.
   (Tipp: Man muss dazu wahrscheinlich mehrere Daten „zusammenfassen.
   Sehen Sie mal in der Data-Palette nach. 😉)

2. Ab und zu verschiebt man einen Knopf versehentlich. Der sollte dann sofort an die richtige Stelle zurückspringen. Realisieren Sie diese Funktionalität.
   (Tipp: Benutzen Sie die Suchmöglichkeit für Befehle oben-rechts über dem Befehlsregister. Ein geeigneter Befehl beginnt mit „wh“. 😉)

3. Besonders „cool“ sehen unsere Knöpfe mit ihren Grautönen ja nicht aus. Sorgen Sie für bessere Farben. Ändern Sie möglichst auch den Konstruktor entsprechend.

4. Bei kurzen Aufschriften sind unsere Knöpfe eigentlich zu breit, bei langen zu schmal. Sorgen Sie dafür, dass sich die Knopfbreite der Aufschrift anpasst.
   (Tipp: Ein geeigneter Befehl findet sich in der Drawing-Palette. 😉)

5. Bei Aufgabe 4 taucht eventuell das Problem auf, dass Sie die Buttonbilder mit Rand gezeichnet haben. Der passt dann nicht so recht zu den veränderten Breiten. Schreiben Sie eine Methode „*zeigeDich*“ der Button-Klasse, die den Hintergrund der Knöpfe selbst zeichnet. Ändern Sie die Ereignisbehandlung („when clicked“, …) entsprechend.
   (Tipp: Geeignete Befehle finden sich in der Drawing-Palette. 😉)

2.4 Einfache Algorithmik und Fehlersuche

Die Knöpfe müssen mit Funktionalität versehen werden. Wir beschränken uns dabei zuerst auf die Rechtecke, bauen dabei die Ereignisverarbeitung aus und demonstrieren den Umgang mit einfachen Skripten. Weiterhin werden verschiedene Möglichkeiten zur Fehlersuche vorgestellt, an die sich die Lernenden früh gewöhnen sollten.

Als Zeichenfläche wählen wir ein *Panel*, dem wir eine geeignete Größe und Hintergrundfarbe zuweisen. Einfach geht das, wenn wir die vorgegebene Klasse *MyClass* in *Panel* umbenennen.

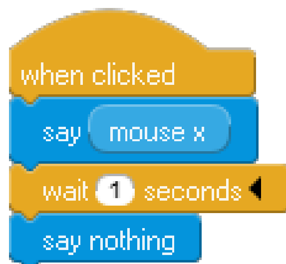
2.4.1 Rechtecke zeichnen

Unser Ziel soll es hier sein, mit der Maus zweimal auf den Bildschirm zu klicken, sodass danach ein Objekt zwischen den angeklickten Punkten gezeichnet werden kann. Dazu sollen zunächst einmal die Koordinaten des angeklickten Punkts angezeigt werden.

Wir klicken auf das Panel - nichts passiert.

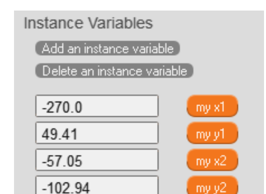
Der Grund ist einfach: niemand beachtet diese Mausklicks. Wir benötigen deshalb eine *Ereignisbehandlung* aus der Rubrik *Control*, die dafür sorgt, dass fortlaufend nachgesehen wird, ob jemand im Panel mit der Maus klickt. Geschieht das, dann wollen wir die *x*-Koordinate der Maus für eine Sekunde anzeigen lassen.

Ereignisbehandlung

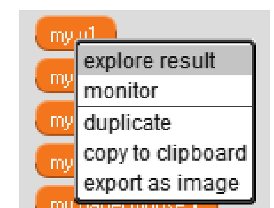


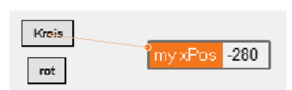
Ein Hat-Block zum Starten eines Skripts

Wollen wir Rechtecke zeichnen, dann benötigen wir die Koordinaten von zwei Punkten, also vier Zahlenwerte. Dazu müssen wir diese Werte zum richtigen Zeitpunkt speichern – wenn die Maustaste gedrückt wird. Geeignet dafür sind *Instanzvariable*, die wir in der Variables-Rubrik erzeugen und bearbeiten. Wir wollen diese Variable in der üblichen Art mit x1, y1, x2 und y2 bezeichnen. Dazu klicken wir den Knopf mit der Beschriftung *Add an instance variable* an und geben im erscheinenden Eingabefeld den richtigen Namen ein.



Wir erzeugen jetzt die vier Variablen auf diesem Wege und klicken bei Bedarf mit der rechten Maustaste auf den Variablennamen, um die Variablenwerte im Bildbereich mithilfe eines *Monitors* anzuzeigen. Ist die Pfeilanzeige (*show arrows*) eingeschaltet, dann zeigt der Monitor auf das Element, auf das sich sein Wert bezieht. Monitore werden gelöscht, indem sie wieder aus dem Ausgabebereich geschoben werden.





Jetzt müssen wir uns nur noch merken, welcher Eckpunkt beim Mausklick gespeichert werden soll – der erste oder der zweite. Wir führen dafür eine neue Variable ersterPunkt ein, der wir abwechselnd die Wahrheitswerte true oder false[6] zuweisen, die wir in der Operators-Rubrik finden. Anfangs erhält sie den Wert true, indem wir den entsprechenden Zuweisungsblock set my <variable> to <value> benutzen[7].

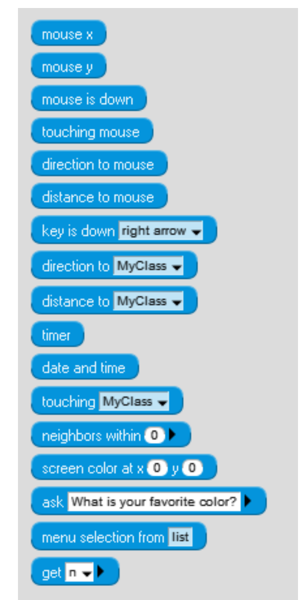
Damit können wir das Vorgehen verfeinern:

<table>
<tr><td colspan="2">Wenn auf das Panel geklickt wurde</td></tr>
<tr><td colspan="2">ersterPunkt?
wahr | falsch</td></tr>
<tr><td>x1 ← x-Koordinate der Maus</td><td>x2 ← x-Koordinate der Maus</td></tr>
<tr><td>y1 ← y-Koordinate der Maus</td><td>y2 ← y-Koordinate der Maus</td></tr>
<tr><td rowspan="2">ersterPunkt ← falsch</td><td>Zeichne ein Rechteck mit den Eckpunkten (x1|y1) und (x2|y2)</td></tr>
<tr><td>ersterPunkt ← wahr</td></tr>
</table>

Operators-Palette

Die Mausposition finden wir in der *Sensing*-Palette – allerdings bezogen auf das Ausgabefenster, nicht auf das Panel. Wir müssen also die Koordinaten noch umrechnen.

Sensing-Palette



[6] Die Wahrheitswerte werden durch Schalter symbolisiert, können aber auch direkt eingegeben werden.
[7] Die spitzen Klammern kennzeichnen die Parameter, also Werte, die beim Aufruf des Blocks angegeben werden müssen.

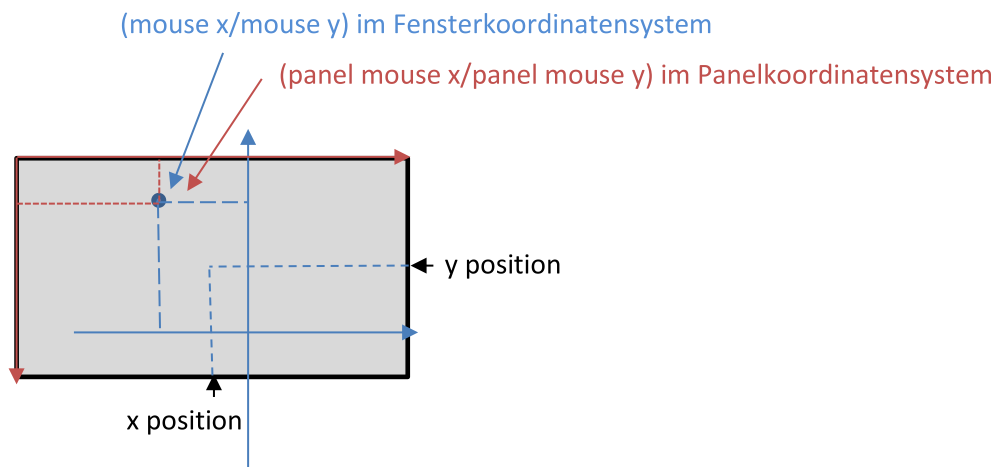
Eine eigene Methode schreiben

zeichne Rechteck this

Bei *zeichne Rechteck* handelt es sich um keine Ereignisbehandlungsmethode, sondern um eine Methode, einen neuen **selbst geschriebenen Befehl**, der Klasse *Panel*, die wir mithilfe der Rubrik *My Blocks* erstellen. Aufgerufen werden eigene Methoden wie die anderen Befehle: man klickt den Block an oder fügt ihn in eine Blocksequenz ein. Da es sich um eine Klassenmethode handelt, muss angegeben werden, welche Instanz dieser Klasse den Befehl ausführen soll – mit ihren eigenen Instanzvariablen. Diese Instanz kann in einer Variablen gespeichert sein oder es kann sich um die gerade aktive handeln. Diese wird mit this bezeichnet.

Koordinaten umrechnen

Mit *mouse x* und *mouse y* erhalten wir die Mauskoordinaten im Koordinatensystem des Ausgabefensters, das seinen Mittelpunkt in der Fenstermitte hat und dessen y-Achse nach oben gerichtet ist. Im unteren Bild ist dieses Koordinatensystem blau gezeichnet. In diesem Koordinatensystem hat das Panel die Koordinaten *x position* und *y position*, die die Mitte des Panels beschreiben. Gezeichnet wird aber im Panel-Koordinatensystem, dessen Nullpunkt die obere-linke Ecke des Panels bildet, wobei die y-Achse nach unten gerichtet ist. Dessen Achsen sind in Rot eingetragen. *width* und *height* ergeben die Maße des Panels.



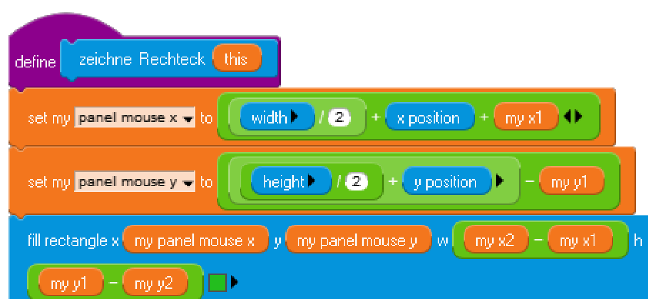
Daraus lassen sich die Panel-Koordinaten des Mausklicks bestimmen:

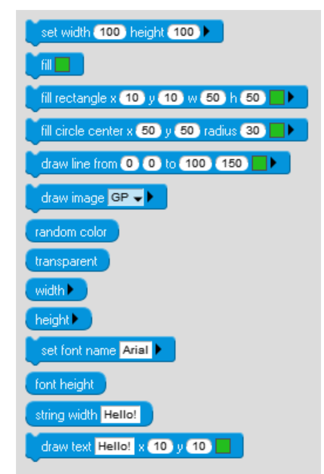
$$panel\ mouse\ x = \frac{width}{2} + x\ position + mouse\ x$$

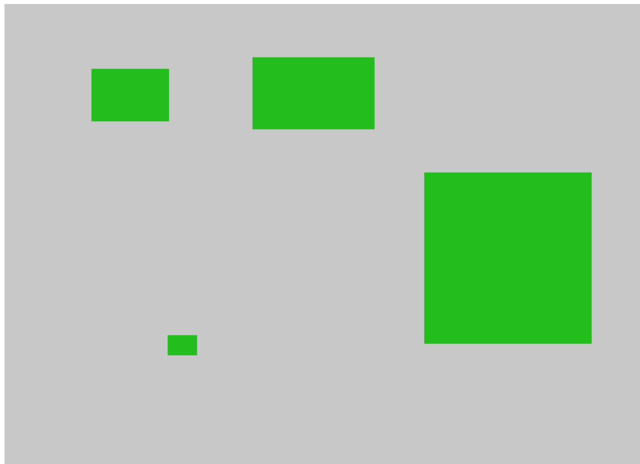
$$panel\ mouse\ y = \frac{height}{2} + y\ position - mouse\ y$$

Drawing-Palette

Damit erhalten wir unsere *zeichne Rechteck*-Methode für grüne Rechtecke zu:

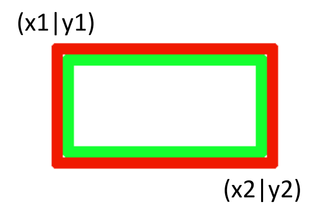






Einfache Rechtecke

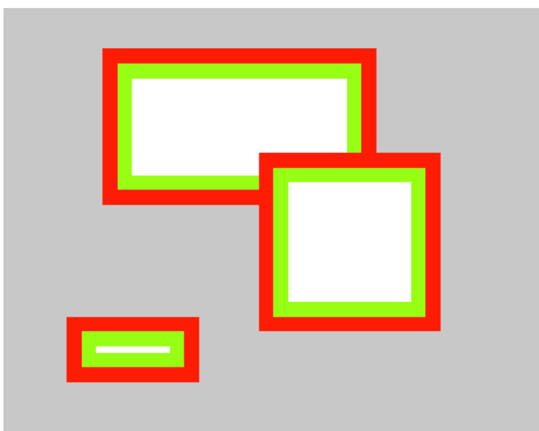
Mittlerweile ist es uns also gelungen, die Koordinaten von zwei Bildschirmpunkten (x1|y1) und (x2|y2) durch Mausklicks zu bestimmen. Zwischen denen wollen wir dann natürlich ein besonders schönes Rechteck zeichnen, das einen breiten roten Rand haben soll und darin einen weiteren grünen. Der Innenraum bleibt weiß.



Der Rahmen ist schnell erstellt: es werden ineinander drei Rechtecke in den Farben Rot, Grün und Weiß gezeichnet, wobei die Eckpunkte jeweils um 10 Pixel „nach innen“ gerückt werden.

```
define zeichne Rechteck this
set my panel mouse x to (width / 2) + x position + my x1
set my panel mouse y to (height / 2) + y position - my y1
fill rectangle x my panel mouse x y my panel mouse y w (my x2 - my x1) h (my y1 - my y2) [rot]
fill rectangle x (my panel mouse x + 10) y (my panel mouse y + 10) w ((my x2 - my x1) - 20) h ((my y1 - my y2) - 20) [grün]
fill rectangle x (my panel mouse x + 20) y (my panel mouse y + 20) w ((my x2 - my x1) - 40) h ((my y1 - my y2) - 40) [weiß]
```

Das Ergebnis ist wie erwartet - erstmal.



schöne Rechtecke

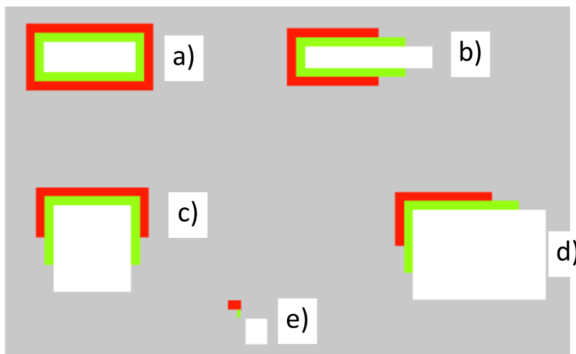
2.4.2 Fehler finden

Skripte systematisch testen

Man sollte sich nicht zu früh freuen, wenn ein Programm einmal gelaufen ist, sondern die Skripte systematisch testen. Viele Fehler zeigen sich erst, wenn das Programm mit unterschiedlichen Werten gestartet wird. In unserem Fall gingen wir davon aus, dass die Benutzer die Eckpunkte des Rechtecks in der genannten Reihenfolge anklicken: erst links-oben, dann rechts-unten. Das müssen sie aber natürlich nicht. Was passiert eigentlich, wenn sie anders klicken? Wir probieren systematisch die unterschiedlichen Möglichkeiten durch. Dazu klicken wir Punkte an, die

a) oben-links und unten-rechts
b) oben-rechts und unten-links
c) unten-links und oben-rechts
d) unten-rechts und oben-links
e) sehr nahe beieinander

im Rechteck liegen. Die Ergebnisse sind abgebildet. Sie entsprechen nicht so richtig unseren Vorstellungen.



Obwohl unser Programm richtig arbeitet, wenn es wie vorgesehen benutzt wird, könnten andere Benutzer/innen es auch anders verwenden – und dann passieren die gut sichtbaren Fehler, die hier natürlich mit der Berechnung der Eckpunkte der „inneren“ grünen und weißen Bereiche zu tun haben. Die muss also dringend verbessert werden.

Verhalten sich Programme nicht wie gewünscht, dann ist es oft schwierig, die Ursache dafür zu finden. Es gibt aber Hilfen:

- Mithilfe eingefügter Pausen (*wait*-Blöcke) kann der Ablauf so verlangsamt werden, dass sich z. B. die Werte der Variablen in der Variables-Palette (wo sie ja angezeigt werden) oder in *Monitoren* (s. o.) verfolgen lassen.

  wait 3 seconds

- Das Programm kann an kritischen Punkten gestoppt werden.

  stop

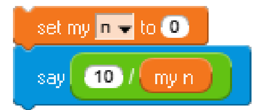
- Testausgaben können direkt ausgegeben werden.

  say bin jetzt innerhalb der Schleife

- Ausgaben im Terminalfenster können berücksichtigt werden. Im Developer-Modus können auch Ausgaben darin erfolgen. In diesem Modus stehen noch weitere Kontrollmöglichkeiten zur Verfügung.

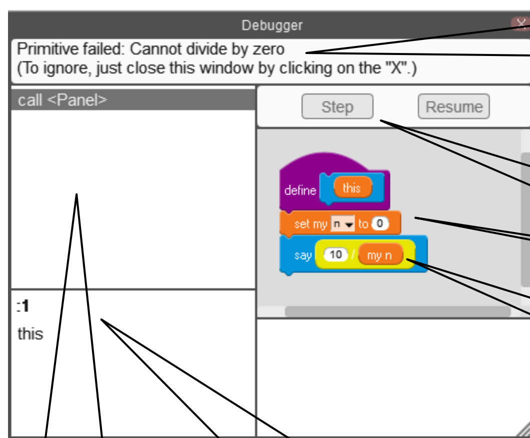
  console print Testing 1, 2, 3

```
 inspect (list 1 2 3) - inspect an ob
 go - restart the user interface
Cannot divide by zero
Testing 1, 2, 3
```

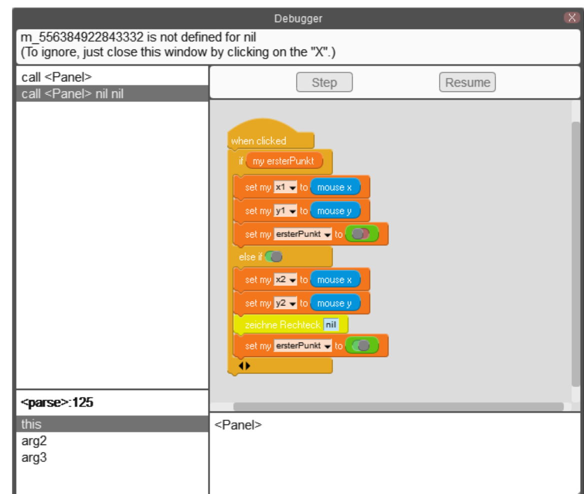


In vielen Fällen öffnet sich das Debugger-Fenster, in dem das fehlerhafte Skript angezeigt und der den Fehler verursachende Block gelb hervorgehoben sind. In diesem Fall wird der sehr einfache Fall gezeigt, in dem durch einen Variablenwert geteilt wird, der momentan Null ist.

Im Debugger



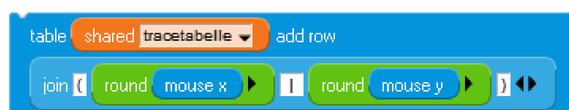
Hier noch ein etwas komplizierterer Fall: es fehlt eine Referenz zur aktuellen Instanz (in diesem Fall: *this*). Der Wert *nil* zeigt das an.



Trace-Tabellen

Eine gute Möglichkeit Programme zu überprüfen, ist der Einsatz von *Tracetabellen*. Dafür wird tabellarisch notiert, welche Variablen ihre Werte ändern. In der Tabelle verläuft die Zeitachse dann von oben nach unten. Ist GP im Developer-Modus, dann können wir dafür leicht eine Tabelle einsetzen, die wir in der Table-Palette finden. In unserem Fall soll die Tabelle tracetabelle heißen und wird als globale (shared) Variable deklariert. Zur Erzeugung und Anzeige der Tabelle benötigen wir zwei Anweisungen:

Tabellenzeilen fügen wir mit *table <tablename> add row <data>* ein. In unserem Fall wollen wir bei Mausklicks die Koordinaten der Maus anzeigen. Wir fügen dazu in die Ereignisbehandlung *when clicked* die folgende Zeile ein:



Tabellen nutzen

Mit *join* aus der *Words*-Palette können wir eine Textzeile aus Einzelteilen (hier: 5) zusammensetzen. Das Ergebnis entsteht schrittweise am Bildschirm.

Table

<table>
<tr><th>11</th><th>(x|y)</th></tr>
<tr><td>1</td><td>(-297 | 58)</td></tr>
<tr><td>2</td><td>(-219 | -59)</td></tr>
<tr><td>3</td><td>(-297 | 65)</td></tr>
<tr><td>4</td><td>(-41 | -108)</td></tr>
<tr><td>5</td><td>(317 | -119)</td></tr>
<tr><td>6</td><td>(311 | 32)</td></tr>
<tr><td>7</td><td>(319 | 158)</td></tr>
<tr><td>8</td><td>(-10 | -141)</td></tr>
<tr><td>9</td><td>(-129 | -109)</td></tr>
<tr><td>10</td><td>(-73 | -34)</td></tr>
<tr><td>11</td><td>(-121 | 108)</td></tr>
</table>

Aus den gemachten Erfahrungen leiten wir einige Regeln für die Erstellung von Programmskripten ab:

Skripte sollten

- durch ausreichend viele Testläufe erprobt werden. Dabei sollten die Extremfälle (sehr große/kleine Werte, positive/negative/fehlerhafte Eingaben, …) systematisch durchgespielt werden.
- so weit möglich mit Testausgaben z. B. der Variablenwerte versehen werden, sodass ihr Ablauf leicht zu visualisieren und so zu überprüfen ist. Tracetabellen sind dafür ein gutes Mittel.
- in kleinen „Happen“ erstellt werden, die einzeln getestet werden können. Aus diesen wird dann das Gesamtskript zusammengesetzt.
- kurz sein. Sind sie das nicht, dann sollten sie in Teile aufgespalten werden.

2.4.3 Aufgaben

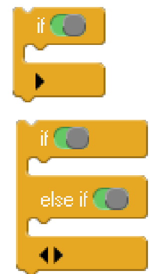
1. Begründen Sie Gleichungen für die die Koordinatentransformation vom Ausgabefenster zum Panel.

2. Erweitern Sie die *zeichne Rechteck* Methode so, dass sie auch in den genannten „Fehlerfällen“ richtig arbeitet.

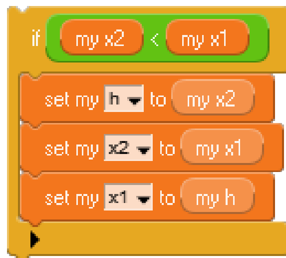
3. Ändern Sie die Anweisungen für die Tracetabelle so, dass die beiden Koordinaten in getrennten Spalten erscheinen.

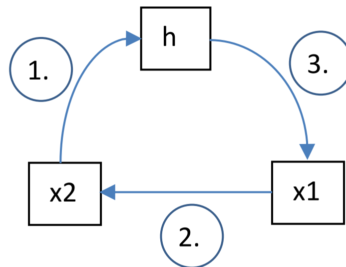
2.5 Rechtecke oder Kreise zeichnen

Zuerst wollen wir eine neue *Panel*-Methode schreiben, die ungeschickt angeklickte Koordinatenwerte korrigiert. Wir nennen sie *korrigiere Koordinaten*. Falsch wäre es z. B., wenn die *x2*-Koordinate links von *x1* liegt. Für solche Zwecke finden wir in der *Control*-Palette die Kontrollstruktur *Alternative*., die wir mithilfe der kleinen schwarzen Pfeile beliebig erweitern können. In unserem Fall sollen die Koordinaten vertauscht werden, wenn sie nicht stimmen. Das erledigen wir mithilfe eines *Ringtausches*, der eine dritte Variable (hier: *h*) benutzt, um Werte zu erhalten, die wir noch benötigen.



Ringtausch





Mit zwei solcher Anweisungsfolgen erledigen wir das Lageproblem der Koordinaten. Liegen sie auch noch zu eng zusammen, dann zeichnen wir gar keine Figur, sondern geben eine Warnung aus. Solche Alternativen lassen sich auch gut benutzen, um Anweisungen nur dann auszuführen, wenn bestimmte Bedingungen erfüllt sind. Speichern wird z. B. die ausgewählte Form in einer globalen Variablen *form*, dann zeichnen wir die entsprechenden Figuren nur dann, wenn *form* einen der beiden zugelassenen Werte hat.

Auf zulässige Werte testen

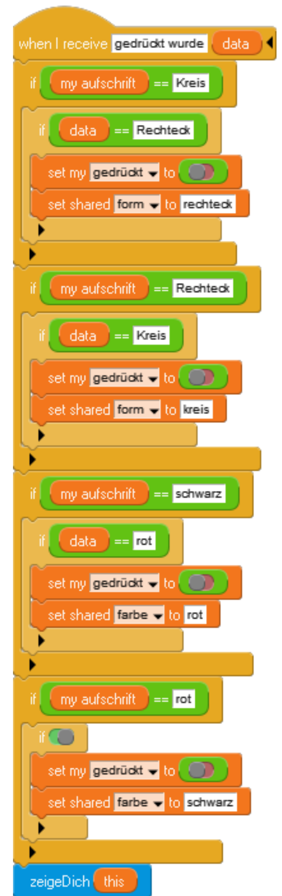
Jetzt kann eigentlich nichts mehr schiefgehen und wir können uns um das Zeichnen der Kreise kümmern. Bei diesen soll der erste Mausklick den Mittelpunkt und der zweite einen Punkt auf dem Kreisrand bedeuten.

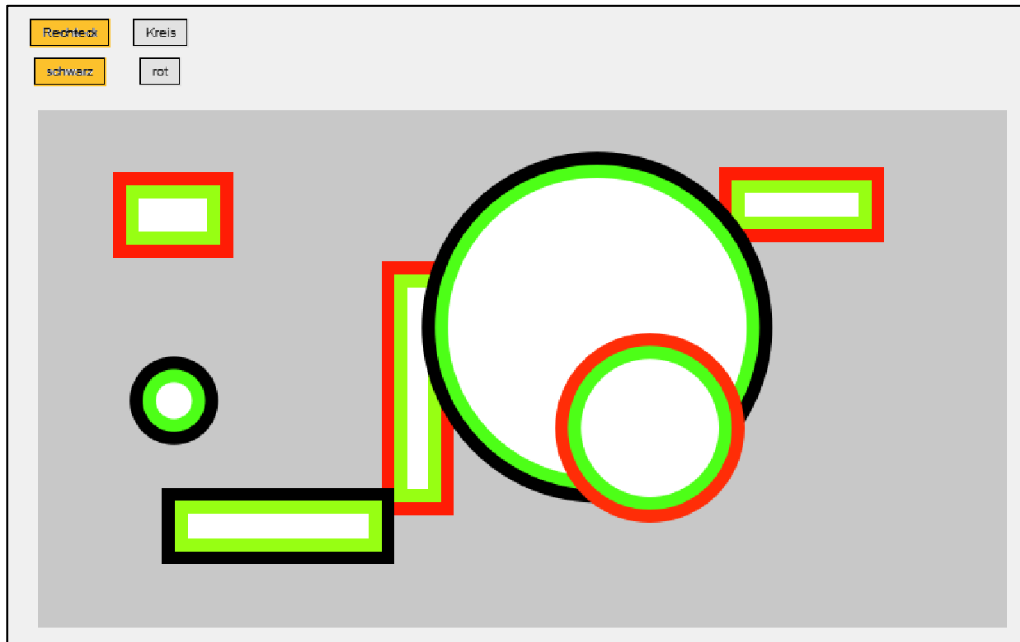
Ein ausführliches Skript mit Kommentaren

```
define zeichne Kreis this
comment Zuerst die Koordinatentransformation für den Mittelpunkt durchführen.
        Der Radius kann aus den Fensterkoordinaten berechnet werden.
set my panel mouse x to ((width / 2) + x position + my x1)
set my panel mouse y to ((height / 2) + y position) − my y1
set my r to sqrt (((my x1 − my x2) × (my x1 − my x2)) + ((my y1 − my y2) × (my y1 − my y2)))
comment Nur zeichnen, wenn der Radius groß genug ist.
if my r < 20
    say Die Punkte liegen zu nahe beieinander!
    wait 3 seconds
    say nothing
    set my ersterPunkt to (false)
else if (true)
    comment Je nach gewählter Farbe unterschiedlich zeichnen.
    if shared farbe == rot
        fill circle center x my panel mouse x y my panel mouse y radius my r (rot)
        fill circle center x my panel mouse x y my panel mouse y radius (my r − 10) (grün)
        fill circle center x my panel mouse x y my panel mouse y radius (my r − 20) (weiß)
    else if shared farbe == schwarz
        fill circle center x my panel mouse x y my panel mouse y radius my r (schwarz)
        fill circle center x my panel mouse x y my panel mouse y radius (my r − 10) (grün)
        fill circle center x my panel mouse x y my panel mouse y radius (my r − 20) (weiß)
```

Programmsteuerung über globale Variable

Jetzt bleibt nur noch, den Programmablauf mithilfe unserer vier Buttons zu steuern. Dafür erzeugen vier zwei globale Variable (*shared variables*) namens *farbe* und *form*, die die ausgewählten Größen speichern. Als Vorgabewerte setzen wir *rot* und *rechteck*. Diese Werte werden bei Buttonclicks geändert. Wir ergänzen diese Änderungen einfach im Eventhandler *when I receive* der Button-Klasse – und können unsere außerordentlich schönen Rechtecke und Kreise zeichnen.

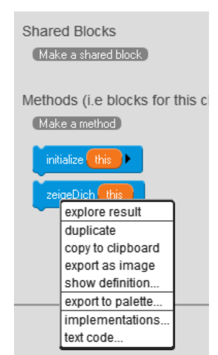




Unser Programm besteht aus ziemlich klar getrennten Blöcken:

- der Entwicklung und Nutzung einer Button-Klasse,
- der Entwicklung und Nutzung einer Panel-Klasse,
- dem Zeichnen von Rechtecken,
- dem Zeichnen von Kreisen
- und der Koordination der Komponenten.

Folgerichtig wurden die einzelnen Komponenten auch getrennt entwickelt und getestet. In der *My Blocks* Palette haben wir neue Methoden entwickelt, in denen Befehlsfolgen als neue Anweisungen zusammengefasst wurden. Solche Abkürzungen für mehrere Befehle nennt man *Makros*. Diese kann man in andere Klassen über das *Clipboard* kopieren oder als exportierte und in anderen Projekten importierte Klasse anderen zur Verfügung stellen. Wenn man will, können solche neuen Befehle auch in eine neue Palette exportiert und so in anderen Klassen genutzt werden.



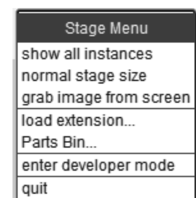
2.6 Arbeitsteilig vorgehen

die Funktionalität aufteilen …

Wenn wir unser Zeichenprogramm etwas genauer ansehen, dann haben wir zwei Gruppen von Knöpfen für die Auswahl der Figuren und Farben, die beide noch dringend erweitert werden müssen, sowie ein Panel, das neu geschriebene Blöcke benutzt, die unabhängig voneinander entwickelt werden können. Das Projekt bietet sich also für arbeitsteiliges Vorgehen an: einige Gruppen entwickeln zuverlässig funktionierende Knopfgruppen, andere sorgen dafür, dass die bei den Mausklicks ermittelten Koordinaten wirklich im Zeichenbereich und nicht sonstwo liegen oder entwickeln einen Radierer, neue Figuren oder Farbverläufe und andere grafische Elemente.

… und in Blöcke verlagern

Eine Möglichkeit zur Arbeitsteilung haben wir schon kennengelernt: die Blöcke. Eine zweite wären zusätzliche Sprites, mit denen neue Möglichkeiten verbunden sind. Kommunizieren können diese einerseits über globale Variable, andererseits durch das Versenden von Botschaften. Dies alles funktioniert aber nur, wenn wir Variable, Blöcke und Sprites von einem Programm zum anderen transportieren können, weil arbeitsteiliges Vorgehen natürlich an verschiedenen Rechnern erfolgt.



Blöcke und lokale Variable sind an Klassen gebunden, globale Variable aber nicht. Die müssen von den Arbeitsgruppen vereinbart werden. Der Export und Import von Klassen geschieht wie oben gezeigt. Auf diese Weise können sie als Bibliotheken dienen, die die gleiche Rolle übernehmen wie in anderen Sprachen. Im Kontextmenü der Arbeitsfläche (*stage-menu*) können sie als Erweiterungen es Systems geladen werden.

Man löst ein Problem arbeitsteilig, indem man

- die Funktionalität auf verschiedene Blöcke aufteilt und diese weitgehend unabhängig entwickelt und testet.
- sich auf Namen und Bedeutung möglichst weniger globaler Variablen einigt, die in mehreren Blöcken verwendet werden und so der Kommunikation dienen.
- die Gestaltung der Programmoberfläche (Lage der Knöpfe, …) möglichst in einer Hand lässt.
- die entwickelten Blöcke und Sprites an den Arbeitsplätzen exportiert und dann in ein gemeinsames Programm, das dann wohl auch die Programmoberfläche gestaltet, importiert.

2.7 Aufgaben

1. Führen Sie Möglichkeiten ein, um
   - a: drei Punkte anzuklicken, zwischen denen ein „schönes“ Dreieck gezeichnet wird.
   - b: bei jedem Mausklick ein Rechteck mit zufällig gewählter Breite und Höhe zu zeichnen. Benutzen Sie den random-Block aus der Operatoren-Palette.
   - c: eine Variable einzuführen, mit der sich nur die Werte 1, 2 und 3 auf irgendeine Weise einstellen lassen. Je nach Variablenwert erscheinen bei Mausklick Rechtecke, Dreiecke oder Rauten.

2. Mithilfe der *Zählschleife* repeat <number> lassen sich mehrere Figuren „im Stück zeichnen. Nutzen Sie diese Möglichkeit für Zufallsbilder.

3. Komplizierte Figuren lassen sich erstellen, indem entsprechende Kostüme entwickelt werden. Diese lassen sich auf den Bildschirm drucken. Wenden Sie dieses Verfahren zur Erstellung von Zufallsgrafiken an.

4. Benutzen Sie entweder die eingebaute Kamera Ihres Computers oder fertige Bilder, die mithilfe einer Digitalkamera (z. B. vom Handy) erstellt wurden. Importieren Sie Bilder als neue Kostüme. Benutzen Sie diese Bilder dann, um Benutzer auf Fehleingaben hinzuweisen. Zu den Bildern sollten passende Sprechblasen mit geeigneten Texten erscheinen.

5. Führen Sie einen Radiergummi ein, mit dessen Hilfe sich Bildteile wieder löschen lassen.

6. Führen Sie als neue Farbmöglichkeit Farbverläufe ein, die die gezeichneten Figuren durch Farbwechsel im Innenbereich „schön“ ausmalen. Sie können auch die Transparenz der Farben ändern und so durchscheinende Bilder erzeugen.

7. Entwickeln Sie eine Schriftart, deren Zeichen auf den Bildschirm gemalt werden können. Lassen Sie dann Texte ausgeben.

8. Versehen Sie eine Instanz mit Kostümen, die die einzelnen Zeichen einer Schriftart darstellen. Lassen Sie Texte ausgeben, indem Sie die Bilder zeichnen lassen.

3 Simulation eines Federpendels

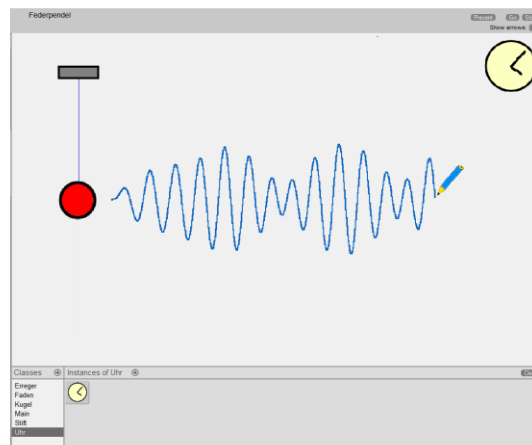
Neben der weitgehenden Syntaxfreiheit sind die exzellenten Visualisierungsmöglichkeiten und das gutmütige Verhalten von GP bei Fehlern ein Anreiz für die Lernenden, experimentell vorzugehen und so eigene Ideen zu erproben. Neben dem analytischen Top-Down-Vorgehen ergibt sich so ein Bottom-Up-Weg des Trials-and-Errors, der für Programmieranfänger wichtig ist, weil sie damit erst einmal Erfahrungen auf diesem Gebiet erwerben können, die dann später selbstverständlich zu systematisieren sind. Experimentelles Vorgehen öffnet so gerade am Anfang Möglichkeiten zum selbstständigen Problemlösen statt zum Nachvollziehen vorgegebener Ergebnisse.

Im Bereich der Simulation, zu der wir auch viele der üblichen Spiele rechnen können, finden wir genügend einfache, aber nicht triviale Problemstellungen, die schon von Anfängern bei etwas gutem Willen bearbeitet werden können. Experimentelles Arbeiten erfordert dabei natürlich ein Interesse, eigene Ideen zu entwickeln. Wir brauchen also Beispiele, die genügend Motivation erzeugen.

Als Beispiel wählen wir die Simulation eines einfachen Federpendels, das an einem periodisch schwingenden Erreger hängt. Ich weiß schon, dass ein Beispiel aus der Physik nicht bei allen Lernenden sehr motivierend wirkt – eher im Gegenteil. Aber ich gebe die Hoffnung nicht auf!

Das Beispiel enthält mehrere weitgehend unabhängig arbeitende Teile, sodass sich arbeitsteilige Gruppenarbeit geradezu aufdrängt.

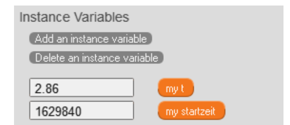
Wir identifizieren



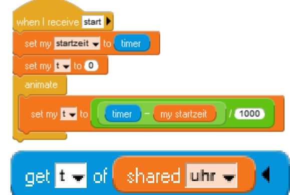
der Bildschirmaufbau

- einen *Erreger*, die dunkle Platte oben-links, der periodisch vertikal schwingt. Seine Frequenz *w* (statt ω) ist eine Instanzvariable und kann in der Variablenanzeige geändert werden.
- eine *Kugel*, die relativ dumm am Faden hängt, aber immerhin so viel Physik versteht, dass sie die Grundgleichung der Mechanik kennt.
- einen *Faden*, der sich selbst immer wieder neu zeichnen muss, damit wir keine überstehenden Enden am Bildschirm sehen.
- einen *Stift*, der das Weg-Zeit-Diagramm der Bewegung aufzeichnet.
- eine *Uhr* für die gemeinsame Zeit.

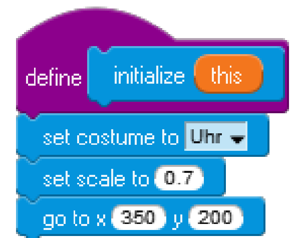
3.1 Die Uhr



Wir zeichnen eine einfache Uhr und erzeugen eine entsprechende Klasse. Die zugehörige Uhr-Instanz setzt, nachdem sie mithilfe der *start*-Nachricht gestartet wurde, die Instanzvariable *t* auf Null und merkt sich die Zeit des in GP eingebauten *Timers* in der Instanzvariablen *startzeit*. Danach übernimmt sie die aktuell vergangene Zeit in Sekunden fortlaufend in die Variable *t*, die den anderen Sprites als Systemzeit zur Verfügung steht.



Da die Zeit *t* logisch zur Uhr gehört, vereinbaren wir sie als lokale Variable. Der Zugriff auf lokale Variable erfolgt von anderen Klassen aus über den *get <attribute> of <object>* - Block der *Sensing*-Palette.



Um nach dem Import der Uhr gleich ein entsprechendes Bild zu haben, schreiben wir einen entsprechenden Konstruktor (eine Initialisierungsmethode).

Die Uhr greift auf keinerlei externe Größen zu und kann deshalb in allen *GP*-Programmen verwendet werden. Wir exportieren sie als *Uhr.gp*. Dabei ist zu beachten, dass Kostüme (*images*) in *GP* global sind, also nicht mit der Klasse exportiert werden. Images müssen deshalb extra exportiert und importiert werden.

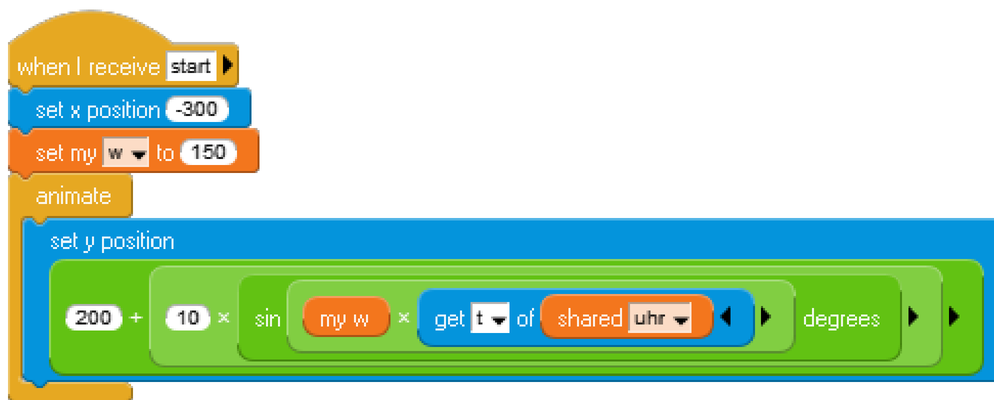
Erweiterung: Lassen Sie die Uhrzeit (Minuten und Sekunden) vom Sprite anzeigen, indem die Zeiger richtig bewegt werden. Sie sollten dazu die Uhr „hohl" zeichnen!

3.2 Der Erreger

Wir zeichnen ein einfaches Rechteck, das eine irgendwo aufgehängte Platte symbolisiert. Da die Platte nur vertikal schwingen soll, benötigt sie eine feste x-Koordinate am Bildschirm (hier: -300) sowie eine Ruhelage in y-Richtung (hier: 200). Um diese schwingt sie mit einer fest eingestellten Amplitude (hier: 10) mit einer variablen Kreisfrequenz ω (hier: 150). Im Laufe der Zeit t, die anfangs den Wert Null hat, berechnet sich die y-Koordinate dann zu

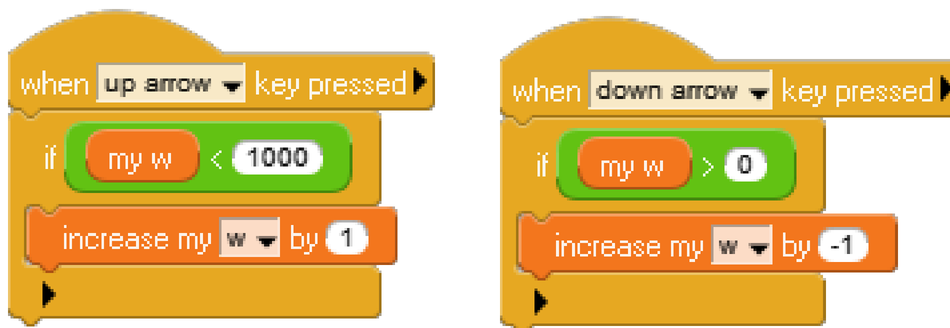
$$y = 200 + 10 * \sin(\omega t)\,.$$

Diese Angaben lassen sich direkt in ein Skript übersetzen.

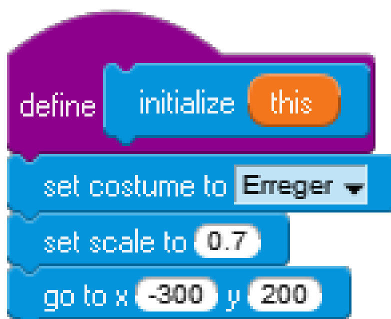


Das Skript beginnt seine Arbeit, wenn die *start*-Botschaft gesendet wird. Da die Skripte der anderen Teile zum gleichen Zeitpunkt gestartet werden müssen, bietet sich diese Möglichkeit an.

Interessanter sind die benutzten Variablen. Die Zeit wird von der Uhr importiert. Die Frequenz wird in keinem anderen Skript benötigt und sollte daher lokal vereinbart werden. Man kann sie mithilfe der Pfeiltasten ändern.



Und natürlich benötigen wir wieder einen Konstruktor.



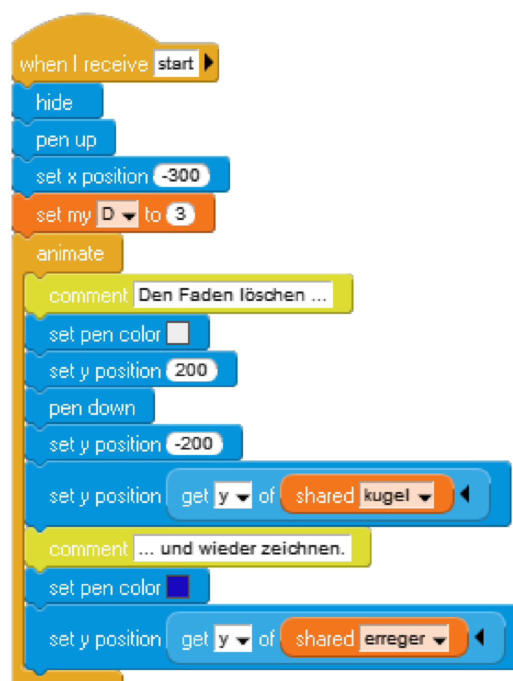
Wir exportieren die Klasse als *Erreger.gp*.

Erweiterung: Lassen Sie auch die „Labordecke" zeichnen, gegen die der Erreger schwingt. Alternativ dazu kann sich auch eine Welle drehen, die über eine Umlenkrolle zu einer senkrechten periodischen Bewegung führt.

3.3 Der Faden

Der Faden ersetzt die Feder. Er verfügt nur über eine einzige Eigenschaft, die Federkonstante *D*. Diese wird einmal auf einen festen Wert gesetzt, danach wird eine helle senkrechte Linie am Ort des Fadens gezeichnet, die seine alte Darstellung löscht (das geht natürlich auch eleganter). Danach wird die momentane Fadenauslenkung gezeichnet. Im Konstruktor wird die Instanz versteckt (hide). Wir exportieren die Klasse als Faden.gp.

Erweiterung: Zeichnen Sie statt des einfachen Fadens eine Spiralfeder mit einer konstanten Anzahl von Windungen, die sich dehnt und wieder zusammenzieht.



hier wurden mal Kommentare in das Skript eingefügt

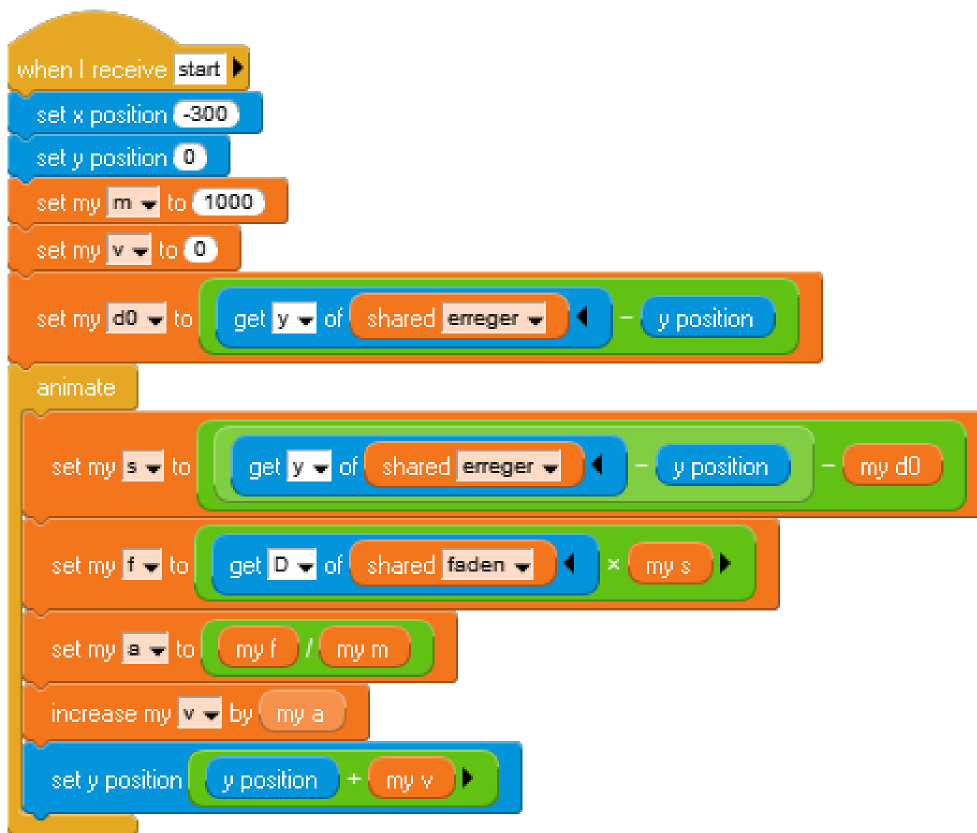
3.4 Die Kugel

In die Kugel werden unsere physikalischen Kenntnisse „eingebaut", die recht dürftig sein können: Wir kennen die Grundgleichung der Mechanik $F = m*a$ sowie das Hookesche Gesetz $F = D*s$, wobei es sich bei s um die Auslenkung aus der Ruhelage handelt. Weiterhin sind die Beschleunigung a als Geschwindigkeitsänderung pro Zeiteinheit und die Geschwindigkeit v als Wegänderung pro Zeiteinheit bekannt. Sonst nichts.

Der Konstruktor weist der Kugel wieder das entsprechende Bild zu.

Als lokale Variable benötigen wir die zu berechnenden Größen sowie die Masse m.

Wir setzen diese Kenntnisse in eine Folge von Befehlen um: wir bestimmen die momentane Auslenkung s, daraus F, daraus a, daraus v und daraus die neue Position.

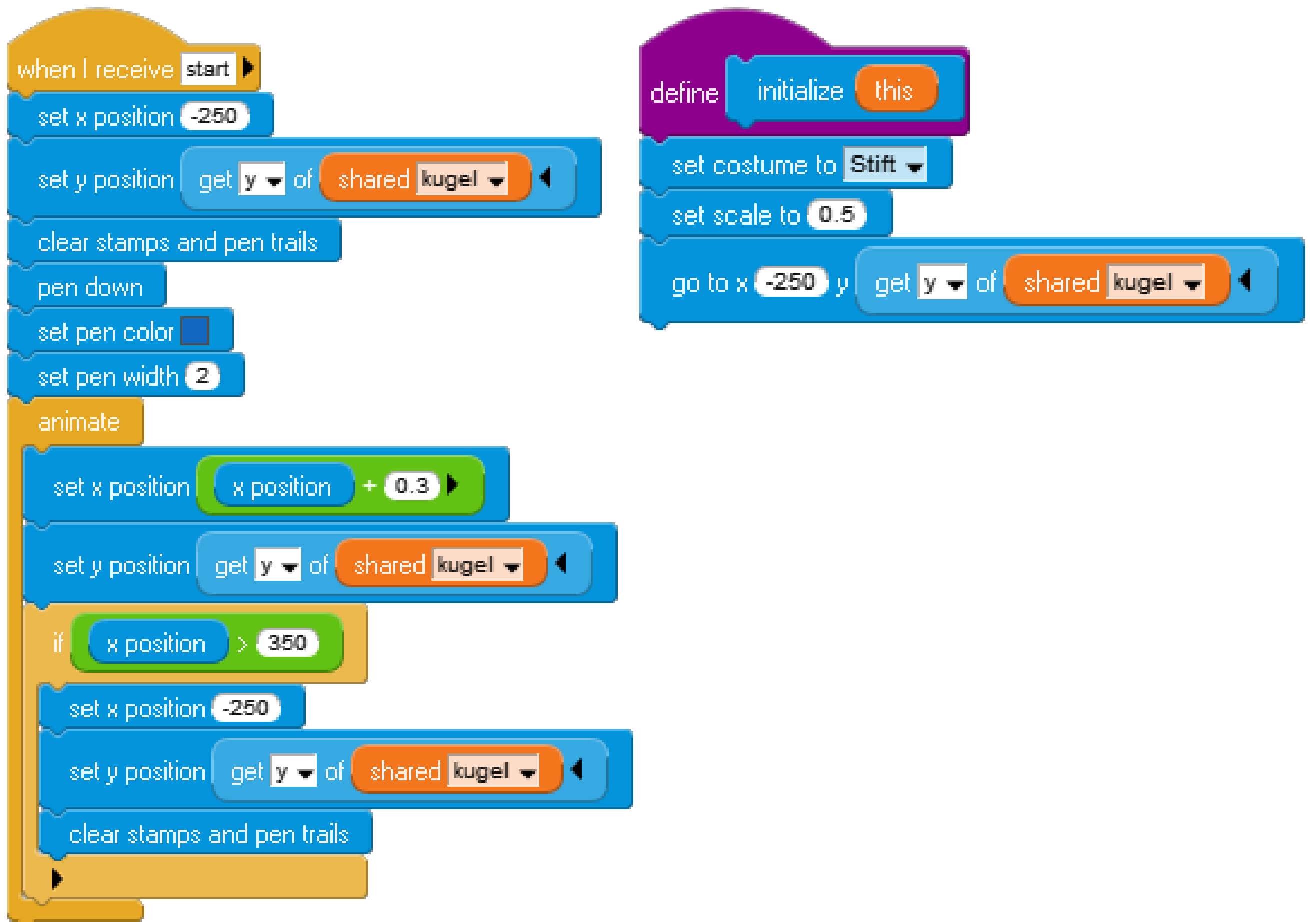


Auch fertig. Wir exportieren die Klasse als *Kugel.gp*.

<u>**Erweiterung**</u>: Führen Sie eine Reibungskonstante R ein, die die Geschwindigkeit um einen bestimmten (kleinen) Prozentsatz mindert. R soll auch interaktiv in einem sinnvollen Bereich änderbar sein.

3.5 Der Stift

Der Stift verfügt über keine lokalen Variablen. Er wandert langsam von links nach rechts und bewegt sich in y-Richtung zur y-Position der Kugel. Dabei schreibt er. Wir fügen als kleines Schmankerl die Funktion ein, dass er neu zu schreiben beginnt, wenn er den rechten Rand erreicht hat.



Wir exportieren die Klasse als *Stift.gp*.

Erweiterung: Führen Sie eine Möglichkeit ein, dass der Stift seine x-Position direkt aus der Systemzeit ableitet. Er soll auch mit unterschiedlichen Geschwindigkeiten laufen können.

3.6 Das Zusammenspiel der Komponenten

Vor Beginn der Arbeit müssen die globalen Variablen und die lokalen Attribute der Sprites (hier: Masse, Federkonstante, …) festgelegt werden. Die einzelnen Arbeitsgruppen erzeugen dann funktionslose Klassen dieses Namens, die über die genannten Attribute verfügen. Damit können sie in ihren Skripten auf entsprechende Attribute zugreifen. Als Alternative fügen sie einfach sinnvolle konstante Werte ein.

Der Zusammenbau beginnt mit einem leeren Projekt, in das die einzelnen Klassen und Bilder importiert werden. Danach werden die benötigten globalen Variablen erzeugt.

Um die Zugriffe zu vereinfachen, wird jeweils eine Instanz der Klassen in gleich benannten globalen Variablen gespeichert. Wir löschen dazu alle schon existierenden Instanzen einer Klasse, die wir über *neighbours within <radius> class <classname>* erhalten, und erzeugen ein neues Exemplar mit *new instance of <classname>*, das wir uns merken.

```
define delete all instances of this class
  while count neighbors within 1000 class class > 0
    delete first neighbors within 1000 class class
```

```
Shared Variables
Add a shared variable
Delete a shared variable
<Erreger>  shared erreger
<Faden>    shared faden
<Kugel>    shared kugel
<Stift>    shared stift
<Uhr>      shared uhr
```

```
when I receive go
hide
delete all instances of this Erreger
set shared erreger to new instance of Erreger
delete all instances of this Kugel
set shared kugel to new instance of Kugel
delete all instances of this Uhr
set shared uhr to new instance of Uhr
delete all instances of this Stift
set shared stift to new instance of Stift
delete all instances of this Faden
set shared faden to new instance of Faden
wait 0.1 seconds
broadcast start and wait
```

Das Verhalten wird in der Klasse Main.gp gespeichert. Fertig.

3.7 Weshalb handelt es sich um eine Simulation?

Unser Beispiel enthält zwar ein paar physikalische Grundkenntnisse, aber über Resonanz, Schwebung usw. ist darin nichts zu finden. Wir überprüfen mit dem Programm, ob die *denknotwendigen Folgen* (nach Heinrich Hertz) der Grundkenntnisse mit den Beobachtungen im Experiment übereinstimmen, ob unsere Vorstellungen von den physikalischen Zusammenhängen also das beobachtete Verhalten ergeben. Wir simulieren ein System, um unsere Vorstellungen zu überprüfen. Als Werkzeug dafür nutzen wir statt der Mathematik einen Algorithmus, der das Systemverhalten über eine Folge von kleinen zeitlichen Änderungen verfolgt. Statt also „mathematisch“ zu integrieren, iterieren wir „informatisch“. Außer in den einfachen Fällen macht ein Tool zur Integration eines Differentialgleichungssystems aber auch nichts anderes.

Etwas ganz anderes ist eine Animation, in die das beobachtete Verhalten einprogrammiert wird. Hier können sich keine neuen Phänomene ergeben, weil alles bekannt ist. *Animationen* stellen etwas dar, *Simulationen* können zu echten Überraschungen führen.

4 Ein Barcodescanner

Als Beispiel für die Nutzung von Blöcken zur Aufteilung eines Problems soll ein einfacher Barcodescanner dienen. Es wird exzessiv Gebrauch von „leeren“ Blöcken ohne Parameter gemacht, die Teilfunktionalitäten beschreiben, die dann nachträglich implementiert werden. Das Beispiel dient zur Verdeutlichung der Möglichkeiten der Top-Down-Entwicklung.

4.1 Der EAN-8-Code

Beim EAN-8-Code (*European Article Number*) handelt es sich um einen einfachen Code, der immer aus genau acht Zeichen besteht. Er dient meist zur Auszeichnung von Artikeln (Waren). Dazu enthält er drei Ziffern, die den Hersteller bezeichnen, und vier Ziffern, die zum Artikel gehören. Zuletzt kommt eine *Prüfziffer*.

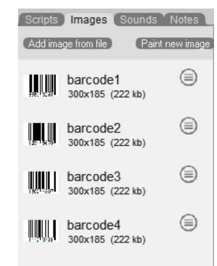
Die Ziffern werden durch wechselnde schwarze und weiße Streifen unterschiedlicher Breite codiert, wobei auch die weißen Streifen mitzählen. Die Streifenbreite kann 1, 2, 3 oder 4 betragen – in willkürlichen Einheiten. Eine (hier etwas vereinfachte) Darstellung der Ziffern 0 bis 9 lautet:

Ziffer	Code	Strichcode
0	3211	
1	2221	
2	2122	
3	1411	
4	1132	
5	1231	
6	1114	
7	1312	
8	1213	
9	3112	

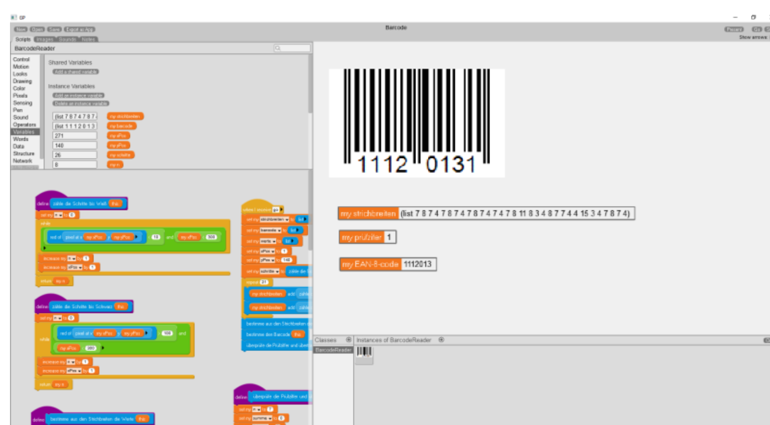
Der Code wird von Doppelstreifen an den Enden und in der Mitte eingerahmt, meist etwas länger gezeichnet (s. o.).

4.2 Blöcke als Strukturierungshilfe

mehrere Kostüme

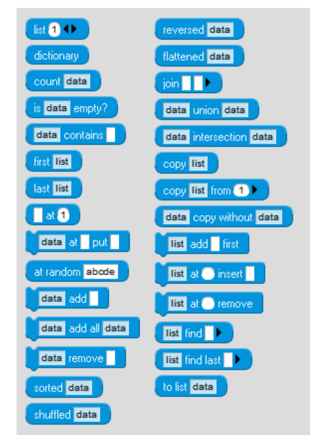


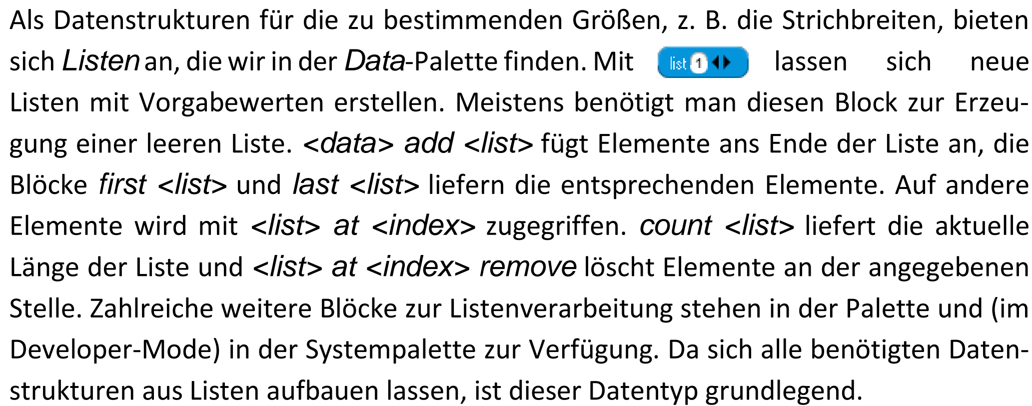
Wir benötigen für unser Beispiel nur eine Klasse, die wir *BarcodeReader* nennen. Der verpassen wir einige Barcodebilder, die jeweils einen EAN-8-Barcode darstellen[8], als Kostüme und passen die Größe der Instanz an die Bildgröße an. Der gelesene Code soll in einer Variablen namens *EAN-8-Code* dargestellt werden. Damit ergibt sich das dargestellte Szenario.



[8] Generatoren für Barcodes finden sich leicht im Internet.

Data-Palette



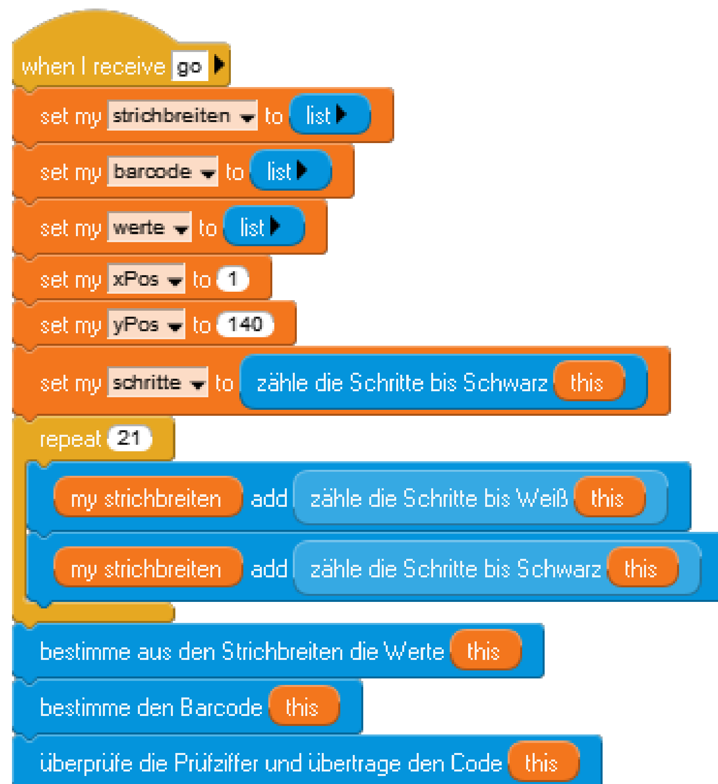


Als Datenstrukturen für die zu bestimmenden Größen, z. B. die Strichbreiten, bieten sich *Listen* an, die wir in der *Data*-Palette finden. Mit [list 1] lassen sich neue Listen mit Vorgabewerten erstellen. Meistens benötigt man diesen Block zur Erzeugung einer leeren Liste. *<data> add <list>* fügt Elemente ans Ende der Liste an, die Blöcke *first <list>* und *last <list>* liefern die entsprechenden Elemente. Auf andere Elemente wird mit *<list> at <index>* zugegriffen. *count <list>* liefert die aktuelle Länge der Liste und *<list> at <index> remove* löscht Elemente an der angegebenen Stelle. Zahlreiche weitere Blöcke zur Listenverarbeitung stehen in der Palette und (im Developer-Mode) in der Systempalette zur Verfügung. Da sich alle benötigten Datenstrukturen aus Listen aufbauen lassen, ist dieser Datentyp grundlegend.

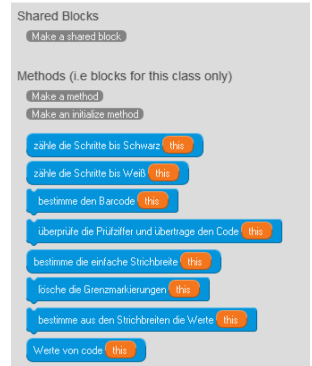
Vor Beginn der Arbeit, zeichnen wir eines der zur Verfügung stehenden Barcode-Kostüme. Dazu schreiben wir vier kleine Eventhandler, die auf die Tasten 1 bis 4 reagieren.

Wir wollen zur Abwechslung unser Problem mal auf dem Top-down-Weg bearbeiten, d. h. wir identifizieren zu lösende Teilprobleme, aus denen sich die Gesamtlösung zusammensetzen lässt, und lösen diese dann schrittweise auf dem gleichen Weg solange, bis wir bei den Blöcken der GP-Paletten angekommen sind.

In unserem Fall löschen wir anfangs die benötigten Listen und sehen uns dann auf einer Linie ungefähr auf halber Barcodehöhe die Farbe der Pixel an. Dann geht es los: Wir zählen so oft wie nötig abwechselnd die Schritte bis zur nächsten Markierung und tragen sie in die Liste *strichbreiten* ein. Danach ersetzen wir die gemessenen Pixelwerte durch die entsprechenden Werte 1 bis 4, bestimmen daraus den Code, prüfen seine Korrektheit und übertragen den Code als Zeichenkette in die dafür vorgesehene Variable. Zur Beschreibung dieses Vorgehens können wir funktionslose leere Blöcke erzeugen, die als Platzhalter im Hauptskript dienen.



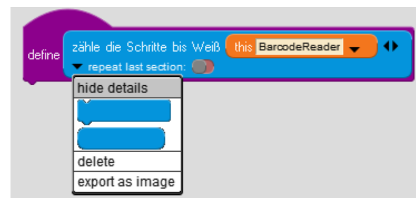
Die benötigten Methoden erzeugen wir in der *My Blocks* Palette mithilfe des Buttons *Make a method*. Dadurch wird ein Methodenkopf erstellt, an den die „richtigen" Blöcke angehängt werden. Das verschieben wir aber auf später. Wir müssen nur darauf achten, den richtigen Blocktyp auszuwählen:



Bei den ersten Blöcken handelt es sich um Funktionen (*reporter*), die ein Ergebnis zurückgeben. Man erkennt das an der ovalen Form.

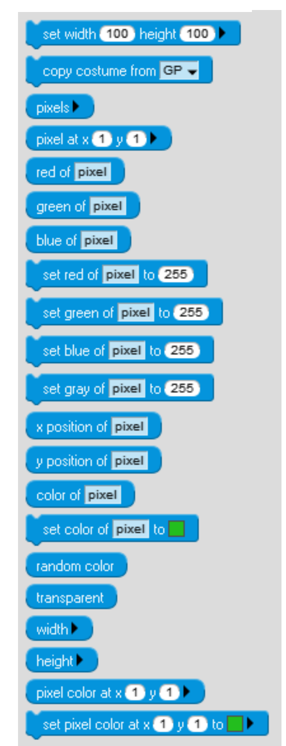
Die letzten drei Blöcke sind Befehle (*commands*), die einfach ausgeführt werden.

Mit einem Rechtsklick auf den Methodenkopf im Skriptbereich erhalten wir das entsprechende Kontextmenü, in dem wir den Methodentyp auswählen können.



Nun zu den einzelnen Methoden.

Pixels-Palette



Da es nur weiße und schwarze Pixel gibt, genügt es, sich die Helligkeit eines Farbkanals, z. B. des roten, anzusehen. Die dafür erforderlichen Blöcke finden wir in der Pixels-Palette. zähle die Schritte bis Schwarz zählt dann die hellen, also nicht schwarzen Pixel bis zum nächsten schwarzen Strich. Deren Zahl gibt der Reporter als Ergebnis zurück. Zähle die Schritte bis Weiß arbeitet entsprechend.

```
define zähle die Schritte bis Schwarz this
set my n to 0
while (red of pixel at x my xPos y my yPos > 100) and (my xPos < 300)
    increase my n by 1
    increase my xPos by 1
return my n
```

Jetzt müssen wir die im Code versteckten Werte ermitteln. Dazu bestimmen wir zuerst mal die Breite eines einfachen Strichs. Die haben wir ja zum Glück: die ersten drei Striche (2x schwarz, einmal weiß dazwischen) haben genau diese Breite. Danach löschen wir die Grenzmarkierungen und die der Mitte; die brauchen wir nicht mehr. Anschließend untersuchen wir alle gespeicherten Strichbreiten darauf, welchem Wert sie am besten entsprechen – und geben dabei eine halbe Strichbreite Spiel.

```
define bestimme aus den Strichbreiten die Werte this
set my breite to bestimme die einfache Strichbreite this
lösche die Grenzmarkierungen this
set my n to 1
repeat count my strichbreiten
    if abs(my strichbreiten at my n - my breite) < my breite / 2
        my werte add 1
    else if abs(my strichbreiten at my n - 2 × my breite) < my breite / 2
        my werte add 2
    else if abs(my strichbreiten at my n - 3 × my breite) < my breite / 2
        my werte add 3
    else if (true)
        my werte add 4
    increase my n by 1
```

Die einfache Strichbreite bestimmen wir durch Mittelwertbildung der ersten drei Messwerte.

```
define bestimme die einfache Strichbreite (this)
return ((my strichbreiten at 1) + (my strichbreiten at 2) + (my strichbreiten at 3)) / 3
```

Entsprechend einfach ist das Löschen der Markierungen.

```
define lösche die Grenzmarkierungen (this)
repeat 3
    my strichbreiten at 1 remove
repeat 5
    my strichbreiten at 17 remove
repeat 2
    my strichbreiten at (count (my strichbreiten)) remove
```

Damit ist das Schlimmste erledigt. Wir können den Barcode bestimmen, indem wir jeweils vier Werte zu einem Code (als Zeichenkette) zusammensetzen und dessen Wert ermitteln. Die erforderliche Codetabelle verpacken wir in einem Reporter-Block *Werte von <code>*. Um die Vergleiche ausführen zu können, müssen wir die Zeichenkette *code* vorher in eine Zahl umwandeln (*to number <string>*).

```
define bestimme den Barcode (this)
repeat 8
    set my n to 1
    set my code to
    repeat 4
        set my code to join (my code) (my werte at (my n))
        increase my n by 1
    repeat 4
        my werte at 1 remove
    my barcode add (Werte von code (this))
```

```
define Werte von code (this)
set my code to (to number (my code))
if (my code) == 3211
    return 0
else if (my code) == 2221
    return 1
else if (my code) == 2122
    return 2
else if (my code) == 1411
    return 3
else if (my code) == 1132
    return 4
else if (my code) == 1231
    return 5
else if (my code) == 1114
    return 6
else if (my code) == 1312
    return 7
else if (my code) == 1213
    return 8
else if (my code) == 3112
    return 9
```

An diesem Punkt der Verarbeitung liegt der Barcode in der dafür vorgesehenen Liste vor. Aber stimmt er überhaupt?

Wir berechnen zuerst die Prüfziffer aus den ersten sieben Ziffern. Dazu werden diese – von hinten beginnend – abwechselnd mit 3 und 1 multipliziert und addiert. Von der Summe wird der Rest bei Division durch 10 genommen und von 10 abgezogen. Sollte sich eine 10 ergeben, dann wird wiederum der Rest genommen.

Stimmt dieses Ergebnis mit der Prüfziffer überein, dann werden die ersten sieben Ziffern angezeigt, sonst ein Fehler.

Prüfsummenberechnung

```
define überprüfe die Prüfziffer und übertrage den Code (this)
set my n to 7
set my summe to 0
set my faktor to 3
while (my n > 0)
    increase my summe by (my faktor × (my barcode at my n))
    if (my faktor == 3)
        set my faktor to 1
    else if
        set my faktor to 3
    increase my n by -1
set my prüfzifer to ((10 − (my summe mod 10)) mod 10)
if (my prüfzifer == (my barcode at 8))
    set my EAN-8-code to
    set my n to 1
    repeat 7
        set my EAN-8-code to join (my EAN-8-code) (my barcode at my n)
        increase my n by 1
else if
    set my EAN-8-code to ERROR!
```

4.3 Aufgaben

1. Informieren Sie sich über andere Formen des EAN-Codes.

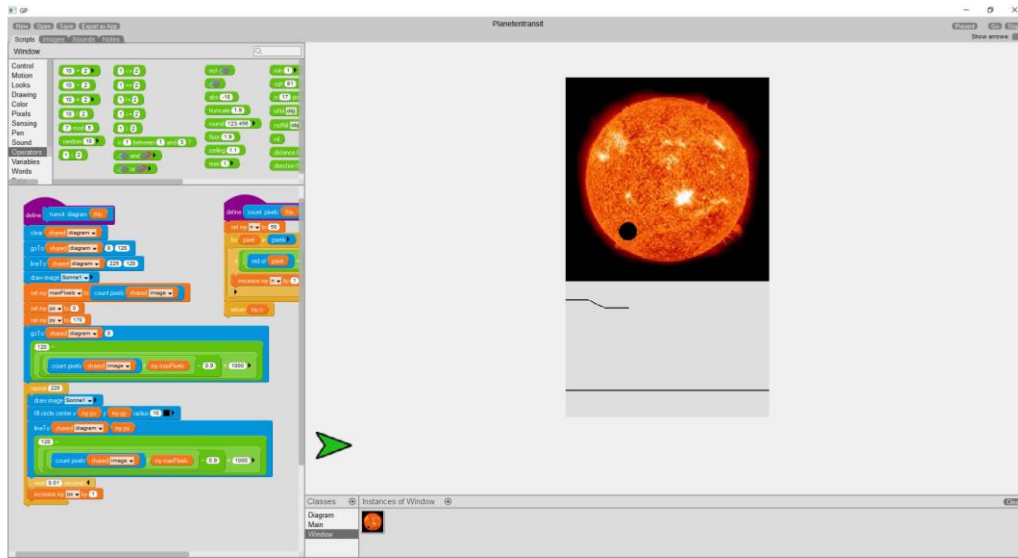
2. Implementieren Sie eine Prüfmethode, zu vor Beginn der Verarbeitung feststellt, ob es sich um einen zulässigen Code handelt.

3. Informieren Sie sich über den QR-Code. Schätzen Sie den Aufwand ab, der erforderlich ist, einen QR-Code-Leser zu implementieren.

5 Planeten-Transits

Eine Möglichkeit, extrasolare Planeten aufzuspüren, ist es, die Verdunkelung zu messen, wenn sie (zufällig) zwischen die Erde und ihre Sonne geraten. Die Methode funktioniert also nur bei einem Teil der Planeten, und die Verdunkelung ist sehr gering.

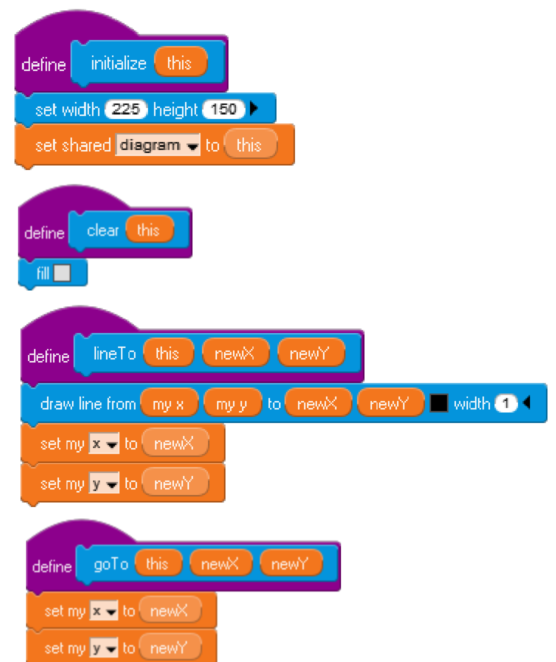
5.1 Das Szenario



Wie besorgen uns wie schon gewohnt ein Bild der Sonne, das wir auf eine Instanz der Klasse *Window* zeichnen. Über dieses Bild lassen wir einen schwarzen Kreis wandern, der den Planeten darstellt, und zählen jeweils die hellen Pixel im Bild. Das Ergebnis stellen wir im Diagramm darunter dar.

5.2 Eine Diagramm-Klasse

Unsere *Diagram*-Klasse ist ein reines Hilfsmittel. Wir schreiben einen Konstruktor, der eine Instanz in der gewünschten Größe erzeugt und sich diese in der globalen Variablen *diagram* merkt. Drei Hilfsmethoden dienen zum Löschen des Diagramms (*clear*), zum Zeichnen von Linien (*lineTo*) und zum Verschieben des Stifts (*goTo*). Sie arbeiten mit den Instanzvariablen *x* und *y*. Als Beispiel fügen wir im Folgenden solch ein Diagramm dem Window-Objekt als Teil hinzu.



5.3 Eine Window-Klasse

Die *Window*-Klasse erhält einen Konstruktor, der das Kostüm auswählt und das Diagramm als zusätzliches Teil (*part*) zu sich selbst hinzufügt. Damit sind Bildfenster und Diagramm ein Objekt.

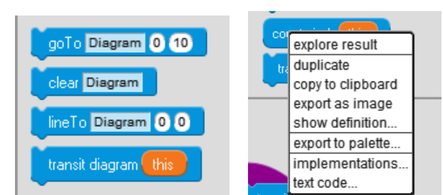
Die Methode *count pixels* benutzt eine Hilfsvariable *n*, die anfangs auf Null gesetzt wird. Anschließend werden in einer Zählschleife alle Pixel durchmustert. Sind sie hell genug, dann wird *n* erhöht. Das Ergebnis, die Zahl der hellen Pixel, wird zurückgegeben.

Die Methode *transit diagram* benutzt die Methoden der Diagramm-Klasse. Sie löscht das Diagramm und zeichnet ein rudimentäres Koordinatensystem. Dann zeichnet sie ihr eigenes Sonnenbild neu. Vom „Planeten" ist noch nichts zu sehen.

Da die Verdunkelung sehr gering ist, berechnet die Methode ein paar Größen, um die Darstellung des Transits etwas zu verbessern. Dafür benutzt sie z. B. die Zahl der hellen Pixel ohne Verdunkelung (*maxPixels*). *px* und *py* geben die Position des Planeten an. Danach zeichnet sie 220-mal zuerst das Sonnenbild und darauf den schwarzen Planeten an jeweils leicht veränderter Position. Die dann noch hellen Pixel werden immer wieder neu gezählt und in das Diagramm übertragen.

Die Main-Klasse ist weitgehend überflüssig. Sie enthält nur einen Block zur Diagrammerstellung.

Interessanter ist die Möglichkeit, die neu erstellten Methoden in einer neuen Palette (hier: *Transit Blocks*) zusammenzufassen. Mit einem Rechtsklick auf einen Methodenkopf in der *My Blocks*-Palette erhalten wir ein Kontextmenü, mit dessen Hilfe wir den Block in eine neue Palette exportieren können.



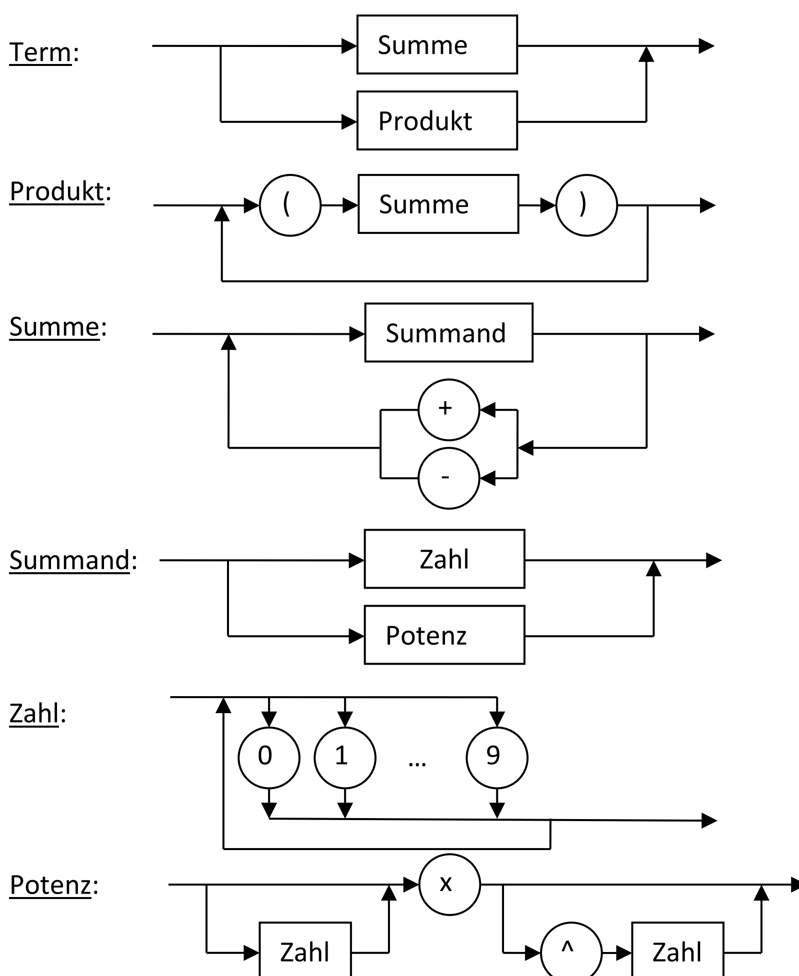
6 Computeralgebra: funktional programmieren

In der My Blocks-Palette können wir vier Arten von Unterprogrammen erzeugen: Methoden, bei denen die Voreinstellung command (Befehle) ist, die aber im Kontextmenü zu reportern (Funktionen) umwandelbar sind, initialize methods (Konstruktoren), die bei der Erzeugung eines Exemplars einer Klasse angewandt werden, und shared blocks - Methoden, die nicht an eine Klasse gebunden sind. Die Namen können völlig frei gewählt werden und auch aus mehreren Worten bestehen oder Sonderzeichen enthalten. Auf das erste Wort des Bezeichners können Parameter (input) und weitere Worte (label) in bunter Reihe folgen. Die freie Gestaltbarkeit der Blockköpfe gestattet es, die Sprache bei Bedarf bekannten Konventionen anzupassen, z. B. Klammern um die Parameter zu setzen etc. – wenn man das will.

6.1 Funktionsterme

Wir wollen die Möglichkeiten von Blöcken anhand eines kleinen „*Computeralgebrasystems*“ zeigen. Dazu müssen wir definieren, was wir unter Funktionstermen verstehen.

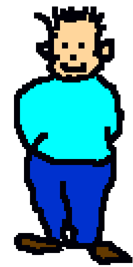
Syntaxdiagramme



Funktionsterme sind also z. B.: 3 4x (2x-1)(x^2+2) (x)(x^2)(1-2x^4)

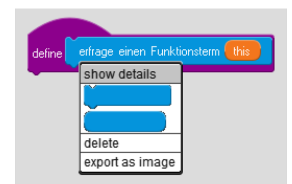
6.2 Funktionsterme mit einer Helper-Klasse parsen

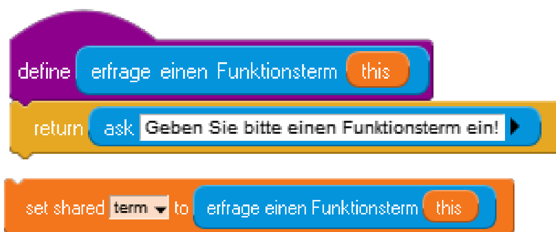
Zur Arbeit mit Funktionstermen brauchen wir natürlich jemanden, der etwas davon versteht. Wir zeichnen deshalb Paul, den kleinen Mathematiker, und machen den danach schlau. Paul benutzt für Berechnungen Helper-Klassen, deren Instanzen nicht sichtbar sind. Damit sind sie ideal für Hilfsaufgaben – wie der Name schon sagt.



Paul, der Mathematiker

Zuerst einmal muss Paul Funktionsterme einlesen können. Dafür bittet er den Benutzer um eine entsprechende Eingabe mithilfe des Blocks ask <question> aus der Sensing-Palette. Das Ganze verlagern wir in eine Methode Pauls, die wir als Funktion definieren. Wir wählen also die ovale Blockform im Kontextmenü aus. Haben wir eine Variable, z. B. namens term, vereinbart, dann können wir dieser das Ergebnis der Eingabe zuweisen.

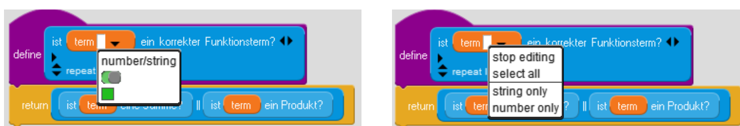




Eine Helper-Klasse einsetzen.

Als nächstes überprüfen wir, ob die Eingabe korrekt ist. Die entsprechenden Methoden verlagern wir in eine Helper-Klasse Parser. Da wir in diesem Abschnitt funktional programmieren wollen, benutzen wir ab jetzt nur noch globale Methoden (shared blocks in der My Blocks-Palette). Wir erzeugen also den Block ist <term> ein korrekter Funktionsterm?. Dazu geben wir als Funktionsbezeichner ist ein, verlängern die Kopfzeile mithilfe des kleinen schwarzen Pfeils um einen Parameter (input) und dann um den Rest des Funktionsnamens (als label). Den voreingestellten Bezeichner foo ändern wir in term. Klicken wir auf den kleinen schwarzen Pfeil daneben, dann können wir den Typ des Eingabeparameters einschränken (hier: number/string). Der ovale Platzhalter für Zahlen wird danach eckig (für Zeichenketten). Ein Rechtsklick in das Eingabefeld gestattet es, den Eingabetyp exakt festzulegen (hier: string only oder number only). Wir wählen string only für alle Eingabefelder, die Terme oder Zeichen enthalten.

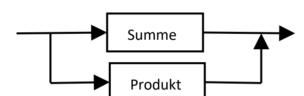
Funktionale Programmierung mit globalen Methoden.



Das Ergebnis sollte entweder true oder false sei, deshalb ist die Funktion ein Prädikat.

Jetzt haben wir zwar einen schönen Titel, aber leider keinen Inhalt. Trotzdem können wir den Block schon in Skripten verwenden – genauso wie weitere Blöcke. Das ermöglich einerseits rekursive Operationen, andererseits eignet es sich zur Top-Down-Entwicklung. Da laut der Syntaxdiagramme aus 6.1. korrekte Terme entweder Summen oder Produkte sind, verschieben wir das Problem dorthin, indem wir zwei entsprechende Prädikate erzeugen – immer noch leere.

Top-Down-Entwurf mit leeren Methoden



GP wertet logische Ausdrücke vollständig aus, was auch nett ist, wenn Nebeneffekte zu berücksichtigen sind. Das erhöht allerdings bei baumartigen Aufrufstrukturen gewaltig die Laufzeit. Deshalb schreiben wir zuerst zwei Prädikate für die *lazy evaluation* boolescher Ausdrücke: der zweite Ausdruck wird nur ausgewertet, wenn der erste nicht schon das Ergebnis bestimmt. Als Bezeichner wählen wir die in Programmiersprachen oft benutzen Operatoren && (lazy and) und || (lazy or). Als Bezeichner der Funktionen wählen wir anfangs das Leerzeichen, geben dann den ersten Parameter ein, dann den Funktionsnamen und dann den zweiten Parameter.

lazy evaluation

Das Prädikat ist <term> ein korrekter Funktionsterm? lässt sich jetzt vollständig angeben.

Dieses Verfahren setzen wir für alle Elemente der Sprachdefinition korrekter Funktionsterme fort. Zuerst nehmen wir uns die Summe vor. Diese besteht entweder aus einem einzelnen Summanden oder einem Summanden, gefolgt vom richtigen Operator (+/-) und einer Summe. Das können wir direkt hinschreiben, wenn wir über ein vorerst noch leeres Prädikat ist <term> ein Summand? verfügen.

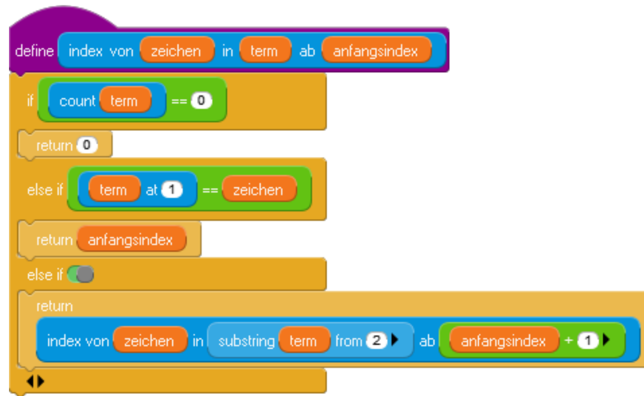
Wir müssen aufpassen, dass unsere Terme – also Zeichenketten – nicht versehentlich als Zahlen interpretiert werden. Aus diesem Grund haben wir den Typ der Eingabeparameter term immer auf „string only" festgelegt. Vergessen wir das, dann könnte z. B. die Zeichenkette „123" als Zahl 123 interpretiert werden. Das zweite Element der *Zeichenkette* ist eine 2, in der *Zahl* 123 gibt es aber kein zweites Element. Ein entsprechender Zugriff würde zu einem Fehler führen.

Achten Sie auf den Typ der Parameter!

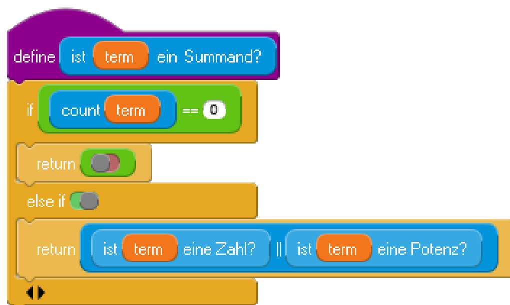
Wie brauchen noch etwas. Der eingegebene Term wird ja nicht mehr insgesamt untersucht, sondern wir müssen ihn ggf. in zwei Teile aufspalten: den anfang von <term> bis <zeichen> und den rest von <term> ab <zeichen>. Beide Funktionen arbeiten mit Zeichenketten und sind leicht zu schreiben. Wir erstellen rekursive Versionen der Methoden, die ohne lokale Variable auskommen.

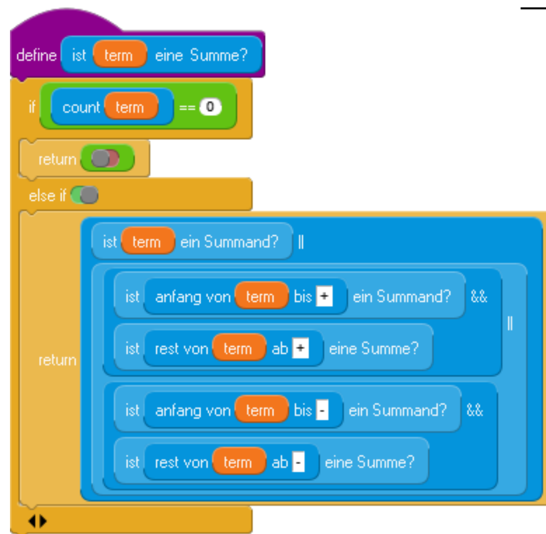
Rekursive Zeichenketten-verarbeitung

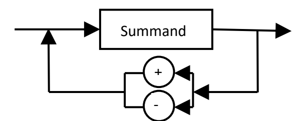
Und wenn wir schon einmal dabei sind, schreiben wir auch eine rekursive Methode, um die Position eines Zeichens in einem String zu bestimmen. Sie gibt 0 zurück, wenn das Zeichen nicht vorhanden ist.



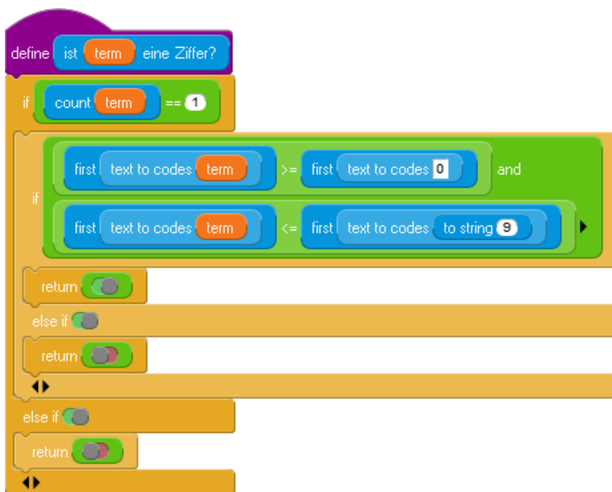
Damit schreiben wir die Prädikate ist <term> ein Summand? und ist <term> eine Summe? – jeweils mit einer zusätzlichen Sicherheitsabfrage.

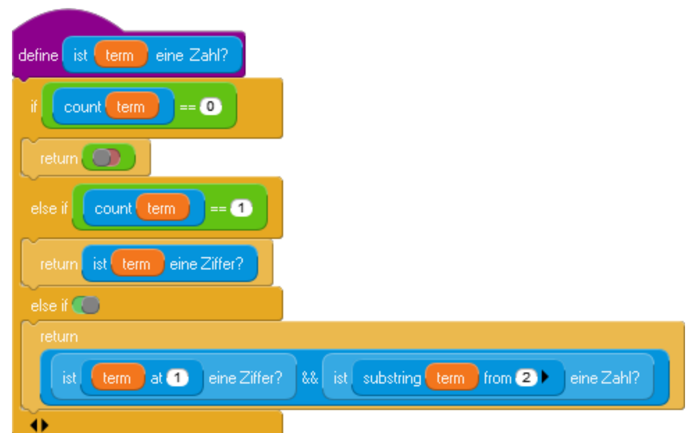






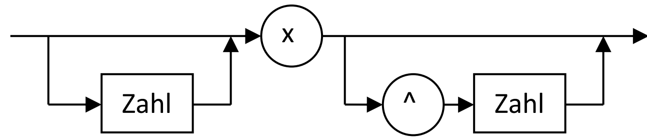
Wir nähern uns dem Ende. ist <term> eine Zahl? ist sehr leicht zu schreiben, wenn man ist <term> eine Ziffer? kennt:





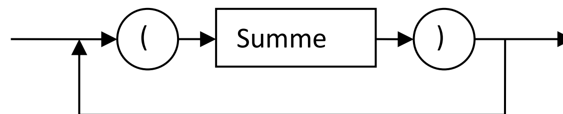
```
define ist (term) eine Potenz?
if (count (term) == 0)
    return false
else if (index von x in (term) ab 1 == 0)
    return false
else if ((term) == x)
    return true
else if not ((ist anfang von (term) bis x eine Zahl?) || (count anfang von (term) bis x == 0))
    return false
else if (count rest von (term) ab x == 0)
    return true
else if not (rest von (term) ab x at 1 == ^)
    return false
else if (ist rest von rest von (term) ab x ab ^ eine Zahl?)
    return true
else if false
    return false
```

Und wie überprüft man eine Potenz? Das steht ja auch im Syntaxdiagramm – wir müssen nur alle Möglichkeiten abschreiben.



Jetzt fehlt nur noch das Produkt, das sich in direkter Analogie zur Summe formulieren lässt, denn ein Produkt besteht (bei uns) entweder aus einer geklammerten Summe oder einer solchen, gefolgt von einem Produkt.

```
define ist (term) ein Produkt?
if (count (term) < 3)
    return false
else if true
    return (((term) at 1 == ( ) && (ist anfang von rest von (term) ab ( bis ) eine Summe?) && ((term) at count (term) == ) ))
           && ((count rest von (term) ab ) == 0) || (ist rest von (term) ab ) ein Produkt?))
```



auch schön rekursiv

Wir können damit überprüfen (parsen), ob ein eingegebener Term der gewählten Syntax entspricht. Ist das der Fall, dann kann damit weitergearbeitet werden.

```
set shared term to (erfrage einen Funktionsterm (this))
ist (shared term) ein korrekter Funktionsterm?   → true
```

6.3 Funktionsterme ableiten

Wir wollen jetzt die erste Ableitung korrekter Funktionsterme bestimmen. Die erforderlichen Methoden sammeln wir in der Helper-Klasse namens Ableiter. Da es nur zwei Möglichkeiten für den inneren Aufbau von Termen gibt, ist der erste Ansatz einfach.

```
define ableitung von term
if ist term eine Summe?
    return wende die Summenregel auf term an
else if
    return wende die Produktregel auf term an
```

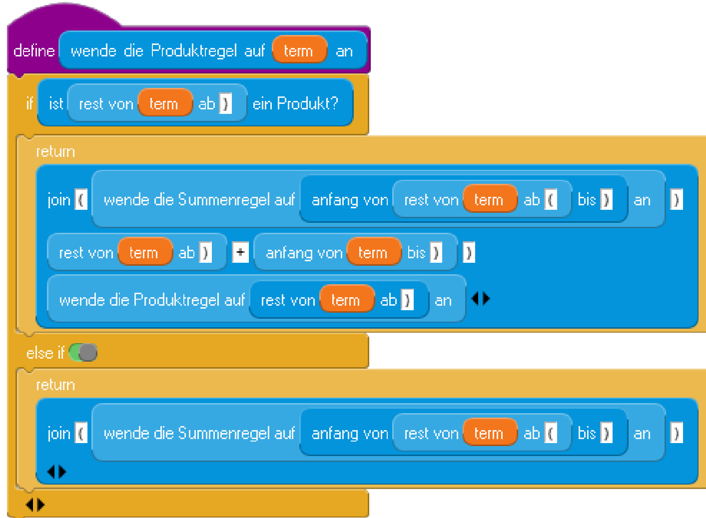
Bei Anwendung der Summenregel müssen wir die Summanden bestimmen und diese ableiten. Weil wir Zahlen ohne Vorzeichen definiert haben, behandeln wir dieses jeweils gesondert, d. h. wir fügen bei Bedarf ein „+“ hinzu und spalten anschließend das Vorzeichen wieder ab. Anschließend werden die verschiedenen Möglichkeiten entsprechend den Regeln der Mathematik behandelt.

```
define summenableitung von summand
if (summand at 1 == +) or (summand at 1 == -)
    set summand to substring summand from 2
if ist summand eine Zahl?
    return 0
else if summand at 1 == x
    if count rest von summand ab x == 0
        return 1
    else if rest von summand ab ^ == 2
        return join (rest von summand ab ^) (x)
    else if
        return join (rest von summand ab ^) (x^) (to number rest von summand ab ^ - 1)
else if count rest von summand ab x == 0
    return anfang von summand bis x
else if rest von summand ab ^ == 2
    return join (to string (to number anfang von summand bis x) × (to number rest von summand ab ^)) (x)
else if
    return join (to string (to number anfang von summand bis x) × (to number rest von summand ab ^)) (x^) (to number rest von summand ab ^ - 1)
```

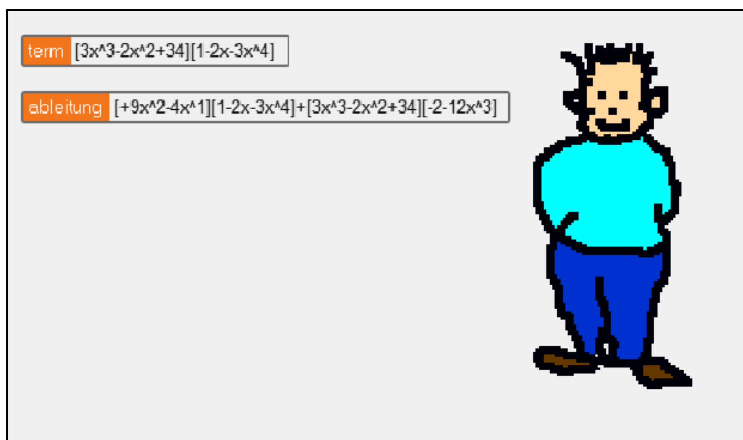
```
define wende die Summenregel auf term an
let ergebnis be ""
let summand be ""
let vorzeichen be ""
let h be term
if not ((term at 1 == +) or (term at 1 == -))
    set h to join + h
while count h > 0
    set vorzeichen to h at 1
    set h to substring h from 2
    if count anfang von h bis - < count anfang von h bis +
        set summand to anfang von h bis -
        set h to join - (rest von h ab -)
    else if count anfang von h bis - == count anfang von h bis +
        set summand to h
        set h to ""
    else if
        set summand to anfang von h bis +
        set h to join + (rest von h ab +)
    set summand to summenableitung von summand
    if summand != to string 0
        set ergebnis to join ergebnis vorzeichen summand
if count ergebnis == 0
    return 0
else if
    return ergebnis
```

Bei der Anwendung der Summenregel wurden zur Abwechslung mal Skriptvariable benutzt. Das kürzt die Schreibweisen etwas ab.

Jetzt fehlt nur noch die Produktregel. Die können wir einfach hinschreiben – unter Hinzufügung einiger Klammern.

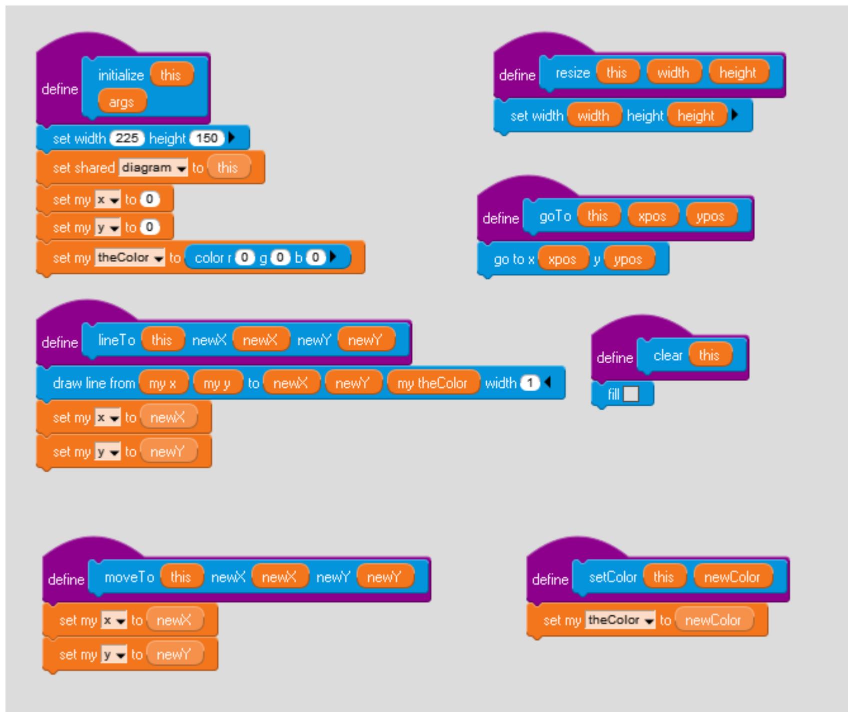


Das Ergebnis kann man sogar halbwegs lesen:

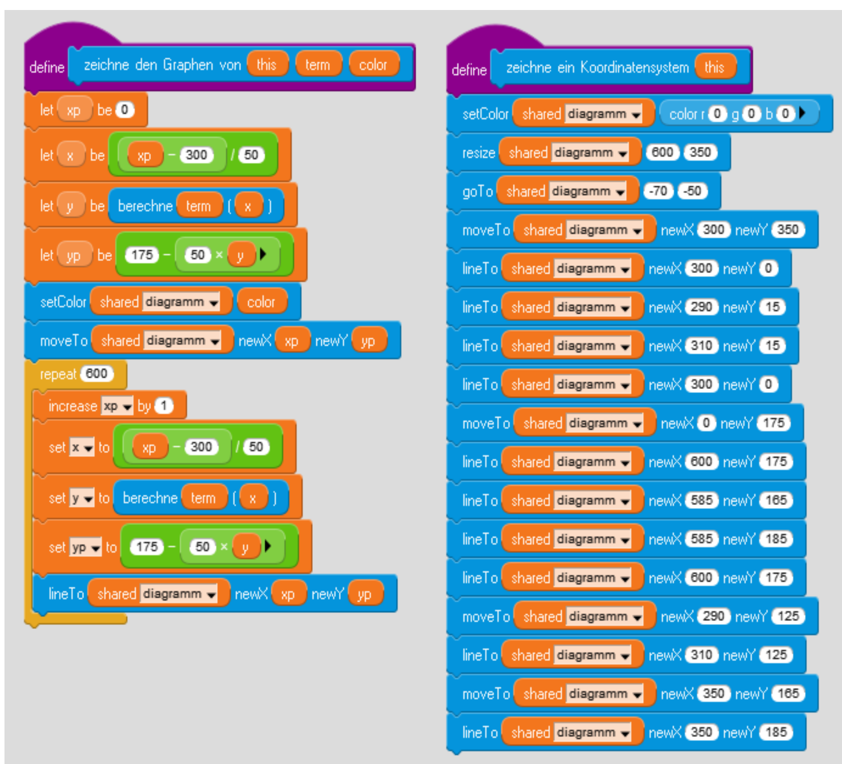


6.4 Funktionswerte berechnen und Graphen zeichnen

Wenn wir Funktionswerte parsen können, dann können wir sie natürlich auch berechnen. Das Vorgehen ist ganz ähnlich wie beim Parsen, und es wird sehr erleichtert, wenn wir schon wissen, dass der eingegebene Term korrekt ist. Wir wollen Funktionswerte berechnen und dann die Graphen der Funktion und ihrer ersten Ableitung zeichnen. Eine Diagrammklasse haben wir ja schon beim Planetentransit benutzt. Wir erweitern sie ein bisschen und geben sie dann Paul zum Spielen.



Diese Methoden exportieren wir in eine Palette Diagramm und legen eine globale Variable diagramm an. Die kann Paul benutzen.



In diesen Skripten sind alle Blöcke schon vorhanden – bis auf einen. Es fehlt noch die Berechnung eines Funktionsterms an der Stelle *x*. Wir überlassen diese Aufgabe einer Klasse *Funktionsrechner* und geben die entsprechenden Skripte nur an.

Mit deren Hilfe kann Paul nun glänzen:

```
when I receive go
set shared term to erfrage einen Funktionsterm this
if ist shared term ein korrekter Funktionsterm?
    say nothing
    set shared ableitung to ableitung von shared term
    zeichne ein Koordinatensystem this
    zeichne den Graphen von this shared term color r 255 g 0 b 0
    if ist shared ableitung ein korrekter Funktionsterm?
        zeichne den Graphen von this shared ableitung color r 0 g 0 b 255
    else if
        say Die Ableitung entspricht leider nicht der gegebenen Syntax.
            Ich kann sie nicht zeichnen.
else if
    say Das ist leider kein korrekter Term!
        Versuchen Sie es noch einmal!
        (Klicken Sie auf "Go")
    wait 5 seconds
    say nothing
```

term (x^3-3x)(2-2x-x^2)

ableitung (+3x^2-3)(2-2x-x^2)+(x^3-3x)(+0-2-2x^1)

Die Ableitung entspricht leider nicht der gegebenen Syntax.
Ich kann sie nicht zeichnen.

6.5 Aufgaben

1. a: Gestalten Sie die Ausgaben etwas lesefreundlicher: führende „+“ sollen entfernt werden etc.
   b: Fassen Sie Ergebnisse in der Ableitung so zusammen, dass diese der gegebenen Syntax entspricht und gezeichnet werden kann.

2. a: Definieren Sie Zahlen mit Vorzeichen und ändern Sie die Verarbeitung der Terme entsprechend ab.
   b: Gehen Sie entsprechend für Gleitpunktzahlen (Zahlen mit Nachkommateil) vor.

3. a: Definieren Sie erweiterte Funktionsterme, die auch Quotienten enthalten können, über Syntaxdiagramme.
   b: Ermöglichen Sie das Parsen dieser Funktionsterme, indem Sie entsprechende Prädikate schreiben.
   c: Bilden Sie Ableitungen, indem Sie auch die Quotientenregel als Zeichenkettenoperation implementieren.

4. Gehen Sie entsprechend der Aufgabe 3 für trigonometrische Funktionen vor.

5. Lassen Sie Funktionsterme zu, die die Anwendung der Kettenregel erfordern. Implementieren Sie entsprechende Prädikate und Zeichenkettenfunktionen.

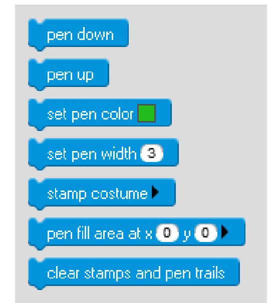
6. a: Lassen Sie die Graphen auch der anderen Funktionsarten zeichnen, nachdem sie geparst wurden.
   b: Lassen Sie eine Auswahl der zu zeichnenden Graphen (Funktion, erste und zweite Ableitung) zu.

7. Führen Sie einen „Funktions-Taschenrechner“ ein: zuerst wird ein Funktionsterm eingegeben. Ist dieser korrekt, dann können wiederholt Werte eingegeben werden, für die die zugehörigen Funktionswerte ermittelt werden.

7 Rekursive Kurven

Nachdem wir nun die Rekursionsmöglichkeiten von GP kennengelernt haben, wenden wir sie auf die implementierte Turtlegrafik an. Die dafür erforderlichen Blöcke verteilen sich auf die Rubriken *Pen* und *Motion*.

Palette Pen



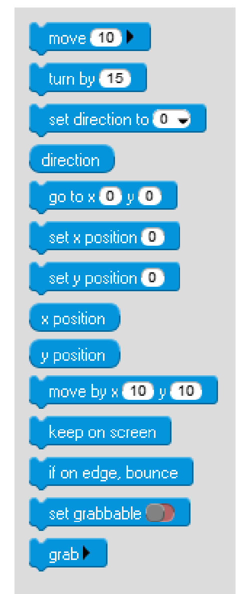
7.1 Die Schneeflockenkurve

Sie entsteht, indem auf einer Seite immer wieder in der Mitte ein Dreieck „ausgestülpt“ wird, solange bis die Seite zu kurz für diesen Prozess ist. In diesem Fall wird die Seite nur als gerade Linie gezeichnet.

zeichne Schneeflockenseite der Länge n

n < 2	
wahr	
zeichne eine Linie der Länge n	zeichne Schneeflockenseite der Länge n/3
	drehe dich um -60°
	zeichne Schneeflockenseite der Länge n/3
	drehe dich um 120°
	zeichne Schneeflockenseite der Länge n/3
	drehe dich um -60°
	zeichne Schneeflockenseite der Länge n/3

Palette Motion

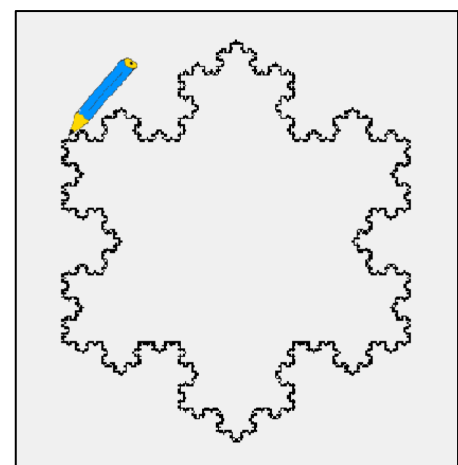


Eine Schneeflocke entsteht, indem ein gleichseitiges „Dreieck“ aus drei solchen Seiten zusammengesetzt wird.

Das Verfahren lässt sich direkt nach GP übersetzen:

```
define zeichne Schneeflockenseite this n
  if n < 2
    move n
  else if
    zeichne Schneeflockenseite this n / 3
    turn by -60
    zeichne Schneeflockenseite this n / 3
    turn by 120
    zeichne Schneeflockenseite this n / 3
    turn by -60
    zeichne Schneeflockenseite this n / 3
```

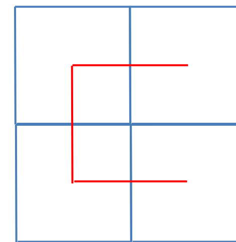
```
define zeichne Schneeflocke this n
  pen down
  repeat 3
    zeichne Schneeflockenseite this n
    turn by 120
  pen up

when I receive go
  clear stamps and pen trails
  set scale to 0.5
  zeichne Schneeflocke this 200
```



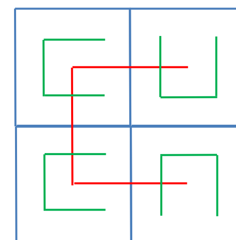
7.2 Die Hilbertkurve

Wir verwenden zur Konstruktion der Kurve eine Version nach László Böszörményi[9].

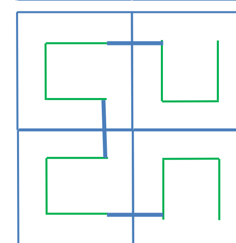
Die Hilbertkurve ist eine der flächenfüllenden Kurven, die als Generator eine Art Kasten hat. Die Ecken des Kastens liegen in den Mittelpunkten der vier Quadranten eines Quadrats. In der nächsten Stufe wird dieser Kasten um die Hälfte verkleinert, und davon werden vier Versionen in gespiegelter bzw. gedrehter Version in den Quadranten neu angeordnet. Zuletzt werden die kleineren Kästen wie gezeigt miteinander verbunden.



der Generator

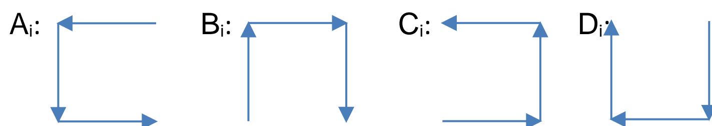


seine Lage im Quadrat

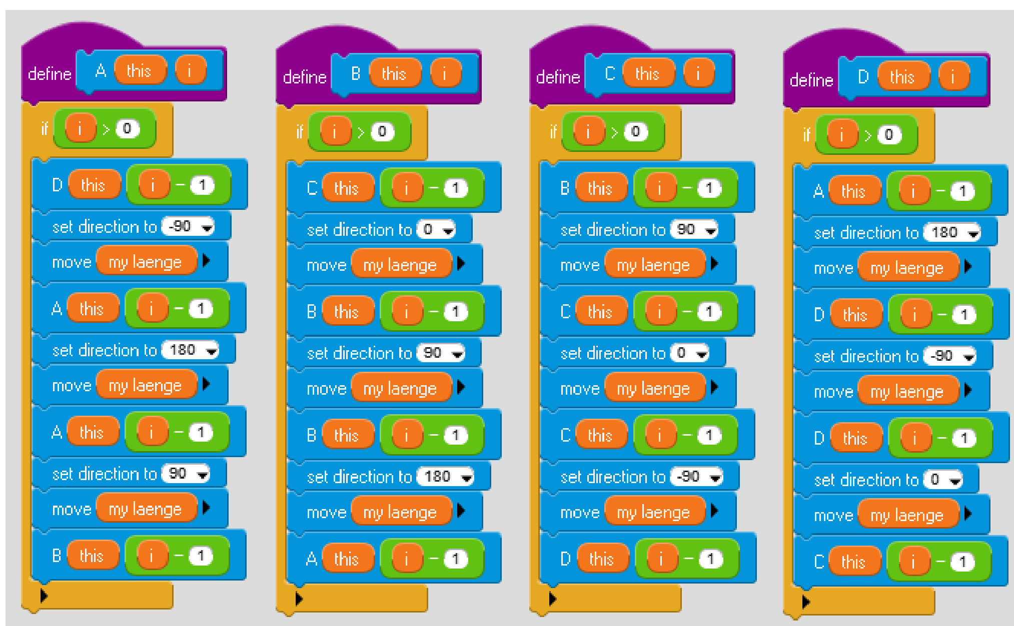


die verkleinerten Kopien und ihre Verbindungen

In der Version von Böszörményi werden die Kästen je nach Orientierung und Umlaufrichtung mit A bis D gekennzeichnet.

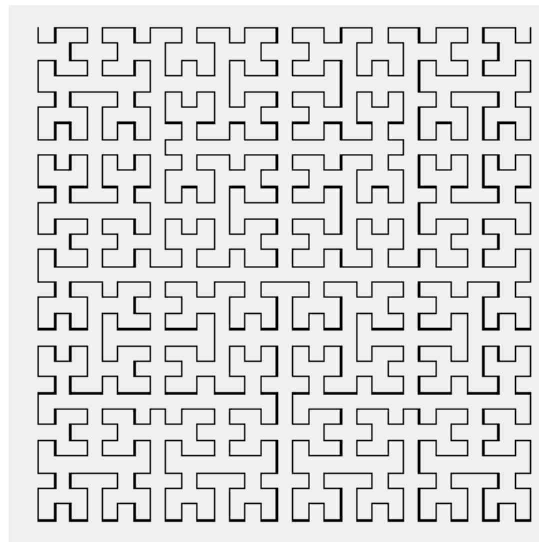


Aus diesen Elementen wird die Hilbertkurve zusammengesetzt, indem man mit A beginnt und die anderen Elemente „verdreht“ aufruft. Der Parameter i gibt die Rekursionstiefe und damit die Größe der Elemente an. Er wird „heruntergezählt“ bis auf Null.



[9] http://bscwpub-itec.uni-klu.ac.at/pub/bscw.cgi/d11952/10.%20Rekursive%20Algorithmen.pdf

Der Aufruf erfolgt wie beschrieben, nachdem das Sprite zum Anfangspunkt rechts-oben geschickt wurde. Die endgültige Länge der zu zeichnenden Teilstrecken wird aus der Rekursionstiefe ermittelt – und dann wird gezeichnet.



die Hilbert-Kurve in der Rekursions-tiefe 5

7.3 Aufgaben

1. a: Informieren Sie sich im Internet zum Thema C-Kurve.
   b: Probieren Sie einige Schritte zur Konstruktion der Kurve „per Hand“ aus.
   c: Implementieren Sie ein Skript zum Zeichnen der Kurve in GP.

2. Verfahren Sie entsprechend für die Dragon-Kurve.

3. Verfahren Sie entsprechend für die Peano-Kurve.

4. Verfahren Sie entsprechend für die Sierpinski-Kurve.

8 Listen und verwandte Strukturen

GP kennt neben atomaren Datentypen wie *Zahlen*, *Wahrheitsswerten* und *Zeichen* die strukturierten Typen *String*, *Array*, *Dictionary* und *List*. In diesem Abschnitt beschäftigen wir uns mit Listen, weil sich aus ihnen als grundlegender Datenstruktur alle höheren Strukturen leicht aufbauen lassen. Die Verwendung von Listen wird zuerst an einem einfachen Fall – dem Sortieren – gezeigt, danach werden *Adjazenzlisten* für die Wegsuche benutzt und zuletzt werden *Matrizen* eingeführt.

8.1 Sortieren mit Listen – durch Auswahl

die Data-Palette

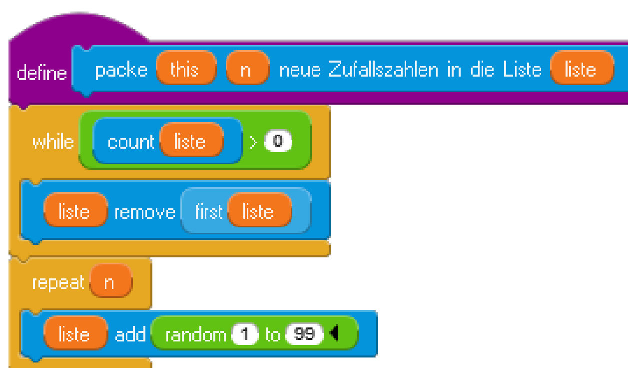
Zuerst einmal benötigen wir eine Liste. Die Blöcke dafür finden wir in der Data-Palette. Da es sich weitgehend um Reporter-Blöcke handelt, müssen wir die Ergebnisse auch irgendwo speichern. Dafür erstellen wir eine (hier) globale Variable namens *daten*. Dieser weisen wir eine leere Liste zu. (Mithilfe der schwarzen Pfeiltasten können wir den Listenblock auch erweitern und so beliebige Anfangswerte in die Liste schreiben.)

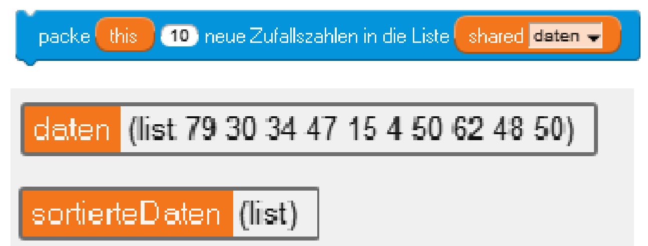
Benutzen wir einen Monitor für unsere Variable (*Rechtsklick auf die Variable in der Variablenpalette*), dann wird dort die Liste angezeigt.

daten (list Peter Meier 42)

Auf die gleiche Art erzeugen wir eine zweite Liste sortierteDaten, die später die sortierten Daten aufnehmen soll.

Zuerst einmal brauchen wir unsortierte Daten – wie üblich Zufallszahlen. Die erzeugen wir mit einem kleinen Skript, wobei wir die Anzahl der Zahlen sowie die zu füllende Liste als Parameter angeben. Dabei sollen zuerst die alten Werte in der Liste gelöscht werden. Wir löschen also das erste Element der Liste, bis die Liste leer ist. Danach fügen wir *n* Zufallszahlen zwischen *1* und *99* der Liste hinzu.

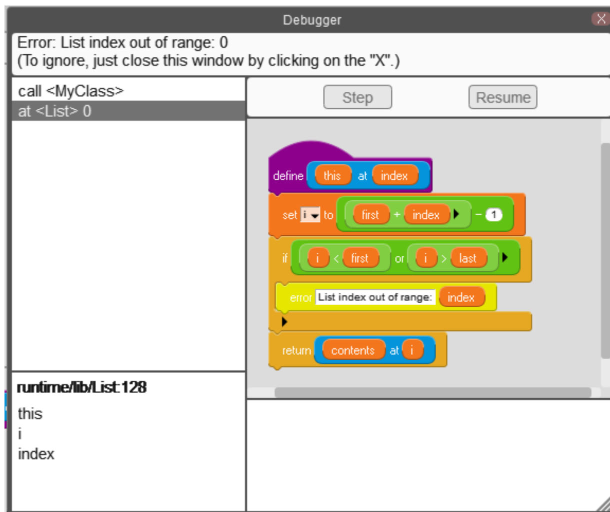




Um diese Daten zu sortieren, wählen wir die jeweils kleinste Zahl aus der Liste der unsortierten Zahlen aus und löschen sie dort. Dazu erzeugen wir für den Block drei Skriptvariable *min, positionVonMin* und *i* (als Zählvariable).

Zur Sicherheit suchen wir nur dann das Minimum, wenn wirklich Zahlen vorhanden sind. Falls wir auf nicht vorhandenen Listenpositionen zugreifen, öffnet sich das Debuggerfenster von GP. Die Fehlermeldung steht oben im Fenster, der den Fehler verursachende Block ist gelb hervorgehoben. Durch schrittweises Weitermachen kann der Fehler lokalisiert werden.

```
define suche die kleinste Zahl der Liste this liste und lösche sie dort
let min be 0
let positionVonMin be 0
let i be 0
if count liste == 0
  return nix
else if
  set min to liste at 1
  set positionVonMin to 1
  set i to 2
  while i <= count liste
    if liste at i < min
      set min to liste at i
      set positionVonMin to i
    increase i by 1
  liste remove min
  return min
```



Damit sind wir fast fertig. Das Sortieren wird durchgeführt, indem die Elemente der Liste *sortiert* gelöscht werden und dann wiederholt die kleinste Zahl aus der unsortierten Liste in die mit den sortieren Elementen überführt wird.

```
define sortiere die Elemente der Liste this unsortiert in die Liste sortiert
while count sortiert > 0
  sortiert remove first sortiert
repeat count unsortiert
  sortiert add suche die kleinste Zahl der Liste this unsortiert und lösche sie dort
```

Der Block `sortiere die Elemente der Liste this shared daten in die Liste shared sortierteDaten` überführt dann die Situation

daten	(list 36 39 48 59 8 55 46 10 70 78)
sortierteDaten	(list)

in

daten	(list)
sortierteDaten	(list 8 10 36 39 46 48 55 59 70 78)

8.2 Sortieren mit Listen – Quicksort

Als zweites, rekursives, Beispiel wollen wir Quicksort in der gleichen Umgebung wie oben realisieren.[10] Als Pivot-Element wählen wir das mittlere der jeweiligen Teilliste.

Quicksort wird gestartet, indem die zu sortierende Liste angegeben wird:

```
define quicksort this liste
    teile und ordne this die Liste liste zwischen 1 und count liste
```

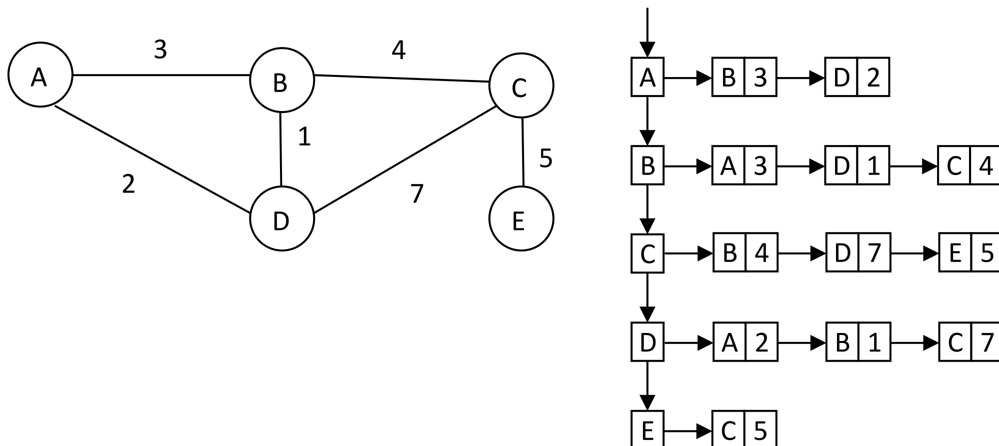
```
define teile und ordne this die Liste liste zwischen links und rechts
    let l be links
    let r be rechts
    let pivot be liste at round ((links + rechts) / 2)
    let h be 0
    while l <= r
        while liste at l < pivot
            increase l by 1
        while liste at r > pivot
            increase r by -1
        if r >= l
            set h to liste at l
            liste at l put liste at r
            liste at r put h
            increase l by 1
            increase r by -1
    if links < r
        teile und ordne this die Liste liste zwischen links und r
    if rechts > l
        teile und ordne this die Liste liste zwischen l und rechts
```

Die eigentliche Arbeit erfolgt im Block *teile und ordne die Liste <liste> zwischen <links> und <rechts>.*

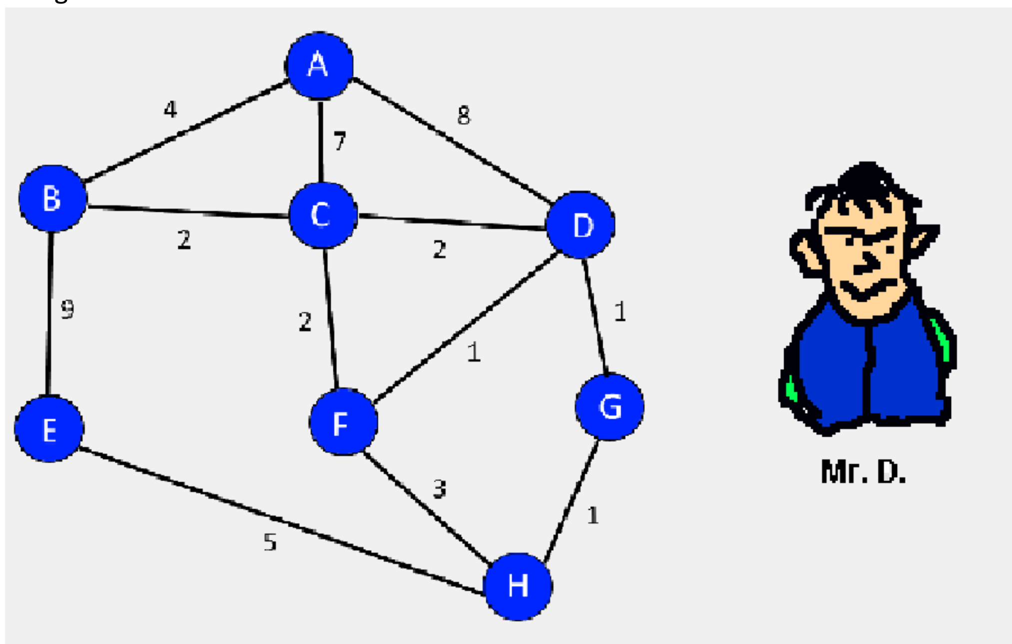
[10] Das Verfahren findet sich in diversen Versionen im Internet, z. B. unter http://de.wikipedia.org/wiki/Quicksort. Hier wurde eine In-place-implementierung gewählt.

8.3 Kürzeste Wege mit dem Dijstra-Verfahren

Gegeben sei ein Graph durch eine *Adjazenzliste*. In dieser sind alle Knoten des Graphen aufgeführt, von denen jeweils Listen „abgehen“, in die die Nachbarknoten mit den jeweiligen Entfernungen eingetragen sind: also diejenigen Knoten, zu denen eine direkte Verbindung existiert. Als Beispiele werden ein sehr einfacher Graph und seine Adjazenzliste angegeben.



Zur Bearbeitung des Problems benötigen wir natürlich einen Spezialisten: wir zeichnen Mr.D. Dieser muss in der Lage sein, die Adjazenzliste eines gegebenen Graphen zu erzeugen. Den Graphen zeichnen wir einfach auf den Hintergrund – hier sehr geschmackvoll geschehen.



Die Liste erzeugen wir statisch durch Einfügen der entsprechenden Elemente in eine lokale Liste, die wir als Ergebnis der Operation zurückgeben. Da wir an dieser Stelle zur Abwechslung mal wieder globale Methoden (*Shared Blocks*) verwenden wollen, schalten wir in den *Developer*-Mode um, denn nur dann finden wir die Blöcke für Skriptvariablen in der *Variables*-Palette.

Knoten und Kanten als Teillisten in andere Liste eintragen

```
define neue Adjazenzliste
let a be list
a add list A list list B 4 list C 7 list D 8
a add list B list list A 4 list C 2 list E 9
a add
  list C list list A 7 list B 2 list D 2 list F 2
a add
  list D list list A 8 list C 2 list F 1 list G 1
a add list E list list B 9 list H 5
a add list F list list C 2 list D 1 list H 3
a add list G list list D 1 list H 1
a add list H list list E 5 list F 3 list G 1
return a
```

Die globale Variable adjazenzliste erhält dann diese Werte über eine einfache Zuweisung: `set shared adjazenzliste to neue Adjazenzliste`

Zur weiteren Bearbeitung benötigen wir noch drei andere Listen:

- Die Liste der offeneTupel nimmt Tupel auf, die den Namen des Knotens, seine Gesamtentfernung vom Startknoten und den Namen des Vorgängerknotens enthalten.
- Die Liste entfernungen nimmt Tupel auf, die den Namen des Knotens und seine Gesamtentfernung vom Startknoten enthalten. Die Liste wird bei Neueintragungen jeweils neu sortiert, sodass der Knoten mit der kürzesten Entfernung vom Start vorne steht.
- Die Liste fertigeKnoten enthält die Namen der Knoten, die bereits fertig bearbeitet wurden.

Die Einrichtung dieser Listen für den Start fassen wir in einer Methode vorbereitung zusammen, der auch der Name des Startknotens übergeben wird. (Das Löschen aller Listenelemente wurde schon weiter oben gezeigt.) Nach ihrem Aufruf ergibt sich das folgende Bild.

```
define vorbereitung startknoten
löscheAlleElementeAus shared offeneTupel
löscheAlleElementeAus shared fertigeKnoten
löscheAlleElementeAus shared entfernungen
shared offeneTupel add list startknoten 0 -
```

```
offeneTupel (list (list A 0 -))
fertigeKnoten (list)
entfernungen (list)
```

Der Zustand zu Beginn der Suche.

Die Wegsuche ist in dieser Version sehr einfach, da der größte Teil der „Intelligenz“ in den Umgang mit den Listen gesteckt wurde.

```
define Routing von (von) nach (nach)
set shared [adjazenzliste] to (neue Adjazenzliste)
vorbereitung (von)
repeat (count (shared [adjazenzliste]))
    schritt
zeigeErgebnis (nach)
```

Die eigentliche Verarbeitung erfolgt damit in der Methode *schritt*:

```
define schritt
let (nachbarn) be []
let (aktuellesTupel) be []
let (aktuellerKnoten) be []
let (dist) be []
let (nachbar) be []
let (i) be (0)
let (aktuellerIndex) be (0)
if (count (shared [offeneTupel])) != (0)
    set [aktuellesTupel] to (shared [offeneTupel]) at (1)
    shared [offeneTupel] at (1) remove
    set [aktuellerKnoten] to (aktuellesTupel) at (1)
    set [dist] to (aktuellesTupel) at (2)
    set [aktuellerIndex] to
        ((to number (first (text to codes (aktuellerKnoten))))
         - (to number (first (text to codes [A])))) + (1)
    set [nachbarn] to ((shared [adjazenzliste]) at (aktuellerIndex)) at (2)
    shared [fertigeKnoten] add (aktuellerKnoten)
    shared [entfernungen] add (list (aktuellerKnoten) (dist))
    set [i] to (1)
    repeat (count (nachbarn))
        set [nachbar] to (nachbarn) at (i)
        if not ((shared [fertigeKnoten]) contains ((nachbar) at (1)))
            shared [offeneTupel] add
                (list ((nachbar) at (1)) (((nachbar) at (2)) + (dist)) (aktuellerKnoten))
        increase [i] by (1)
sortiereOffeneTupel
entferneDoppelteTupel
```

lokale Variable vereinbaren

das Tupel mit der bisher kleinsten Entfernung bearbeiten

Knotennamen und Entfernung auslesen und Index in der Adjazenzliste aus dem Namen berechnen

die Nachbarn des Knotens auslesen

Knoten als bearbeitet markieren und Entfernung zum Startknoten merken

alle unbearbeiteten Nachbarn mit neuer Gesamtentfernung und Vorgängerknoten in offeneTupel eintragen

offeneTupel nach der Entfernung sortieren und Tupel mit größeren Entfernungen löschen

Wie man sortiert, haben wir weiter oben gesehen. Hier geschieht es durch Auswahl des Kleinsten.

```
define sortiereOffeneTupel
let sortierteTupel be list
let i be 0
let min be 0
let pos be 0
repeat count shared offeneTupel
    set min to shared offeneTupel at 1 at 2
    set pos to 1
    set i to 2
    repeat count shared offeneTupel − 1
        if shared offeneTupel at i at 2 < min
            set min to shared offeneTupel at i at 2
            set pos to i
        increase i by 1
    sortierteTupel add shared offeneTupel at pos
    shared offeneTupel at pos remove
löscheAlleElementeAus shared offeneTupel
repeat count sortierteTupel
    shared offeneTupel add sortierteTupel at 1
    sortierteTupel at 1 remove
```

die Liste sortierteTupel nimmt die sortierten Tupel auf

Annahme, dass die kleinste Entfernung ganz vorne steht.

ggf. noch kleinere Entfernungen finden

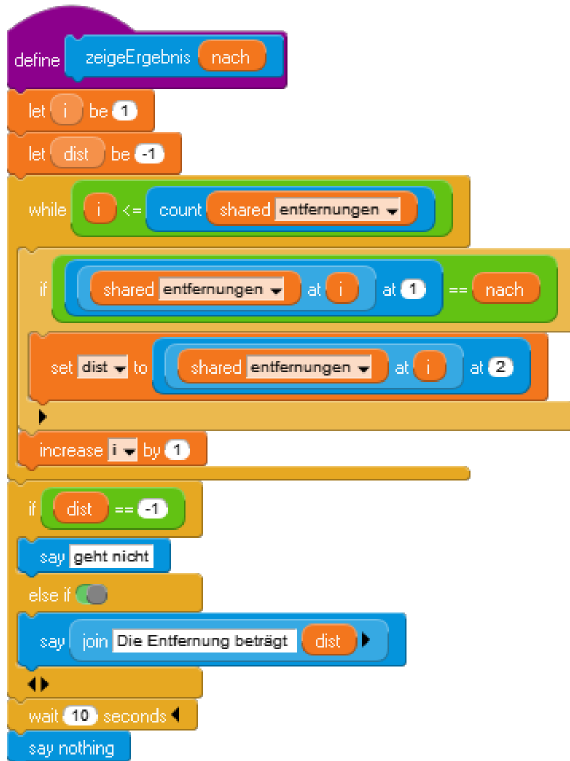
das Tupel mit der kleinsten Entfernung zu sortierteTupel hinzufügen und in offeneTupel löschen

zuletzt die sortierte Liste zurück kopieren

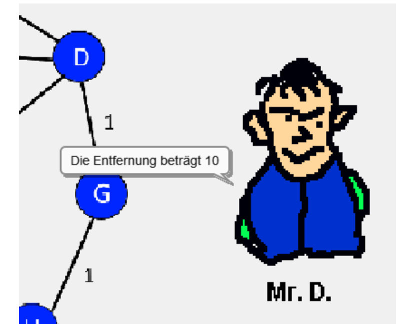
```
define entferneDoppelteTupel
let k be 0
let i be 1
let j be 0
while i < count shared offeneTupel
    set k to shared offeneTupel at i at 1
    set j to i + 1
    while j <= count shared offeneTupel
        if shared offeneTupel at i at 1 == k
            shared offeneTupel at j remove
        else if
            increase j by 1
    increase i by 1
```

Jetzt steht für jeden Knoten das Tupel mit der kleinsten Entfernung vorne in der Liste. Falls noch andere Tupel für diesen Knoten auftreten, werden sie gelöscht.

Zuletzt müssen wir nur noch die Entfernung zum gesuchten Knoten aus der Liste entfernungen heraussuchen und von Mr.D. anzeigen lassen.



Mr.D. kriegt es raus!



8.4 Matrizen und Tabellen

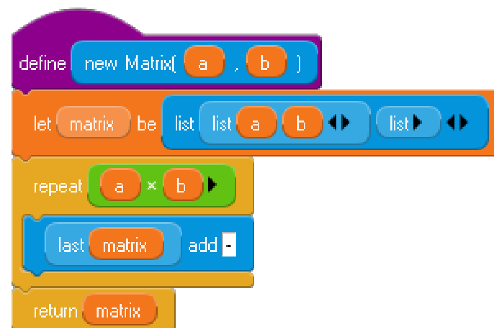
Wenn wir Listen mit direktem Zugriff auf jedes Element haben, dann benötigen wir eigentlich keine speziellen Reihungen, Stapel, Schlangen usw. Alle höheren Datenstrukturen lassen sich aus Listen aufbauen. Wir basteln uns die Datenstruktur *matrix*, weil sie traditionell z. B. bei den Adjazenzmatrizen Verwendung findet. Zur Darstellung von Matrizen benutzen wir Tabellen (aus der *Table*-Palette, nur im Developer-Modus). *(Achtung: der Kürze halber verzichten wir auf alle Sicherheitsabfragen!)*

Wir verpacken eine Matrix natürlich in einer Liste. Dafür vereinbaren wir (willkürlich) die folgende Listenstruktur:

[[Liste mit Größe der Indexbereiche][Liste mit Daten.........]]

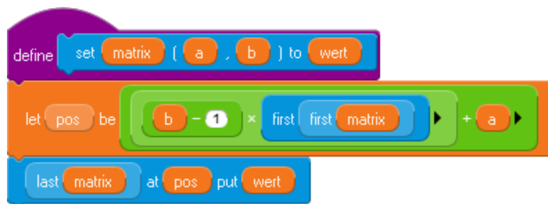
Die Dimension der Matrix ergibt sich dann direkt aus den Einträgen der ersten Teilliste. Als Beispiel wählen wir eine zweidimensionale Reihung mit jeweils zwei Werten pro Zeile. Sie hätte die folgende Struktur: [[2,3][1,2,3,4,5,6]]

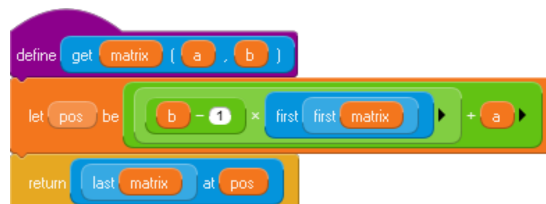
Wir legen wir eine zweidimensionale Matrix der Größe *a x b* an, indem wir die beiden gewünschten Listen erzeugen. Die erste enthält die beiden übergebenen Parameter, die zweite soll als leer gekennzeichnet sein, z. B. durch ein Minuszeichen. Das Ergebnis geben wir zurück. Wir benutzen wieder globale Methoden.

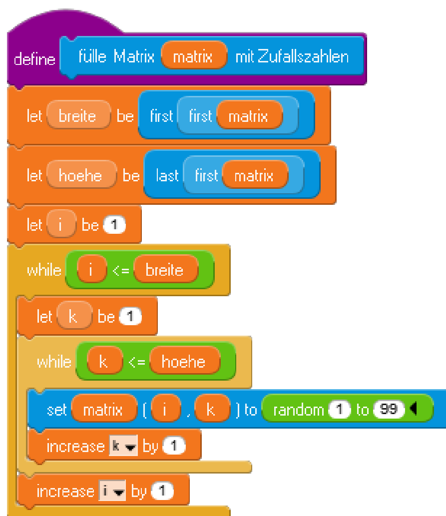


Die Syntax kann völlig frei gewählt werden, zum Beispiel auch mit Klammern, wenn man das mag!

Jetzt schreiben wir mit *set* Werte in die Matrix, schön übersichtlich. Wir holen uns zuerst die Dimensionen und bestimmen die Breite der Matrix. Dann berechnen wir den Platz der zu ändernden Stelle und überschreiben den entsprechenden Listeneintrag. Zum Lesen von Matrixeinträgen dient die Methode *get*.

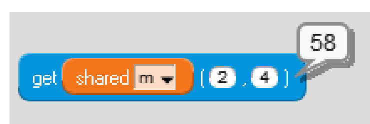


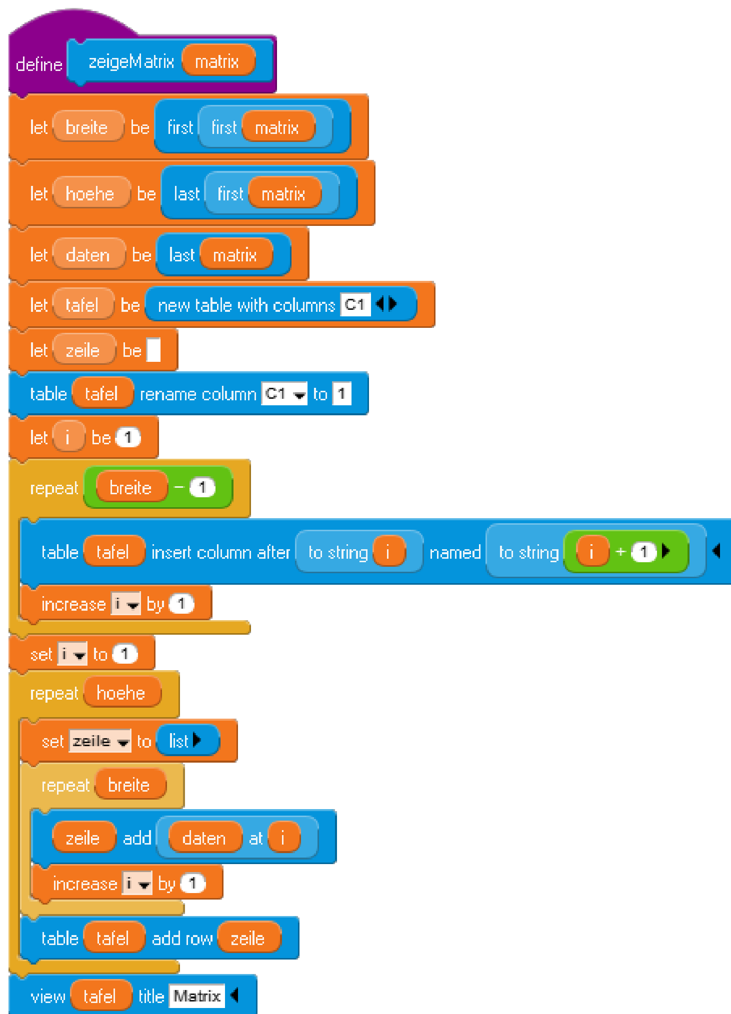




Mit diesen Methoden kann man schon arbeiten, z. B. um eine Tabelle mit Zufallszahlen zu füllen.

Eingetragene Wert sind auch zugänglich:





Etwas aufwändiger ist die Anzeige einer Matrix. Wir benutzen dafür eine Tabelle, deren Spalten wir durchnummerieren. Danach fügen wir die Zeilen der Matrix als Tabellenzeilen an. Zuletzt wird das Ergebnis angezeigt.

Der Aufruf dieser Methode lässt die Tabelle etwa in der Mitte des Bildschirms erscheinen.

zeigeMatrix shared m

Matrix

4	1	2	3
1	71	99	33
2	74	29	36
3	90	52	73
4	79	58	1

8.5 Aufgaben

1. Informieren Sie sich im Netz über die verschiedenen Sortierverfahren. Implementieren Sie einige davon wie Shakersort, Gnomsort, Insertionsort, …

2. Ergänzen Sie die angegebenen Methoden so, dass Fehleingaben abgefangen werden.

3. Implementieren Sie Matrizen anders, indem Sie die verwendeten Listen anders strukturieren.

4. a: Informieren Sie sich über die Datenstruktur dictionary.
   b: Implementieren Sie die Struktur mit geeigneten Zugriffsoperationen.

5. a: Implementieren Sie die Datenstruktur Stapel.
   b: Implementieren Sie die Datenstruktur Schlange.

6. Implementieren Sie einen einfachen Binärbaum mit den Operationen
   a: new Baum
   b: rein <element> in <baum>
   c: zähle die Elemente in <baum>
   d: ist <element> vorhanden in <baum>?
   e: raus <element> aus <baum>
   f: ermittle die maximale Tiefe von <baum>
   g: balanciere <baum> aus

9 Im Netz arbeiten

GP kann ebenso wie *snap!* http-Aufrufe absenden, und damit steht dieselbe auf diesem Block fußende Funktionalität zur Verfügung. Als Beispiel wählen wir den Server *snapextensions.uni-goettingen.de*, auf dem sich eine Reihe frei zugänglicher Datenbanken befinden, die vom Benutzer *snapexuser* mit dem Password *snap!user* abgefragt werden können. Dafür wollen wir eine Klasse *SQLconnector* schreiben, die den Zugriff auf diesen Server erlaubt, und eine weitere Klasse *SQL-access*, in der die Funktionalität genutzt wird. Danach werden JSON-Datenbanken verwandt, wobei die gelesenen Daten in Tabellenform ausgegeben werden. Es folgen ein einfaches Chat-Programm und der Zugriff auf ein Sensorboard.

9.1 SQL-Datenbanken verwenden

Wir erzeugen eine neue Klasse *SQLconnector*, deren Blöcke wir später in die Palette *SQL* exportieren. Für diese legen wir eine Reihe von Instanzvariablen an (s. Bild). Um diesen einen anfänglich akzeptablen Zustand zuzuweisen, erstellen wir die Methode *clearData*.

```
define clearData this
comment clear all instance variables
set my connectionOK to (false)
set my lastSQLmessage to nil
set my theServer to nil
set my theUser to nil
set my thePassword to nil
set my theDatabases to list
set my theTables to list
set my theColumns to list
set my theCurrentDatabase to nil
set my theCurrentTable to nil
set my theServerresponse to nil
set my theQuery to nil
set my theAnswer to nil
```

Instance Variables	
true	my connectionOK
ok	my lastSQLmessage
snapextensions.u	my theServer
snapexuser	my theUser
snap!user	my thePassword
(list snapex_exam	my theDatabases
(list Aufgabe1 Auf	my theTables
(list pnr label shap	my theColumns
fruits	my theCurrentTable
snapex_example	my theCurrentDatabase
HTTP/1.1 200 OK	my theServerresponse
SELECT|label,pric	my theQuery
(list red apple,1,ro	my theAnswer

Zum Arbeiten benötigen wir eine funktionierende Verbindung. Diese stellen wir mithilfe der Methode *set mySQLconnection*, der wir die Parameter *server*, *user* und *password* übergeben. Diese versucht, den Server zu erreichen (s. u.), bereitet die Antwort etwas auf und sieht nach, ob alles in Ordnung ist. Im Erfolgsfall setzt sie die entsprechenden Variablen auf „positive" Werte.

```
define set mySQLconnection this with server server user user pwd password
comment set connection and user data
set my theServer to server
set my theUser to user
set my thePassword to password
call server this with command connect
let answer be lines my theServerresponse
set answer to answer at (count answer) - 1
if answer == ok
    set my connectionOK to (true)
    set my lastSQLmessage to answer
else if (false)
    clearData this
```

Wie ruft man einen Server? Dafür muss man wissen, welche Anweisungen der Server „versteht“. Unser reagiert auf die Befehle *connect*, *getDBs*, *useDB*, *getTables*, *getColumns* und *answerQuery*, was bedeutet, dass er (in diesem Fall mit PHP) diese Anweisungen umsetzt und die etwas aufbereiteten *mySQL*-Antworten zurücksendet. Unsere Methode *call server* setzt dafür einen Anfragestring syntaktisch richtig zusammen, sendet diesen an den Server und speichert die Antwort in der Variablen *theServerresponse*. Diese Antwort können wir uns bei Bedarf auch „von außen“ mithilfe der Methode *get answer* ansehen.

```
define get answer of this
  return my theServerresponse
```

Entsprechendes gilt für drei andere, eventuell „interessante“ Variableninhalte.

```
define get last SQL-message this
  return my lastSQLmessage

define get connectioOK this
  return my connectionOK

define get last query this
  return my theQuery
```

Jetzt möchten wir natürlich wissen, welche Datenbanken auf dem Server zugänglich sind. Das erreichen wir mit der Methode *get databases*. Diese prüft, ob eine Verbindung zum Server besteht, übermittelt diesem den entsprechenden Befehl und bereitet die Antwort so auf, dass „überflüssige“ Zeilen gar nicht erst erscheinen. Die erreichbaren Datenbanknamen schreibt sie in eine Liste, die sofort als Tabelle angezeigt wird.

```
define call server this with command command
  comment HTTP request to server. The answer is stored in "theServerresponse"
  let message be
  if command == connect
    set message to join /mysqlquery.php?type=connect&server= my theServer &user= my theUser
        &password= my thePassword
  else if command == getDBs
    set message to join /mysqlquery.php?type=getDBs&server= my theServer &user= my theUser
        &password= my thePassword
  else if command == useDB
    set message to join /mysqlquery.php?type=useDB&command=USE| my theCurrentDatabase &server=
        my theServer &user= my theUser &password= my thePassword
  else if command == getTables
    set message to join /mysqlquery.php?type=getTables&command=SHOW|TABLES|FROM|
        my theCurrentDatabase &server= my theServer &user= my theUser &password=
        my thePassword
  else if command == getColumns
    set message to join /mysqlquery.php?type=getColumns&command=SHOW|COLUMNS|FROM|
        my theCurrentTable &database= my theCurrentDatabase &server= my theServer
        &user= my theUser &password= my thePassword
  else if command == answerQuery
    set message to join /mysqlquery.php?type=query&query= my theQuery &database=
        my theCurrentDatabase &server= my theServer &user= my theUser &password=
        my thePassword
  set my theServerresponse to http host my theServer path message

define get databases this
  comment reads the list of databases
  if not my connectionOK
    set my lastSQLmessage to ERROR: connect to the server first
  else if
    call server this with command getDBs
    let answer be lines my theServerresponse
    set my theDatabases to list
    for i in range from 11 to count answer - 1
      my theDatabases add answer at i
    show lists as table with headline available databases , column titles list Databases
        and row values my theDatabases
```

Das letztere geschieht mit der Methode *show lists as table*.

Diese ist ziemlich misstrauisch und sieht sich deshalb den Typ der übermittelten Parameter genau an. Gefällt ihr der, so legt sie eine neue Tabelle an, der Spalten mit den angegebenen Überschriften angefügt werden.

In diese Struktur werden die Tabellenzeilen mit den übergebenen Daten eingefügt. Da die Zeilendaten mit Kommas getrennt übergeben werden, splittet sie zuerst die Zeile auf und fügt die erhaltenen Listenelemente aneinander.

Zuletzt wird die Tabelle am Bildschirm angezeigt.

Lassen wir jetzt die folgende Befehlsfolge ausführen, dann erhalten wir:

Die folgenden drei Methoden *use database*, *get tables* und *get Columns* sind entsprechend aufgebaut und sollten selbsterklärend sein.

```
define use database (this) no. (n)
comment [chooses a database]
if not (my connectionOK)
  set my [lastSQLmessage] to [ERROR: connect to the server first]
else if ((count (my theDatabases)) < 1)
  set my [lastSQLmessage] to [ERROR: read databases from the server first]
else if (((n) < 1) or ((n) > (count (my theDatabases))))
  set my [lastSQLmessage] to [ERROR: wrong number]
else if
  set my [theCurrentDatabase] to ((my theDatabases) at (n))
  call server (this) with command [useDB]
```

```
define get tables (this)
comment [reads the list of tables]
if not (my connectionOK)
  set my [lastSQLmessage] to [ERROR: connect to the server first]
else if ((my theCurrentDatabase) == nil)
  set my [lastSQLmessage] to [ERROR: choose database first]
else if
  call server (this) with command [getTables]
  let (answer) be (lines (my theServerresponse))
  set my [theTables] to (list)
  for (i) in (range from 11 to ((count (answer)) - 1))
    (my theTables) add ((answer) at (i))
  show lists as table with headline [tables of] , column titles (list (my theCurrentDatabase)) and row values (my theTables)
```

```
define get columns of (this) (table)
comment [reads the list of columns]
if not (my connectionOK)
  set my [lastSQLmessage] to [ERROR: connect to the server first]
else if ((my theCurrentDatabase) == nil)
  set my [lastSQLmessage] to [ERROR: choose database first]
else if not (((my theTables) find (table)) > 0)
  set my [lastSQLmessage] to [ERROR: wrong tablename]
else if
  set my [theCurrentTable] to (table)
  call server (this) with command [getColumns]
  let (answer) be (lines (my theServerresponse))
  set my [theColumns] to (list)
  for (i) in (range from 11 to ((count (answer)) - 1))
    (my theColumns) add ((answer) at (i))
  show lists as table with headline [columns of] , column titles (list (table)) and row values (my theColumns)
```

Jetzt kommen wir zum eigentlichen Sinn dieses Abschnitts: dem Stellen von Anfragen an die Datenbank. Im einfachsten Fall geschieht dieses durch eine simple SQL-Anfrage wie *SELECT label, price, shape FROM fruits, prices.* Diese muss ausgewertet werden.

```
define SELECT this attributes FROM tables WHERE predicate
comment simple SELECT query
let atts be delete blanks from this attributes
let tbls be delete blanks from this tables
if not my connectionOK
  set my lastSQLmessage to ERROR: connect to the server first
else if my theCurrentDatabase == nil
  set my lastSQLmessage to ERROR: choose database first
else if
  set my theQuery to
    replace this   with | in
      join SELECT| atts |FROM| tbls |WHERE| predicate ;
  call server this with command answerQuery
  let answer be lines my theServerresponse
  set my theAnswer to list
  for i in range from 11 to count answer - 1
    my theAnswer add answer at i
  if atts == *
    show lists as table with headline answer , column titles my theColumns and row values my theAnswer
  else if
    show lists as table with headline answer , column titles to list splitWith to string attributes , and row values my theAnswer
```

Überflüssige Leerzeichen werden entfernt.

Die Verbindung wird überprüft.

Die Anfrage wird aufbereitet, weil GP in der aktuellen Version Probleme mit Leerzeichen im Sendestring hat, und abgeschickt.

Die Antwort wird aufbereitet …

… und als Tabelle angezeigt.

answer

240	label	price	shape
1	red apple	1	round
2	green apple	1	round
3	tomato	1	round
4	orange	1	round
5	apricot	1	oval
6	banana	1	long
7	cherry	1	round
8	cucumber	1	long
9	green grape	1	round
10	blue grape	1	round
11	aubergine	1	oval
12	plum	1	oval
13	asparagus	1	long
14	blackberry	1	round
15	black radish	1	round
16	red apple	2	round
17	green apple	2	round

Die angezeigte Tabelle.

Für etwas differenziertere Anfragen steht ein erweiterter Block zur Verfügung.

```
define
SELECT (this) (attributes) FROM (tables) WHERE (whereclause) GROUP BY (groupattribute) HAVING (havingclause) ORDER BY (orderattribute) (direction) LIMIT (limit)
comment (almost) full SELECT query
let (atts) be delete blanks from (this) (attributes)
let (tbls) be delete blanks from (this) (tables)
if not my connectionOK
  set my lastSQLmessage to ERROR: connect to the server first
else if my theCurrentDatabase == nil
  set my lastSQLmessage to ERROR: choose database first
else if
  set my theQuery to
    join SELECT| atts |FROM| tbls |WHERE| whereclause
  if groupattribute !=
    set my theQuery to join my theQuery |GROUP|BY| groupattribute
  if havingclause !=
    set my theQuery to join my theQuery |HAVING| havingclause
  if orderattribute !=
    set my theQuery to join my theQuery |ORDER|BY| orderattribute
    if direction == DESC
      set my theQuery to join my theQuery |DESC
    else if
      set my theQuery to join my theQuery |ASC
  set my theQuery to join my theQuery |LIMIT| limit ;
  call server (this) with command answerQuery
  let (answer) be lines my theServerresponse
  set my theAnswer to list
  for (i) in range from 11 to count answer - 1
    my theAnswer add answer at i
  show lists as table with headline answer , column titles
    to list splitWith to string attributes ,  and row values my theAnswer
```

```
SELECT shared theSQLconnector label,shape,price FROM fruits,prices WHERE
true GROUP BY HAVING ORDER BY ASC LIMIT 10
```

answer

10	label	shape	price
1	red apple	round	1
2	green apple	round	1
3	tomato	round	1
4	orange	round	1
5	apricot	oval	1
6	banana	long	1
7	cherry	round	1
8	cucumber	long	1
9	green grape	round	1
10	blue grape	round	1

Weitere Blocks etwa zum Anlegen oder Löschen von Tabellen, dem Update von Daten usw. lassen sich leicht nach dem gleichen Muster anlegen – wenn man die entsprechenden Zugriffsrechte hat. Das ist auf dem angegebenen Server aber nicht der Fall.

9.2 JSON-Datenbanken

Im Internet finden wir gerade im Bereich der *Open Data* für Schulen recht interessanten Datensammlungen, die oft in der *JavaScript Object Notation* (*JSON*) vorliegen. Als Beispiel soll hier eine Liste der New Yorker Fahrrad-Entleihstationen dienen (https://catalog.data.gov/dataset/citi-bike-live-station-feed-json-d1c27).

JSON-Dateien enthalten entweder *atomare Werte* (Wahrheitswerte, Zahlen, …), *Zeichenketten*, *Arrays* (begrenzt von eckigen Klammern) oder *Objekte* (begrenzt von geschweiften Klammern). Da wir hier nur mit Listen arbeiten, übersetzen wir solche Dateien in eine Struktur, die entweder eine atomare Größe (Wahrheitswert, Zahl, Zeichenkette, …) oder eine Liste enthält, die aus atomaren Größen und/oder Teillisten besteht, die als ersten Eintrag den Typ der originalen Daten (Liste oder Dictionary) enthalten. In Dictionaries folgen als weitere Elemente zweielementige Listen mit Schlüssel/Wert-Paaren. Im Beispiel wird aus den Daten eine Tabelle erzeugt, die nur die Spalten *Stationsname*, *Status* und *verfügbare Fahrräder* enthält. Dabei machen wir von GPs Fähigkeit Gebrauch, JSON-Dateien zu kodieren und zu dekodieren (im Developer-Mode im der *Network*-Palette). In Abhängigkeit vom ersten Zeichen entscheidet die Methode *readJSONdata*, welche Art von Liste erzeugt wird.

```
define readJSONdata JSON
  let data be to string JSON
  let JSONtest be json decode to string JSON
  let char be " "
  if count data > 0
    set char to data at 1
    if char == [
      return readList from substring data from 2 to count data - 1
    else if char == {
      return readDictionary from substring data from 2 to count data - 1
    else if char == "
      return readAtom from substring data from 2
    else if true
      return readAtom from data
```

Am einfachsten ist es, ein Atom aus der Zeichenkette zu lesen. Wenn kein Begrenzungszeichen für die anderen Typen kommt, werden die Zeichen einfach aneinandergehängt.

```
define readAtom from vomString
let data be letters join to string vomString #
let atom be
let i be 1
let length be count data
let char be data at i
while i <= length and not list [ ] { } , : # " contains char
    set atom to join atom char
    increase i by 1
    set char to data at i
return atom
```

Komplizierter ist es mit den Arrays, die hier in Listen übersetzt werden. Es können in den Listen wieder Listen, atomare Größen oder Objekte auftreten, die bunt gemischt sind. Für solche Zwecke setzt man Rekursionen ein, in diesem Fall indirekt über *readJSONdata*.

```
define readList from vomString
let result be list Liste
let data be letters join to string vomString #
let item be
let brackets be 0
let i be 1
let length be count data
let char be data at i
while i < length
    set item to
    while i < length and not char == , and brackets == 0
        if char == { or char == [
            increase brackets by 1
        else if char == } or char == ]
            increase brackets by -1
        set item to join item char
        increase i by 1
        set char to data at i
    result add readJSONdata item
    if char == ,
        increase i by 1
        set char to data at i
return result
```

Mit den Objekten verfährt man genauso …→:

… und zuletzt wird solche eine Struktur wieder als Tabelle dargestellt ↓ .

```
define makeTable
let tabledata be (shared theList at 3 at 2)
let i be 2
let nextItem be
let nextStation be 0
let j be 0
let data be
let key be
set shared table to new table with columns station name, status value, available bikes
while i <= count tabledata
    set nextStation to tabledata at i
    set j to 2
    set data to list
    while j <= count nextStation
        set nextItem to nextStation at j
        set key to nextItem at 1
        if key == "stationName" or key == "availableBikes" or key == "statusValue"
            data add nextItem at 2
        increase j by 1
    table shared table add row data
    increase i by 1
view shared table
```

Table

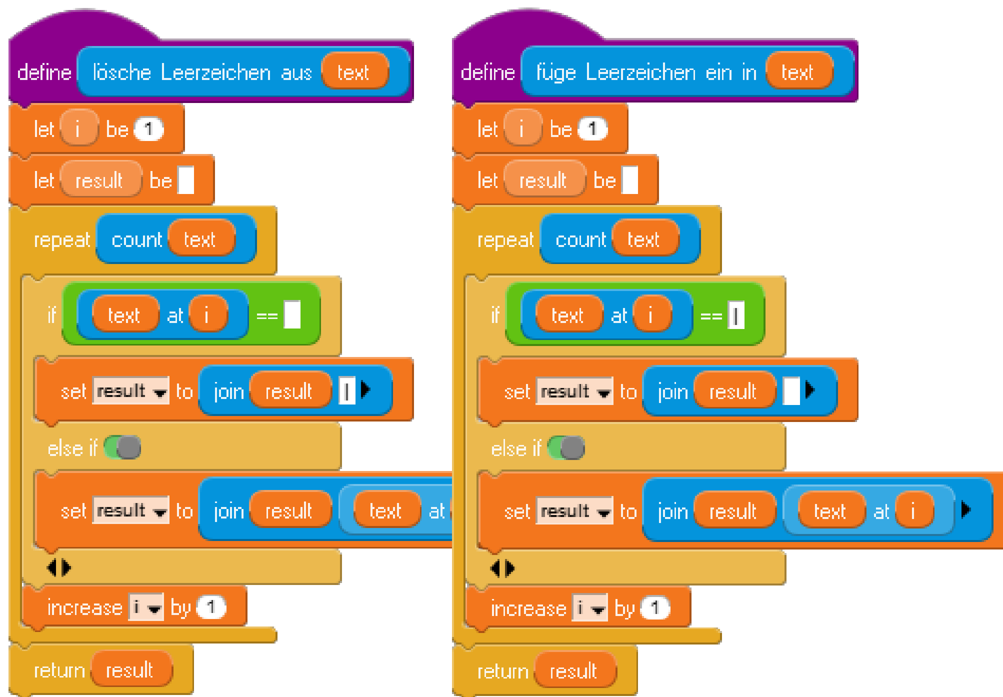
666	station name	status value	available bikes
649	West Drive & Pros	In Service	20
650	E 106 St & Lexingt	In Service	6
651	2 Ave & E 104 St	In Service	1
652	Lafayette St & Jers	In Service	19
653	8 Ave & W 16 St	In Service	14
654	Hanson Pl & Ashla	In Service	28
655	Richardson St & N	In Service	0
656	E 35 St & 3 Ave	In Service	0
657	W 88 St & West Er	In Service	1
658	Grand St & Elizabe	In Service	24
659	Greenwich St & Hu	In Service	18
660	Riverside Dr & W S	Not In Service	0
661	E 76 St & 3 Ave	In Service	1
662	Fulton St & Adams	In Service	15
663	W 52 St & 6 Ave	In Service	20

```
define readDictionary from vomString
let result be list Dictionary
let data be letters (join (to string vomString) #)
let key be
let value be
let brackets be 0
let i be 1
let length be count data
let char be data at i
while i < length
    set key to
    while i < length and not (char == : and brackets == 0)
        if char == { or char == [
            increase brackets by 1
        else if char == } or char == ]
            increase brackets by -1
        set key to join key char
        increase i by 1
        set char to data at i
    if char == :
        increase i by 1
        set char to data at i
    set value to
    let brackets be 0
    while i < length and not (char == , and brackets == 0)
        if char == { or char == [
            increase brackets by 1
        else if char == } or char == ]
            increase brackets by -1
        set value to join value char
        increase i by 1
        set char to data at i
    if char == ,
        increase i by 1
        set char to data at i
    result add list key (readJSONdata value)
return result
```

9.3 Chatten

Wir gehen davon aus, Zugang zu einem Server zu haben, der uns den Zugriff auf eine einfache Textdatei gestattet. In diese dürfen wir eine Textzeile schreiben und/oder den Dateiinhalt lesen. Die Datei nennen wir *chat.txt*.

Hinweis: In der aktuellen Version hat GP (wie andere Systeme) Probleme, Zeichenketten über den http-Block zu senden, die Leerzeichen enthalten. Wir ersetzen deshalb diese durch senkrechte Striche (beim Senden) und umgekehrt (beim Empfangen).



Mit dieser kleinen Nothilfe können wir einfach Textzeilen auf dem Server ablegen bzw. von dort lesen.

Wir wählen die folgenden Regeln für die Kommunikation:

- Wir beschränken die Kommunikation (erstmal) auf zwei Partner. Diese lesen regelmäßig die Datei *chat.txt* auf dem Server. Enthält diese mehr als einen Gedankenstrich, dann wird der Inhalt an eine Tabelle angehängt, die den Chatverlauf darstellt. Danach wird sie gelöscht.
- Nachrichten bestehen aus dem Absendernamen, gefolgt von einem Doppelpunkt und der eigentlichen Nachricht.
- Texteingaben erfolgen nach Drücken der Leertaste, mit „0“ wird ein Chat neu gestartet, „1“ und „2“ bewirkten Kostümwechsel.

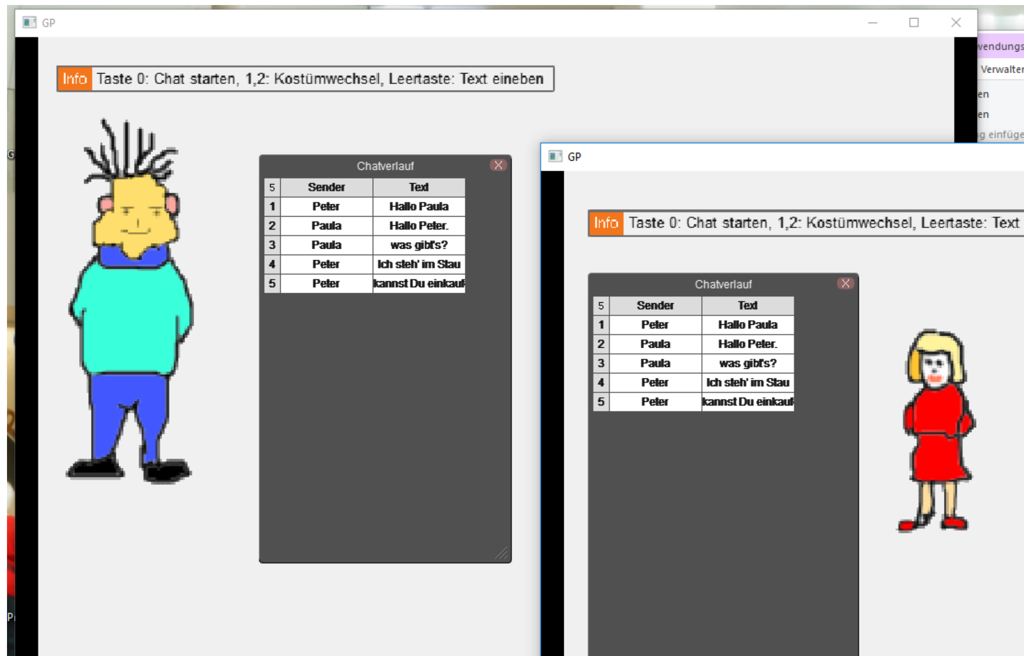
```
define starte Chat
set shared name to ask Wie heißt Du?
set shared chat to new table with columns Sender Text
view shared chat title Chatverlauf
schreibe auf Server join 192.168.2.106 den Text - in die Datei chat.txt
let antwort be
animate
  set antwort to lies vom Server join 192.168.2.106 aus der Datei chat.txt
  if antwort != -
    set shared sendung to splitWith antwort :
    if shared sendung at 1 != shared name
      table shared chat add row shared sendung
      schreibe auf Server join 192.168.2.106 den Text - in die Datei chat.txt
  wait 1 seconds
```

```
when space key pressed
let text be ask Gib den Text ein!
schreibe auf Server join 192.168.2.106 den Text
  join shared name : text in die Datei chat.txt
table shared chat add row list shared name text
```

```
when I receive go
starte Chat

when 0 key pressed
starte Chat

when 1 key pressed
set costume to Peter

when 2 key pressed
set costume to Petra
```

Wir speichern diesmal die *Chat.gpp*-Datei nicht nur, sondern exportieren sie als App, mit *Export as App*. Danach finden wir im Programmverzeichnis, in dem sich auch der Programmtext befindet, z. B. unter Windows eine ausführbare Datei *Chat.exe*. Diese starten wir zweimal und arrangieren sie so, dass jede einen Chatpartner darstellt. (Natürlich können wir sie auch auf unterschiedlichen Rechnern starten!) Danach können wir „chatten“. 😉

Ausführbare Dateien erzeugen.



9.4 Aufgaben

1. Verbessern Sie die Funktionalität des Chatprogramms, indem Sie Buttons zur Bedienung, eine Möglichkeit zum Löschen aller Texte / der letzten Zeile / ausgewählter Zeilen / … einführen.

2. Erweitern Sie das Chatprogramm um die Möglichkeit, mehrere Partner in den Chat aufzunehmen.

3. Chats sollen natürlich privat bleiben. Verschlüsseln und entschlüsseln Sie die Daten mit einem Password, sodass nur Teilnehmer/innen mit Kenntnis des Passwords die Klartexte sehen können.

4. Eine Chat-Instanz soll einen Bankautomaten darstellen, die andere die Bank.
   - a: Verteilen Sie die Funktionalität geeignet.
   - b: Diskutieren Sie Möglichkeiten zu einer sicheren Kommunikation zwischen den beiden.
   - c: Realisieren Sie das Projekt.

5. Eine Chat-Instanz stellt einen Online-Shop dar, mehrere andere die Kunden. Behandeln Sie das Problem so wie in Aufgabe 4.

6. Zwei Spieler sollen ein Spiel[11] miteinander im Netz spielen können. Behandeln Sie das Problem wie in Aufgabe 4.

7. Eine Chat-Instanz stellt einen Mailserver dar, mehrere andere die Clients. Behandeln Sie das Problem wie in Aufgabe 4 unter besonderer Berücksichtigung der Sicherheit.

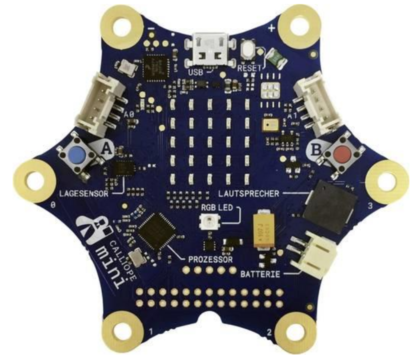
8. Entwickeln Sie eine Möglichkeit zum verteilten Rechnen im Netz. Ein Server verwaltet mehrere Clients, die arbeitsteilig ein Problem lösen können.
   - a: Informieren Sie sich im Netz über mögliche Einsatzgebiete verteilten Rechnens und entsprechende Projekte.
   - b: Diskutieren Sie die Eignung unterschiedlicher Problemklassen für diese Aufteilung. Gehen Sie dabei auch auf die funktionale Programmierung ein.
   - c: Realisieren Sie ein Projekt.

9. Mehrere Chat-Instanzen stellen ein sternförmiges Netzwerk dar mit einem einzigen Server. Die Clients können Daten entweder an Objekte der eigenen Instanz adressieren oder an Objekte in anderen Instanzen senden.
   - a: Entwickeln Sie geeignete Adressierungsarten, anhand derer die Clients entscheiden können, ob Datenpakete an eigene Objekte oder andere auszuliefern sind.
   - b: Entwickeln Sie ein entsprechendes Protokoll und realisieren Sie es.

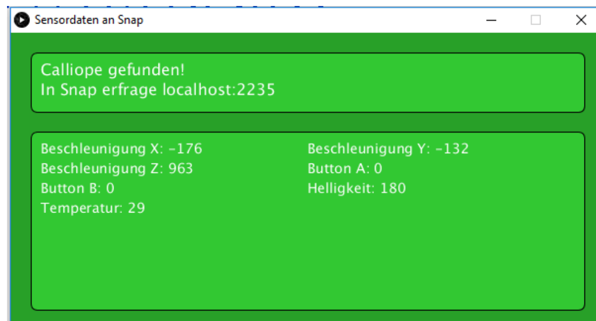
---

[11] z. B. „Schiffeversenken“

9.5 Ein Sensorboard benutzen



Wir benutzen eines handelsüblichen Sensorboards, in diesem Fall das *Calliope mini*. Für dieses gibt es ein Programm von Andreas Flemming[12], das kontinuierlich die Messwerte des Boards über einen internen Server sendet und so über das http-Protokoll auch Browseranwendungen zugänglich macht. Starten wir das Programm, dann wird nach kurzer Sucher das Calliope-Board gefunden und die Messwerte werden angezeigt.



Wir extrahieren die Messwertzeile aus dem erhaltenen Text z. B. durch

Die Messwerte stehen in der Reihenfolge *Beschleunigung in x-, -y und z-Richtung*, Zustand der Knöpfe *A* und *B* sowie *Helligkeit* und *Temperatur*, jeweils in freien Einheiten. Diese Zeichenkette können wir leicht zerlegen mit

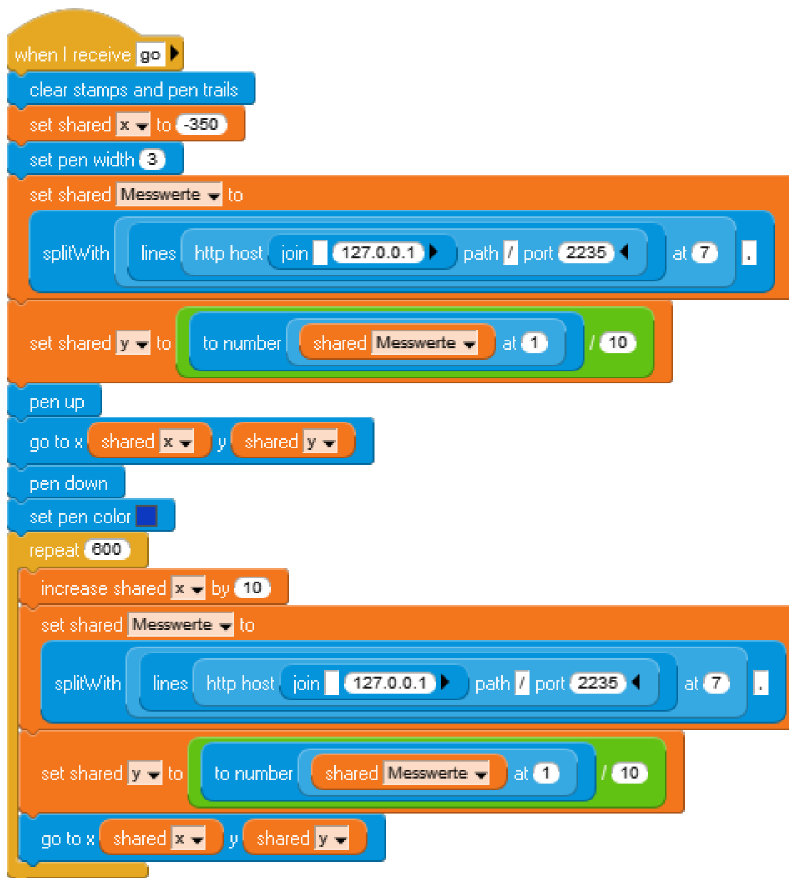
Danach sind die einzelnen Werte als Inhalte eines Arrays zugänglich.

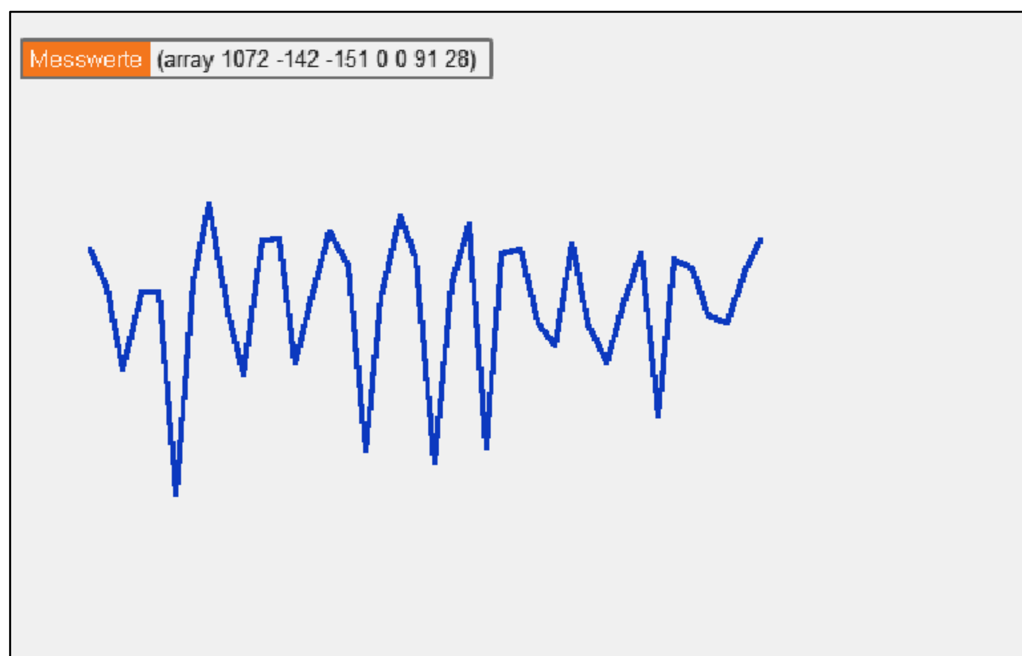
In einem kleinen Skript versuchen wir, nach einer Idee von Annika Eickhoff-Schachtebeck den Beschleunigungssensor in x-Richtung zu einem Schrittzähler[13] umzufunktionieren, wie er etwa in Smartwatches benutzt wird. Wir befestigen deshalb das Sensorboard am Arm oder am Bein und stellen die erhaltenen Messwerte grafisch dar. *(Dafür sollten wir allerdings ein genügend langes Kabel zwischen Board und Computer zur Verfügung haben!)*

---

[12] https://www.uni-goettingen.de/de/software+zur+verwendung+des+calliope+mini+mit+scratch+1.4%2c+byob+und+snap%21+%28andreas+flemming%29+download/569672.html

[13] Nach einer Idee aus https://www.uni-goettingen.de/de/unterrichtsbeispiel+fitnessarmband+%28dr.+annika+eickhoff-schachtebeck%29+download/565581.html





Das Calliope-Board als Schrittzähler.

10 Objektorientierte Programmierung

Klassenbasierte OOP

GP arbeitet natürlich die ganze Zeit mit Objekten[14], die hier *Instanzen* genannt werden. Sie verfügen über eigene Attribute (z. B. Position, Richtung, Kostüm, … oder Instanzvariable), auf die mithilfe unterschiedlicher Blöcke zugegriffen werden kann. Anders als die Vorgänger *BYOB* und *snap!* arbeitet GP mit *Klassen*, wobei zu jeder Klasse mindestens eine Instanz als Prototyp erzeugt wird. Für Klassen werden klassenspezifische Methoden entwickelt, die dann mit den Attributen der Instanzen arbeiten. Damit ähneln die *Strukturen in GP* in diesem Bereich eher denen von traditionellen OOP-Sprachen wie z. B. Java als z. B. denen von BYOB. Es gibt allerdings zwei Einschränkungen: die Autoren halten weder die Unterstützung von *Closures* noch die von *Vererbung* für sinnvoll, weil beides für die angepeilten Zielgruppen überflüssig wäre.

Das Vorhandensein mindestens einer Instanz jeder Klasse macht es möglich, dass die *Arbeit mit GP* trotzdem eher prototyporientiert ist. Obwohl meist Klassenmethoden geschrieben werden, erprobt man diese am konkreten Prototypen, überprüft das Funktionieren einzelner Blöcke und Blockkombinationen einfach durch Anklicken usw. Wird also ein „frei herumliegender“ Block angeklickt, dann arbeitet er im Namensraum der aktiven Instanz. Er muss dafür nicht in eine Methode verpackt und dann aufgerufen werden. Experimentelles Bottom-up-Arbeiten nach Liebermanns[15] Delegation-Modell wird sehr gut unterstützt.

Die hierarchische Gliederung erfolgt in GP auf mehreren Ebenen:

Drei Arten von Variablen

- Bei den *Variablen* gibt es globale Variable (*Shared Variables*), die überall zugänglich sind, Instanzvariable (*Instance Variables*), die nur innerhalb einer Instanz bekannt sind, und Skriptvariable (*Script Variables*), die man nur innerhalb des Skriptes verwenden kann, in dem sie definiert wurden. Die ersten beiden Arten können über Monitore im Arbeitsbereich beobachtet werden. Ihre Werte lassen sich aber auch in der Variables-Palette verfolgen.

Zwei Arten von Methoden

- Bei den *Methoden* gibt es einerseits globale Versionen (*Shared Blocks*), die nicht auf eine Klasse bezogen sind, und Klassenmethoden (*Method*), die für eine Klasse gelten und die als Parameter immer ein Objekt der Klasse erwarten (z. B. *this*), wenn sie aufgerufen werden.

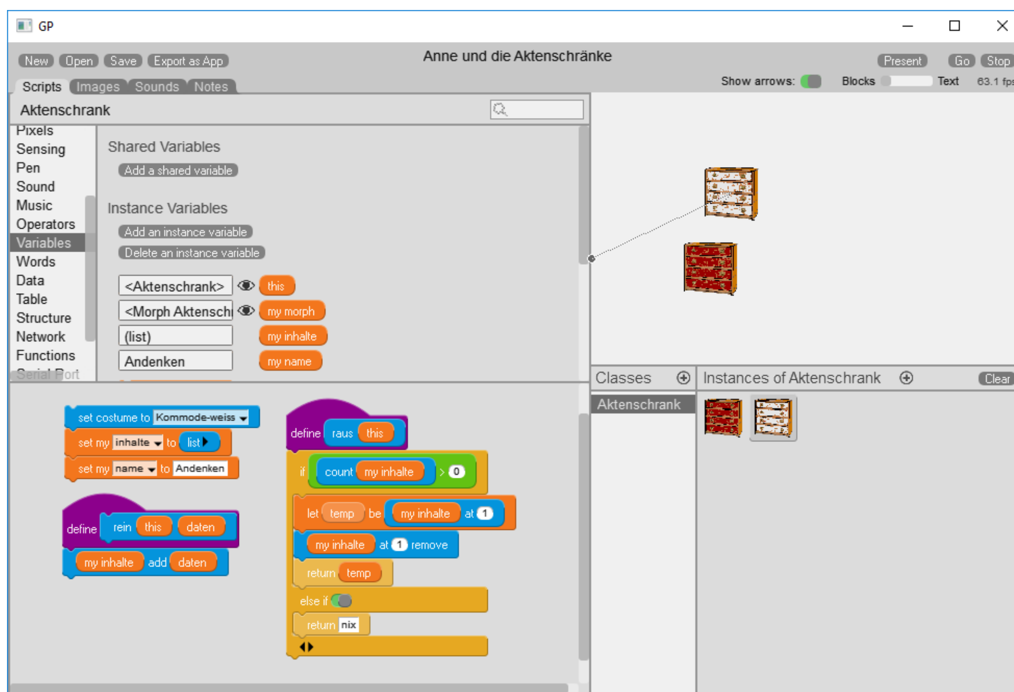
Zwei Arten von Objekten

- Objekte werden durch ein Bild im Arbeitsbereich symbolisiert (anfangs immer durch den Pfeil *ship*), aber sie können auch aus mehreren Teilen zusammengesetzt sein.

[14] weitgehend nach Modrow, Objektorientierte Programmierung mit BYOB, LOG IN 171 (2012)
[15] Lieberman, Henry: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, ACM SIGPLAN Notices, Volume 21 Issue 11, Nov. 1986

10.1 Die Kommunikation zwischen Objekten

Als Beispiel erzeugen wir die Klasse *Aktenschrank* einfacher Datenspeicher mit je einer lokalen Liste *inhalte* und einem *namen*. Die Instanzen repräsentieren wir durch Kommoden und nennen sie *Akten* und *Andenken*. Wir statten sie mit lokalen Zugriffsmethoden auf die Daten aus, indem wir die Methoden rein <daten> und raus implementieren. So erhalten wir simple Queues. Die zusätzliche Instanz wird durch Klicken auf den +-Knopf über dem Instanzenfenster erzeugt (eine ist ja automatisch da). Wir weisen ihnen zwei unterschiedliche Kostüme *Kommode-weiss* und *Kommode-rot* zu. Die gerade aktive Instanz wird durch einen Pfeil markiert (wenn diese Funktion eingeschaltet ist). Die Wertzuweisungen erfolgen „per Hand“.



Anne und die Aktenschränke

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. Wir benötigen also einen Zugriff auf die Methoden des Objekts von außen, und deshalb brauchen wir eine IT-Beauftragte Anne, die natürlich einer eigenen Klasse entstammt.

Wie kann Anne auf ihre Datenspeicher zugreifen?

Zugriff auf Objekte

Zuerst einmal muss sie überhaupt wissen, welche Aktenschränke zur Verfügung stehen. Dafür gibt es mehrere Möglichkeiten:

set shared schrank1 to this

1. Die Aktenschränke werden *globalen Variablen* zugewiesen, über die sie dann erreichbar sind.
2. Anne fragt herum, was an Aktenschränken so in der *Umgebung* zu finden ist.

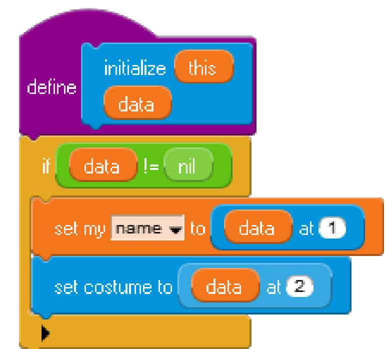
   set my schränke to neighbors within 1000 class Aktenschrank

   Sie erhält als Antwort eine Liste, in sich ggf. Aktenschränke befinden. Die kann sie dann weiterverarbeiten, z. B. indem sie die Elemente ihren eigenen Instanzvariablen zuweist.

set my andenken to my schränke at 1
set my akten to my schränke at 2

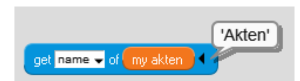
3. Anne erzeugt die Aktenschränke selbst mithilfe eines Konstruktors (*Initialize Method*). Will sie danach Attribute der Instanz ändern, dann müssen entsprechende Setter-Methoden in der Klasse existieren und veröffentlicht sein (s. u.).

   Alternativ dazu wird in der Klasse ein Konstruktor selbst geschrieben, dem eine Liste mit den gewünschten Attributen übergeben wird. Deren Elemente (hier: Name und Kostüm) können dann den Attributen zugewiesen werden.



Anne kennt jetzt zwei Aktenschränke namens *akten* und *andenken*, und sie kann deren Attribute direkt lesen. Dafür benutzt sie *get <attribut> of <object>* aus der *Sensing*-Palette.

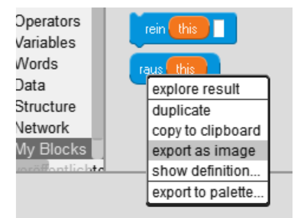
Zugriff auf Attribute

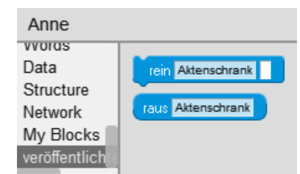


Wie aber greift sie auf die Methoden der Objekte zu?

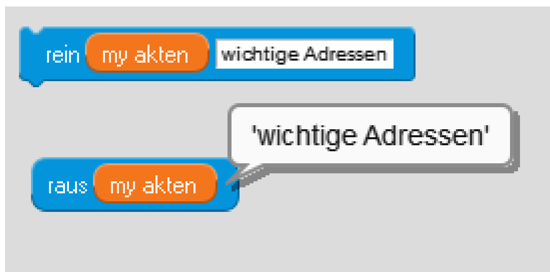
Das darf sie ja eigentlich gar nicht, weil Methoden Privatsache sind. Sollen Methoden einer Klasse von außen erreichbar sein, dann müssen diese – und nur diese – auf einer Palette veröffentlicht werden. Wir nutzen dazu das Kontextmenü der Instanz-Methoden in der *My Blocks*-Palette und exportieren die Methoden in eine neue Palette (hier: veröffentlichte Methoden). Von dort aus kann sie dann jedes andere Objekt „sehen".

Veröffentlichen von Methoden

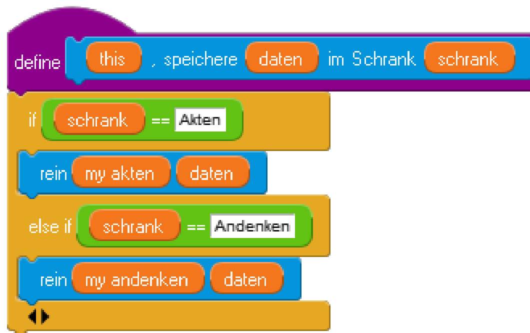




In unserem Fall kann Anne, die ja beide Aktenschränke kennt, die veröffentlichten Methoden mit entsprechenden Parametern aufrufen und so die Schränke nutzen.



Anne als gut ausgebildete IT-Beauftragte kann solche Befehle natürlich absetzen, ein normaler Benutzer vielleicht aber nicht. Anne bietet ihre Dienste deshalb öffentlich über die Palette *IT-Dienstleistungen* als Sachverständige an, indem sie eine globale Variable *IT-Dienstleisterin* anlegt. Sie veröffentlicht zwei neue Blöcke *speichere <daten> im Schrank <schrankname>* und *hole Daten aus Schrank <schrankname>*, die als Parameter die zu speichernden Daten und den zu benutzenden Aktenschrank erhalten. Damit wird die Benutzung sehr vereinfacht.



Nutzung durch Dritte

10.2 Aufgaben

1. Implementieren Sie bei den Aktenschränken selbst oder bei der IT-Beauftragten eine Zugriffskontrolle
   - a: durch eine Passwordabfrage.
   - b: mit Benutzerlisten und zugeordneten Passwords.

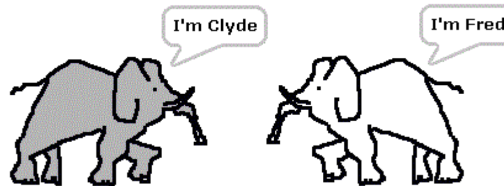
2. Verarbeiten Sie die Daten selbst, indem Sie
   - a: Plausibilitätsprüfungen einführen.
   - b: Verschlüsselung einführen.
   - c: Organisationsformen in Listen, Reihungen, Stapeln, Schlangen, Bäumen usw. einführen.

3. Verteilen Sie die Funktionalität mithilfe der Netzwerkmöglichkeiten von GP auf unterschiedliche Instanzen, die auf unterschiedlichen Rechnern laufen.

10.3 Programmierung mit Prototypen

Bisher haben wir als Prototypen eines Datenspeichers unsere Kommode, von der wir Instanzen erzeugen, an die wir – wie in anderen OOP-Sprachen auch – Botschaften in Form von Methodenaufrufen senden. Die Wirkung der Methoden und von Blöcken/Blockkombinationen erproben wir im Kontext dieses Prototypen.

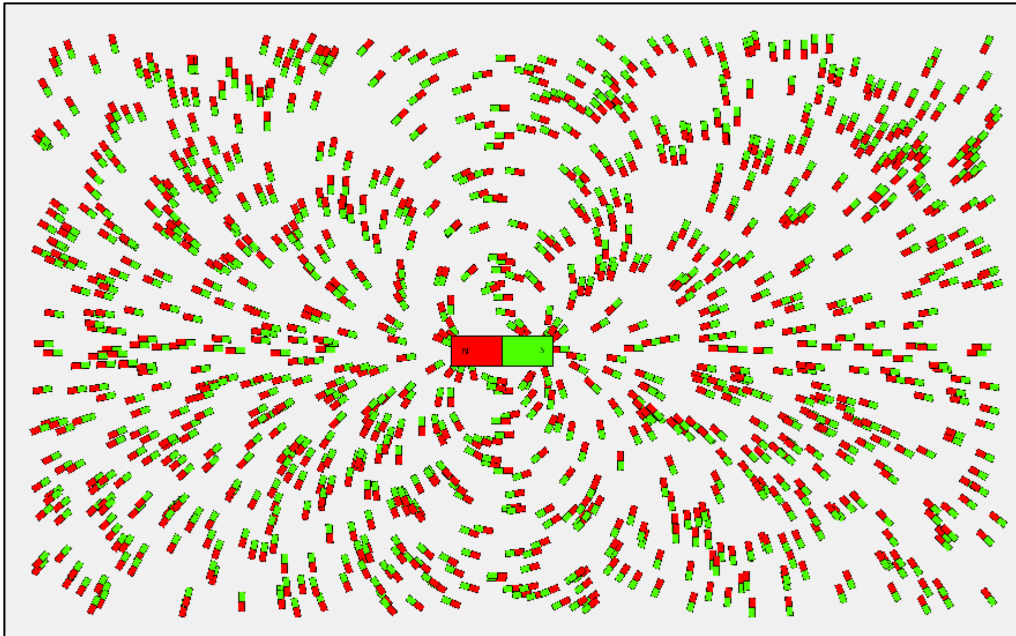
Im Originalartikel von Lieberman werden Objekte als Verkörperung der Konzepte ihrer Klasse verstanden. So steht dort der Elefant Clyde für alles, was der Betrachter unter einem Elefanten versteht. Stellt sich dieser einen Elefanten vor, dann erscheint vor seinem geistigen Auge nicht etwa die abstrakte 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."*



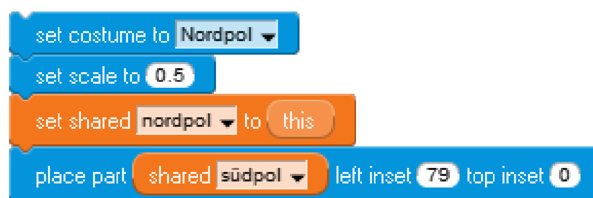
Was bedeutet dieser Ansatz für den Lernprozess? Kennt der Lernende nur ein Exemplar einer Klasse (hier: Clyde), dann beschreibt der Prototyp seine Kenntnisse vollständig, eine Abstraktion ist für ihn sinnlos. Lernt er danach andere Exemplare kennen und beschreibt diese durch Modifikationen am Original, 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 GP gehört zu jeder Klasse eine Instanz, die die Rolle als Prototyp übernimmt. Sie wird mit den gewünschten Attributen und Methoden ausgestattet. Ist deren Verhalten genügend erprobt worden, dann können weitere Instanzen entweder statisch im Instanzenbereich oder dynamisch mithilfe der Blöcke aus der Structure-Palette erzeugt werden.

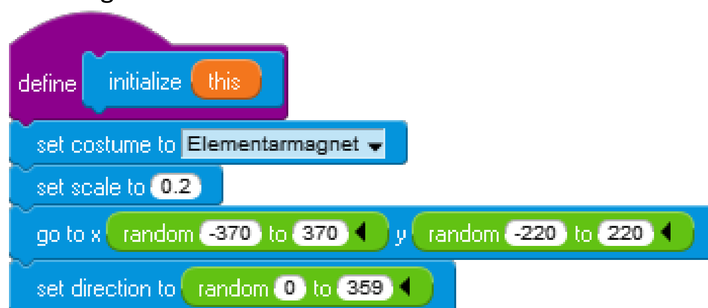
10.4 Magnete



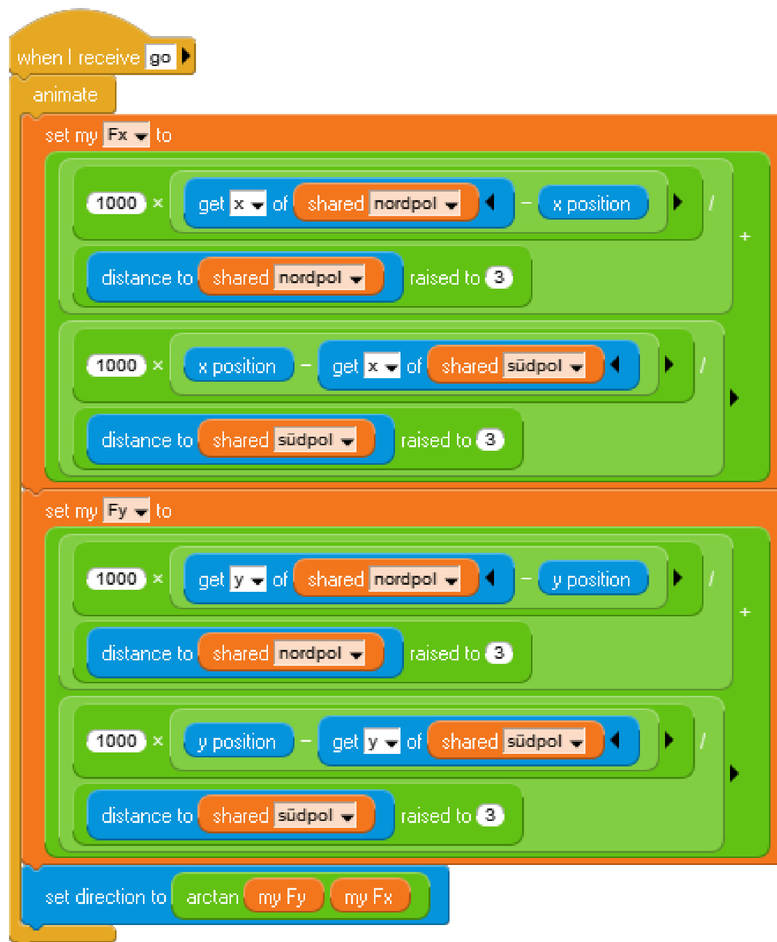
Als ein sehr einfaches Beispiel für den Umgang mit vielen Objekten wählen wir einen Stabmagneten, der von zahlreichen „Elementarmagneten“ umgeben ist. Wir zeichnen den Nordpol (rot) und Südpol (grün) des Stabmagneten und weisen diese Bilder zwei globalen Variablen namens *Nordpol* und *Südpol* zu. Aus diesen setzen wir den Magneten zusammen. Viel mehr Funktionalität besitzt er nicht, man kann ihn nur durch die Gegend schieben.



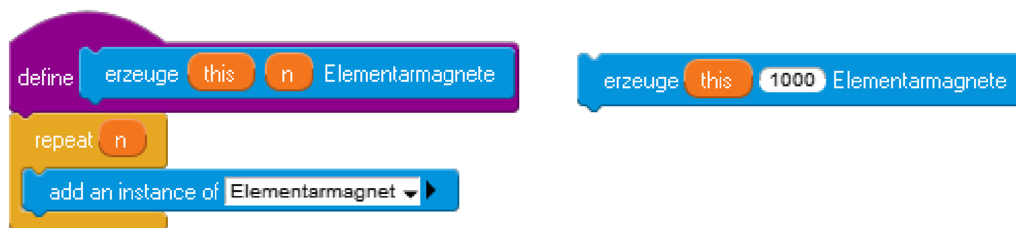
Die Klasse unserer Elementarmagnete verfügt über einen Konstruktor, der ihnen das richtige Kostüm zuweist, sie etwas verkleinert und an einen zufälligen Ort mit zufälliger Richtung erscheinen lässt.



Diese Elementarmagnete berechnen die Kraftkomponenten *Fx* und *Fy*, die sich aus ihrem Abständen zu den Polen und den Richtungskomponenten ergeben. Aus diesen bestimmen sie den Winkel der Kraft und drehen sich entsprechend.



Von diesen Elementarmagneten können wir jetzt beliebig viele erzeugen. Sie reagieren alle gleich als Instanzen der gleichen Klasse. Wir schreiben dazu an geeigneter Stelle (hier: beim Stabmagneten) eine entsprechende Methode und rufen sie auch gleich auf.

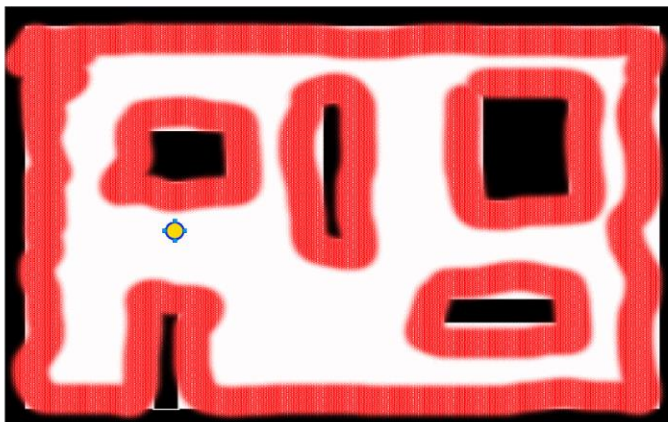


Das Ergebnis sieht man oben.

10.5 Ein lernender Roboter[16]

Als weiteres Beispiel für die Verwendung mehrerer Klassen wollen wir einen Roboter betrachten, der über vier Berührungssensoren verfügt. Kommt einer von diesen mit einem Hindernis in Berührung, dann ändert der Roboter seine Richtung, hat aber auch eine neue Beule.

Wir zeichnen mit einem Zeichenprogramm ein Bild einer Welt, die von schwarzen Wänden begrenzt ist und in der einige schwarze Hindernisse stehen. Aus Gründen, die wir gleich kennenlernen werden, versprühen wir mit der Sprühdose einen diffusen roten Nebel um die Gegenstände herum und an den Wänden entlang. In diese Welt setzen wir Roby – als kleine kreisrunde Instanz der Klasse Roboter. Weiterhin zeichnen wir ein ganz kleines blaues Sprite, das einen Wandsensor symbolisieren soll. Die entsprechende Klasse Wandsensor wird mit einem Prädikat berührt <instanz> die Wand? ausgestattet. Von dieser Klasse erzeugen wir vier Instanzen und heften sie an den Roboter. Es entsteht eine Aggregation. Den Roboter statten wir mit zwei lokalen Variablen vx und vy aus, die die Geschwindigkeitskomponenten in diesen Richtungen beschreiben. Meldet nun ein Wandsensor eine Wand, dann wird die entsprechende Geschwindigkeitskomponente geändert. Wir erhalten die folgende Konfiguration, in der sich Roby sicher zwischen den Hindernissen bewegt – wie gesagt, unter Inkaufnahme von vielen Beulen:



```
set costume to Roboter
set my wandSensorOben to new instance of Wandsensor
place part my wandSensorOben left inset 20 top inset -5
set my wandSensorUnten to new instance of Wandsensor
place part my wandSensorUnten left inset 20 top inset 40
set my wandSensorLinks to new instance of Wandsensor
place part my wandSensorLinks left inset -5 top inset 18
set my wandSensorRechts to new instance of Wandsensor
place part my wandSensorRechts left inset 43 top inset 18
```

[16] Das Beispiel hat den Laufroboter von Prof. Florentin Wörgötter, Bernstein Zentrum für Computational Neuroscience Göttingen als Vorlage, beschrieben z. B. in http://www.chip.de/news/Schnellster-Roboter-lernt-bergauf-zu-gehen_27892038.html

Um festzustellen, ob ein Berührungssensor ein Hindernis berührt, überprüfen wir alle Pixel um ihn herum, ob sie „sehr dunkel“ sind, also alle Farbkanäle einen „kleinen“ Wert haben (hier: weniger als 50). Die Farbwerte erhalten wir von einer Instanz der Klasse Main, die das Labyrinth als Kostüm erhält und den Zugriff auf dessen Pixel in die Palette Roby exportiert. Dabei müssen wir beachten, dass die Position des Roboters auf der Arbeitsfläche in einem mittig nach oben und rechts gerichteten Koordinatensystem gemessen wird, während die Position der Pixel in einem System gemessen wird, das den Ursprung oben-links hat und nach unten und rechts gerichtet ist.

Zugriff auf Pixel der Klasse Main

```
define labPixel at (this) (xp) (yp)
return pixel at x (xp) y (yp)
```

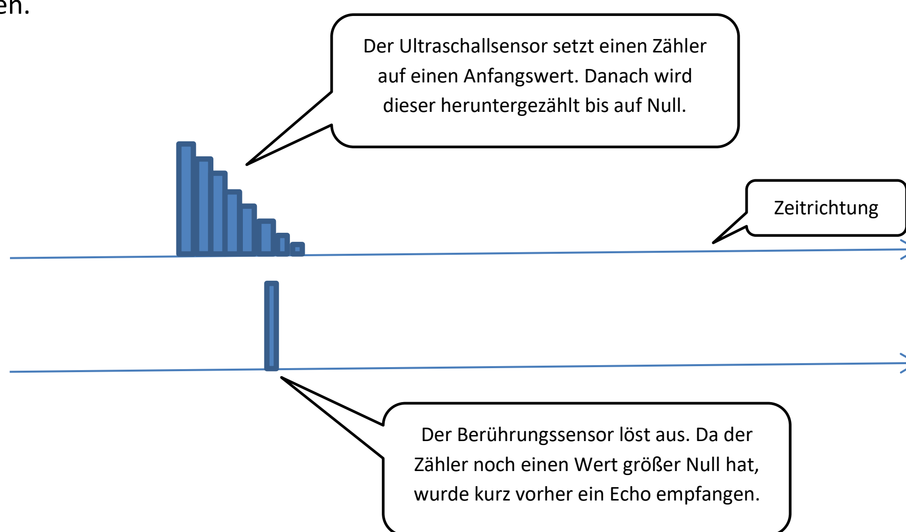
Prädikat der Wandsensoren und Roboterskript

```
define berührt (this) die Wand?
set my ergebnis to (false)
set my xp to ((x position) + (400))
set my yp to ((250) − (y position))
for (i) in (7)
    set my pixel to
        labPixel at (shared laby) ((my xp) + (i) + (-3)) ((my yp) + (-3))
    if <((red of (my pixel)) < 50) and ((green of (my pixel)) < 50) and ((blue of (my pixel)) < 50)>
        set my ergebnis to (true)
    set my pixel to
        labPixel at (shared laby) ((my xp) + (i) + (-3)) ((my yp) + (3))
    if <((red of (my pixel)) < 50) and ((green of (my pixel)) < 50) and ((blue of (my pixel)) < 50)>
        set my ergebnis to (true)
    set my pixel to
        labPixel at (shared laby) ((my xp) + (-3)) ((my yp) + (-3) + (i))
    if <((red of (my pixel)) < 50) and ((green of (my pixel)) < 50) and ((blue of (my pixel)) < 50)>
        set my ergebnis to (true)
    set my pixel to
        labPixel at (shared laby) ((my xp) + (3)) ((my yp) + (-3) + (i))
    if <((red of (my pixel)) < 50) and ((green of (my pixel)) < 50) and ((blue of (my pixel)) < 50)>
        set my ergebnis to (true)
return (my ergebnis)
```

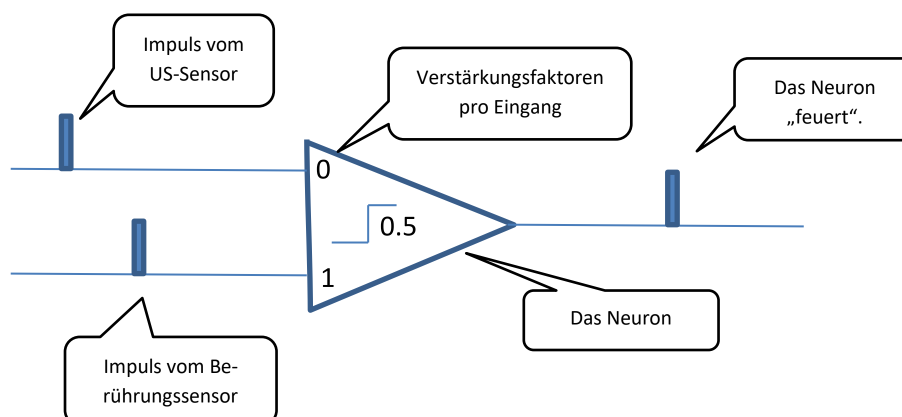
```
when I receive go
set my vx to 1
set my vy to 1
animate
    if berührt (my wandSensorOben) die Wand?
        set my vy to ((-1) × (my vy))
    if berührt (my wandSensorUnten) die Wand?
        set my vy to ((-1) × (my vy))
    if berührt (my wandSensorLinks) die Wand?
        set my vx to ((-1) × (my vx))
    if berührt (my wandSensorRechts) die Wand?
        set my vx to ((-1) × (my vx))
    move by x (my vx) y (my vy)
```

Jetzt kommt die rote Sprühfarbe rund um die Hindernisse und Wände ins Spiel. Diese soll Bereiche kennzeichnen, in denen ein Ultraschallsensor Echos von den Gegenständen empfängt. Wir statten also den Roboter mit vier Schallsensoren aus, die auf diese rote Farbe reagieren. Das Skript dafür entspricht dem der Wandsensoren, außer dass es nach einem hohen Rotwert Ausschau hält.

Der Roboter soll nun lernen, dass ein Ultraschallecho oft einer Kollision vorausgeht, und dass es deshalb besser ist, schon bei diesem Echo umzukehren. Wir brauchen also einen Mechanismus, der feststellt, dass vor einer Kollision ein Echo kam. Eine Möglichkeit, dieses zu erreichen, ist ein Zähler, der auf einen Anfangswert (hier: 10) gesetzt wird, wenn er rote Farbe (also ein Echo) feststellt. Dieser Zähler wird kontinuierlich auf Null heruntergezählt – und ggf. schon vorher wieder heraufgesetzt. Hat dieser Zähler bei einer Kollision einen Wert größer als Null, dann ist das Echo kurz vorher empfangen worden.



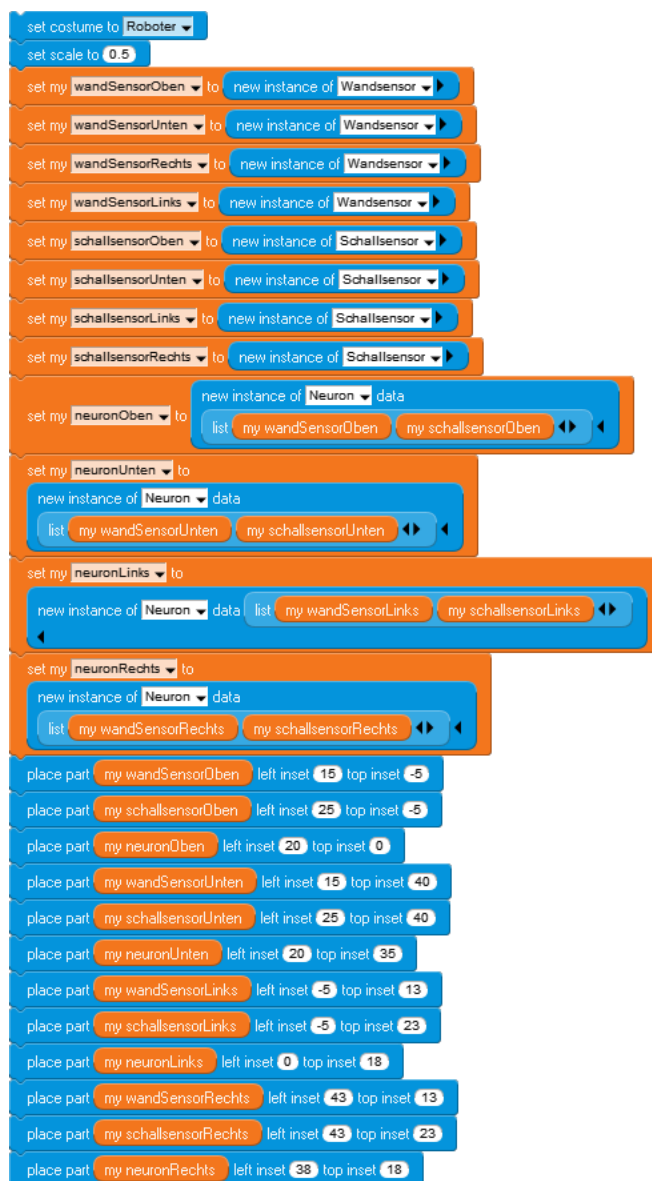
Durch diese Konstellation wird ein Lernschritt in Gang gesetzt, der in einem *Neuron* stattfindet. Dieses verfügt über zwei Eingänge, die vom zugeordneten Berührungssensor bzw. Ultraschallsensor kommen und jeweils mit einem *Gewicht* behaftet sind, sowie einen *Schwellwert*. Die Leitung vom Berührungssensor hat das Gewicht 1. Kommt von dort ein Signal z. B. der Stärke 1, dann wird dieses mit dem Gewicht 1 multipliziert. Das Ergebnis ist größer als der Schwellwert (hier: 0.5) und das Neuron „feuert“. Das Gewicht des Schallsensors hat anfangs den Wert Null. Es wird immer dann erhöht, wenn der Berührungssensor bei einer Kollision feststellt, dass der Zähler des zugeordneten Ultraschallsensors einen Wert größer als Null hat. Finden genügend viele solcher kleinen Lernschritte statt, dann überschreitet das Produkt aus Gewicht und Signal auch beim Schallsensor den Schwellwert des Neurons und dieses feuert auch in diesem Fall.



Diese Form *Pawlowschen Lernens* realisieren wir jetzt.

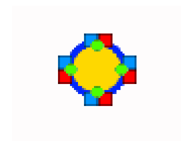
Wir erstellen eine Klasse Neuron, die vier lokale Variablen für den Zähler, das Gewicht sowie einen Wandsensor und einen Schallsensor enthält. Wenn die Skripte gestartet werden, dann fragt das Neuron seine Sensoren ab und arbeitet wie beschrieben.

Von außen, hier also vom Roboter, können Neuronen „befragt“ werden, ob sie „feuern“. Dazu exportiert die Neuron-Klasse das folgende Skript in die Palette *Roby*.



Wir konfigurieren jetzt unseren Roboter neu. Dieser erhält neben den Berührungssensoren noch vier Ultraschallsensoren sowie vier Neuronen. Diese platzieren wir oben, unten, links und rechts – und wir ordnen die Sensoren jeweils dem richtigen Neuron zu.

Der neue Roboter



Zuletzt ändern wir das Verhalten von Roby: der ändert seine Richtung, wenn das entsprechende Neuron feuert.

Roby sucht sich jetzt seinen Weg, anfangs zwischen den Hindernissen, dann entlang des „Echobereichs“.

10.6 Aufgaben

1. Verpassen Sie Roby eine Oberfläche, mit der sich die wesentlichen Faktoren ändern leicht lassen: seine Geschwindigkeit, die Gewichte, die Schwellwerte.

2. Führen Sie weitere Sensortypen, ggf. auch unter Einbeziehung eines Sensorboards, sowie weitere Ereignisse neben den Kollisionen ein.
   a: Lassen Sie Roby Korrelationen zwischen Sensorwerten und Ereignissen in unterschiedlichen „Welten“ finden. Roby passt sich so seiner Umgebung an.
   b: Diskutieren Sie Möglichkeiten, dass Roby sich an eine veränderliche Umwelt anpasst.

3. Diskutieren Sie den Bedarf nach „Vergessen“ sowie Möglichkeiten, diesen Prozess zu realisieren.