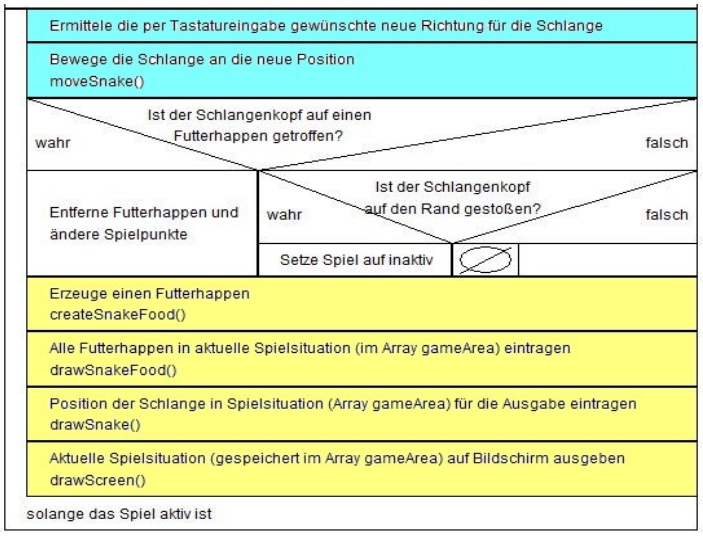
Ohne Mampf kein Kampf



Nebenan ist nochmals das Struktogramm dargestellt, das den Ablauf in der Schleife darstellt, die bei jedem Zeitschritt durchlaufen wird.

Die gelb unterlegten Blöcke waren Thema von Teil 2, die cyan unterlegten Blöcke von Teil 3.

Jetzt, in Teil 4, geht es zuerst einmal darum, dass die Schlange die Futterhappen auch tatsächlich „wegmampft“ und sich dadurch die Spielpunkte erhöhen.

Nach jeder Bewegung der Schlange um eine Zelle, muss also geprüft werden, ob der Schlangenkopf auf die Position eines Futterhappens gelangt ist.

Diese Prüfung soll in einer Methode implementiert werden. Als Bezeichner für diese neue Methode habe ich **`checkForSnakeFood`** gewählt. Wie immer muss man sich nun überlegen, welche Informationen diese Methode zur Prüfung benötigt und welche Art von Information zurück gegeben werden sollte.

Zuerst einmal überlegen wir, welche Informationen von dieser Methode benötigt werden, um festzustellen, ob der Schlangenkopf auf einen Futterhappen gestoßen ist. Sicherlich wird man dann sofort an das Array mit den Positionsangaben zur Schlange und an das Array mit den Positionsangaben zu den Futterhappen denken. Und natürlich die momentane Anzahl an Futterhappen. Mit diesen drei Informationen lässt sich die Prüfung durchführen.

Nun muss man sich überlegen, ob die Methode einen Ergebniswert haben soll, oder nicht. Und wenn sie einen Ergebniswert haben soll, welcher soll das sein. Hier sind letztendlich die Überlegungen des Programmierers gefragt. Im einfachsten Fall könnte man hier einen boolschen Wert (**`true`** oder **`false`**) zurückliefern, der angibt, ob die Schlange auf einen Futterhappen gestoßen ist. Mit Hilfe dieses Rückgabewertes würde dann dieser Futterhappen entfernt und der Stand der erreichten Spielpunkte, am Besten durch Aufruf einer neu zu erstellenden Methode, geändert.

Meine Entscheidung ist aber nun eine Andere, wobei ich zuerst auch so gedacht habe! In der Methode **`checkForSnakeFood`** soll ein eventuell gefressener Futterhappen auch gelöscht werden. Wie aber bereits in Teil 2 festgelegt, sollen die Futterhappen verschiedene Werte haben, die dann auf den aktuellen Spielstand aufaddiert werden. Das bedeutet, dass zumindest der Wert des gefressenen Futterhappens zurück gegeben werden muss.

Aber ich lasse noch einen zweiten Wert zurückgeben, nämlich die neue Anzahl von Futterhappen und diese ist der bisherige Wert um Eins vermindert. Man kann darauf auch verzichten! Aber ich wollte hier auch zeigen, wie man ggf. mehrere Werte durch eine Methode zurückliefern kann!

Also: Es soll die „neue“ Anzahl an Futterhappen UND der Wert des gerade gefressenen Futterhapens zurück geliefert werden. Was wird aber zurück geliefert, wenn kein Futterhappen gefressen wurde? Nun ja, ganz einfach: Der Wert ist dann Null und die „neue“ Anzahl ist gleich der „alten“ Anzahl an Futterhappen.

Und wie liefert man zwei Werte mit dem gleichen Datentype (beide Werte sind vom Typ Integer) zurück? Ganz einfach: Mit einem eindimensionalen Integer-Array mit zwei Zeilen.

Somit soll anhand meiner Entwurfsentscheidung der Rückgabe-Datentyp ein Integer-Array sein.

Der Methodenkopf für die Methode **`checkForSnakeFood`** sieht nun wie folgt aus:

Bis auf den Rückgabe-Datentyp ist dies nichts Neues. Wie geschrieben soll nun ein eindimensionales Integer-Array zurück geliefert werden; somit ist der Datentyp eben **`int[]`**.

```
static int[] checkForSnakeFood(int[,] positionsOfSnakeFood,
                               int amountOfSnakeFood,
                               int[,] positionsOfSnake)
{
    // To Do
}
```

Gehen wir nun an die Implementation der Methode. Zuerst einmal muss anhand der neuen, aktuellen Position des Schlangenkopfes überprüft werden, ob sich dort auch ein Futterhappen befindet.

Die Position des Schlangenkopfes ist ja der erste Eintrag in dem Array **`positionsOfSnake`**, der Eintrag mit dem Index 0. Wir speichern die Zeilen- und Spaltennummer des Schlangenkopfes zur besseren Lesbarkeit in einer Variablen.

```
// Zeile für Schlangenkopf
int rowSnakeHead = positionsOfSnake[0, 0];
// Spalte für Schlangenkopf
int columnSnakeHead = positionsOfSnake[0, 1];
```

Nun müssen wir das Array mit den Positionen der Futterhappen durchlaufen. Die Anzahl der Futterhappen kennen wir, die steht in der Variablen **`amountOfSnakeFood`**. Da die Anzahl der zu prüfenden Arrayelemente bekannt ist, wird eine **`for`**-Schleife verwendet.

Im Anweisungsblock der **`for`**-Schleife erfolgt nun die Prüfung, ob einer der gespeicherten Futterhappen die gleichen Positionsdaten wie der Schlangenkopf hat.

Da zwei Angaben auf Identität zu prüfen sind, muss die Bedingung eine UND-Verknüpfung der beiden Teil-Bedingungen sein.

```
if (positionsOfSnakeFood[i, 0] == rowSnakeHead &&
    positionsOfSnakeFood[i, 1] == columnSnakeHead)
{
    // To Do
}
```

Mit der ersten Teil-Bedingung wird die Identität der Zeilennummer überprüft und mit der zweiten Teil-Bedingung die Identität der Spaltennummer. Was muss jetzt passieren, wenn die Prüfung erfolgreich war? Damit sind wir bei dem „To Do“.

Nun, jetzt muss natürlich der Wert für diesen Futterhappen ermittelt und für die spätere Rückgabe in einer Variablen gespeichert werden. Diese Variable erstellt man am Besten am Beginn der Methode und initialisiert sie mit dem Wert 0. In meinem Programmcode hat diese Variable den Bezeichner **`valueOfSnakeFood`**. Der Wert eines Futterhappen befindet sich ja in der dritten Spalte (Index 2).

Jetzt wird es etwas anspruchsvoller! Dieser vom Schlangenkopf gefressene Futterhappen muss jetzt natürlich aus dem Array mit den Futterhappen entfernt werden. Die beiden Tabellen für das Futterhappen-Array sollen diese Situation beschreiben. Angenommen, der Schlangenkopf befindet sich an der Position Zeile=56 und Spalte=12. In der linken Tabelle sind die momentan vor der Prüfung vorhandenen Futterhappen aufgeführt. Beim Durchlaufen des Arrays wird also bei der Zeile 2 (Index = 1) die Identität festgestellt.

Zeile	Spalte	Wert
5	35	8
56	12	2
13	42	4
30	17	9
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Zeile	Spalte	Wert
5	35	8
13	42	4
30	17	9
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Nachdem der Wert des gefressenen Futterhap-

pens (hier: 2) in der zuvor beschriebenen Variable **`valueOfSnakeFood`** gespeichert wurde, muss nun dieser Eintrag entfernt werden.

Schaut man sich den Zustand vor dem Fressen (linke Tabelle) und den Zustand nach dem Fressen (rechte Tabelle) an, sieht man, dass eigentlich nur die noch folgenden Positionen der Futterhappen einfach eine Zeile nach oben „rutschen“ müssen. Und dann muss die vorher letzte Zeile auf die Werte 0 gesetzt werden. Und zum Schluss muss die Anzahl der Futterhappen um Eins vermindert werden.

Okay...schauen wir und dass im Detail an!

In meinem Programmcode lasse ich bei Fressen eines Futterhappens auch einen Ton erklingen. Dies macht die Methode **`Beep()`** des Konsolen-Objektes.

Dann wird wie oben beschrieben der Wert des gefressenen Futterhappen gespeichert.

Und jetzt wird es etwas „tricky“! Nun werden die Positionen der nachfolgenden Futterhappen nach oben verschoben. Die Verschiebung erfolgt wieder mit einer Zählschleife, da wir ja wissen, wie viele Futterhappen noch folgen!

```
// Prüfe, ob der Kopf auf einen Futterhappen gestoßen ist
for (int i = 0; i < amountOfSnakeFood; i++)
{
    if (positionsOfSnakeFood[i, 0] == rowSnakeHead &&
        positionsOfSnakeFood[i, 1] == columnSnakeHead)
    {
        Console.Beep();

        // Wert des Futterhappens speichern
        valueOfSnakeFood = positionsOfSnakeFood[i, 2];

        // Gefressenen Futterhappen entfernen
        // durch Verschieben der nachfolgenden
        // um eine Position Richtung Beginn

        for (int j = i; j < amountOfSnakeFood - 1; j++)
        {
            positionsOfSnakeFood[j, 0] = positionsOfSnakeFood[j + 1, 0];
            positionsOfSnakeFood[j, 1] = positionsOfSnakeFood[j + 1, 1];
            positionsOfSnakeFood[j, 2] = positionsOfSnakeFood[j + 1, 2];
        }

        // Letzten Futterhappen löschen
        positionsOfSnakeFood[amountOfSnakeFood - 1, 0] = 0;
        positionsOfSnakeFood[amountOfSnakeFood - 1, 1] = 0;
        positionsOfSnakeFood[amountOfSnakeFood - 1, 2] = 0;

        // Anzahl der Futterhappen dekrementieren
        amountOfSnakeFood--;

        // Weitere Prüfung beenden
        break;
    }
}
```

Diese „innere“ Zählschleife verwendet die Zählvariable **`j`**. Diese Zählvariable wird nun mit der Position des „gefressenen“ Futterhappens initialisiert (**`int j = i`**). In unserem Beispiel mit den beiden Tabellen oben erhält **`j`** den Wert 1.

Die Laufbedingung lautet nun: Solange **`j`** kleiner als die momentane Anzahl Futterhappen ist, soll die Zählschleife durchlaufen werden (**`j < amountOfSnakeFood`**). Die Laufvariable **`j`** wird nach jedem Durchlaufen des Anweisungblocks um Eins erhöht (**`j++`**).

Im Anweisungsblock werden nun die drei Zahlenwerte für einen Futterhappen (Zeilennummer, Spaltennummer und Wert) vom nachfolgenden Futterhappen (**`j+1`**) auf die aktuelle Position (**`j`**) kopiert.

Wenn diese **`for`**-Schleife abgearbeitet wurde, befindet sich aber noch der letzte Futterhappen doppelt in der Tabelle. Deshalb wird der momentan letzte Futterhappen mit der Zeilen- und Spaltennummer 0 und dem Wert 0 versehen. Und zum Schluss wird noch die Anzahl der Futterhappen um Eins vermindert. Jetzt enthält das Array nur noch die „ungefressenen“ Futterhappen!

Die nachfolgende Anweisung **`break`** bewirkt, dass die äußere **`for`**-Schleife vorzeitig beendet wird. Nachdem wir ja einen Treffer hatten, müssen die nachfolgenden Futterhappenpositionen ja nicht mehr überprüft werden!

Die Anweisung **`break`** kann in Entscheidungs- oder Wiederholungsstrukturen verwendet werden, um diese vorzeitig zu beenden. Benutzt wurde sie bereits bei der **`switch`**-Anweisung, um eine weitere Auswertung von Fällen zu verhindern.

Jetzt muss nur noch der Rückgabewert erstellt werden. Es soll ja ein eindimensionales Array mit zwei Werten zurück gegeben werden. Also erstellen wir ein solches Array mit der Anweisung `int[] snakeFoodInfo = new int[2];`. und speichern an der Stelle 0 die neue Anzahl von Futterhappen und an der Stelle 1 den Wert des gefressenen Futterhappens. Umgekehrt ginge das natürlich auch :-)

Mit der Anweisung `return snakeFoodInfo;` wird dann die Methode beendet und das Array **`snakeFoodInfo`** zurück geliefert.

Okay...wie sieht das nun in der **`do...while`**-Schleife aus, in der das Spiel abläuft?

Hier wird nun die eben besprochene Methode aufgerufen un der Ergebniswert, das Integer-Array wird einem solchen Array zugewiesen.

Ich verwende hier den gleichen Bezeichner, was aber nicht nötig ist!

```
// Prüfe, ob der Schlangenkopf einen
// Futterhappen getroffen hat und liefere dessen Wert
// und die neue Anzahl Futterhappen zurück
int[] snakeFoodInfo = checkForSnakeFood(positionsOfSnakeFood,
                                        amountOfSnakeFood,
                                        positionsOfSnake);

int newAmountOfSnakeFood = snakeFoodInfo[0];
int newPoints = snakeFoodInfo[1];
```

Die beiden Werte in diesem Array werden nun den zwei neu erstellten Variablen **`newPoints`** und **`newAmountOfSnakeFood`** zugewiesen, die im Anschluss verwendet werden. Dies dient nur der besseren Lesbarkeit! Man kann genauso gut auch die beiden Werte anschließend unter den Bezeichnungen `snakeFoodInfo[0]` und `snakeFoodInfo[1]` verwenden!

Wie ermittele ich aber nun, dass tatsächlich ein Futterhappen gefressen wurde?

Nun, wenn der Wert von `newAmountOfSnakeFood` kleiner als der Wert von `amountOfSnakeFood` ist, wurde ein Futterhappen gefressen! Wenn nicht, dann eben nicht :-)

Mit der Bedingung `newAmountOfSnakeFood < amountOfSnakeFood` wird also ermittelt, ob ein Futterhappen gefressen wurde oder nicht. Außerdem ist ein gefressener Futterhappen ein Indikator, dass die Schlange **nicht** auf den Rand oder auf sich selbst gestoßen ist, was ja das Spiel beenden würde.

Falls diese Bedingung den Wert **`true`** liefert, muss also folgendes erledigt werden:

1. Die bisher erreichte Punktzahl muss um den Wert des gefressenen Futterhappens erhöht werden,
2. die aktuelle Zahl der noch existierenden Futterhappen muss angepasst werden und
3. die Schlange muss um ein Körperelement wachsen.

Die beiden ersten Punkte sind sehr leicht abzuarbeiten, für den dritten Punkt werden wir gleich noch eine Methode erstellen.

Die neue Anzahl an Futterhappen und der Wert des gefressenen Futterhappens wurde ja von der vorher aufgerufenen Methode in Form des Array **`snakeFoodInfo`** zurückgeliefert und dann zwecks besserer Lesbarkeit den beiden Variablen zugewiesen.

```
int newAmountOfSnakeFood = snakeFoodInfo[0];
int newPoints = snakeFoodInfo[1];

// Wenn ein Futterhappen gefressen wurde:
// Aktualisiere: Menge der Futterhappen
//               Erreichte Punktezahl
//               Länge der Schlange
if (newAmountOfSnakeFood < amountOfSnakeFood)
{
    amountOfSnakeFood = newAmountOfSnakeFood;
    points += newPoints;
    lengthOfSnake = addSnakePartToEnd(positionsOfSnake,
                                      lengthOfSnake,
                                      lastPositionOfSnakeTail);
}
else
{
    // Es wurde kein Futterhappen gefressen:
}
```

Für die bisher erreichten Spielpunkte ist allerdings noch eine Variable notwendig, die ebenfalls zu Beginn der **`Main`**-Methode zu deklarieren ist. Da es nur „ganze“ Spielpunkte gibt, ist der Datentyp **`int`** und der von mir gewählte Bezeichner lautet **`points`**. Zu dem aktuellen Wert von **`points`** werden nun die Punkte des gerade gefressenen Futterhappen hinzu addiert (Achtung! Kurzschreibweise).

Etwas (aber nicht viel) schwieriger ist nun das Verlängern der Schlange um ein Körperelement. Das neue Körperelement muss an der Stelle entstehen, an der sich **vor** dem Bewegen der Schwanz der Schlange befunden hat. Da die Schlange aber gerade bewegt wurde (mittels **`moveSnake`**), ist diese Position quasi verloren!

Deshalb ist es (bei dieser Art des Entwurfs) notwendig, das „alte“ Ende der Schlange zu speichern! Die Speicherung der Positionen der einzelnen Körperelemente erfolgt ja in dem zweidimensionalen Array **`positionsOfSnake`**.

Für das Speichern der letzten Position des Schlangenendes habe ich mich ebenfalls für ein zweidimensionales Array mit dem Bezeichner **`lastPositionOfSnakeTail`** entschieden. Dieses Array hat aber lediglich eine Zeile mit eben zwei Spalten.

Die Deklaration erfolgt mit `int[,] lastPositionOfSnakeTail = new int[1, 2];` zu Beginn der **`Main`**-Methode.

Das bedeutet nun aber, dass an allen Stellen im Programm, an denen sich die Position der Schlange ändert, müssen die Werte in diesem Array angepasst werden. Aufgrund der Modularität des Programms ist dies nur eine Methode, nämlich diejenige, in der das Bewegen der Schlange stattfindet (`moveSnake`).

Diese Methode erhält nun zusätzlich zu den bisherigen Parametern das gerade erstellte Array **`lastPositionOfSnakeTail`** übergeben und vor der „Bewegung“ der Schlange wird dann diese letzte Position im Array gespeichert.

Und nun zurück zur Spielschleife in der **`Main`**-Methode! Wie oben im Kasten dargestellt, muss die Länge der Schlange geändert werden, in dem an der Position, an der sich vor der Bewegung das letzte Körperelement der Schlange befand, nun ein weiteres Körperelement als quasi „neuer“ Schwanz hinzugefügt wird. Dieser Vorgang soll in einer neuen Methode stattfinden.

Diese neu zu erstellende Methode soll den Bezeichner **`addSnakePartToEnd`** erhalten. Wie immer muss man sich nun überlegen, wie der Methodenkopf aussehen muss. Zuerst die Parameter! Welche Informationen werden von der Methode für ihre Aufgabe benötigt?

Nun ja, im Wesentlichen alle Informationen über die Schlange: Positionsangaben, Länge **und** eben die Position, an der sich das Ende der Schlange zuletzt befunden hat.

Muss die Methode einen Wert zurückgeben? Nicht zwangsläufig. Die Methode soll ja „nur“ die Länge der Schlange inkrementieren. Somit könnte man die neue Länge der Schlange eigentlich auch in der **`Main`**-Methode festlegen….eigentlich! Aber „uneigentlich“ ist das nicht so geschickt, da es ja eine maximale Länge der Schlange gibt, und wenn diese erreicht wurde, wird die Schlange trotz weiterer gefressener Futterhappen nicht verlängert.

Deshalb ist es sinnvoll, die (neue) Länge der Schlange als Rückgabewert zu verwenden!

```
// Fügt ein neues Körperteil am Ende der Schlange hinzu
static int addSnakePartToEnd(int[,] positionsOfSnake,
                             int lengthOfSnake,
                             int[,] lastPositionOfSnakeTail)
{
    // To Do
}
```

Dieser Wert wird in einer Integer-Variablen gespeichert und daher ist der Typ der Rückgabe ein **`int`**.

Was muss nun **in** der Methode (**`// To Do`**)passieren? Das ist nicht wirklich kompliziert. Zuerst sollte überprüft werden, ob die maximale Länge der Schlange erreicht wurde. Die aktuelle Länge wird ja als Parameter übermittelt und kann dann mit der maximalen Länge in `maxLengthOfSnake` verglichen werden. Ist diese Länge erreicht, endet die Methode sofort und liefert eben diesen Wert zurück.

Andernfalls muss in der Liste mit den Positionsangaben `positionsOfSnake` „am Ende“ die alte Position des Schlangenendes als neues Schlangenende hinzugefügt werden. Der Wert von **`lengthOfSnake`** muss dann inkrementiert werden und danach endet die Methode und liefert diesen neuen Wert von **`lengthOfSnake`** zurück.

Bei Aufruf der Methode im Rahmen der Spielschleife wird dieser zurück gelieferte Wert dann der Variablen **`lengthOfSnake`** als neue Länge zugewiesen.

Damit kommen wir nun zu dem **`else`**-Teil der Entscheidungsstruktur im Kasten auf Seite 5!

Sollte die Schlange **keinen** Futterhappen gefressen haben, könnte es ja sein, dass die Schlange entweder auf den Rand oder auf ihren eigenen Körper gestoßen ist. Beides führt zum Ende des Spiels. Auch diese „Kollisionsprüfung“ soll in einer Methode erfolgen.

Für diese Methode habe ich den Bezeichner **`checkCollisionWithBorderOrSnakeBody`** gewählt. Damit die Methode eine Kollision ermitteln kann, muss diese wissen, wo sich der Kopf der Schlange befindet und auch die aktuelle Spielsituation kennen. Ersteres erhält sie über das Array **`positionsOfSnake`** und Letzteres durch das Array **`gameArea`**.

Hier ist auch ein Designmangel „versteckt“, der sich aus der fehlenden Speicherung der Spielfeldbegrenzung ergibt. Dazu aber später mehr :-)

```
static bool checkCollisionWithBorderOrSnakeBody(int[,] positionsOfSnake,
                                                char[,] gameArea)
{
    // To Do
}
```

Ob eine Kollision stattgefunden hat oder nicht, soll die Methode durch einen boolschen Wert anzeigen. Somit ist der Rückgabetyp **`bool`**. Dieser zurückgegebene Wert wird dann auch verwendet, um die Spielschleife weiterlaufen zu lassen oder zu beenden.

Was passiert nun in der Methode? Zuerst einmal benötigen wir eine boolsche Variable, die durch

die Methode zurück gegeben werden soll. Diese zeigt an, ob der Spielfeldrand oder der Schlangenkörper getroffen wurde. Diese in der Methode neu zu erstellende boolsche Variable heißt in meinem Projekt **`isBorderOrBodyHit`** und wird mit dem Wert **`false`** initialisiert.

Analog zur Methode für die Prüfung, ob ein Futterhappen gefressen wurde, soll wird zuerst die Zeilen- und Spaltennummer des Schlangenkopfes ermittelt. Mit diesen beiden Positionsangaben kann nun überprüft werden, ob sich im Spielfeld an dieser Position entweder das Zeichen für den Spielfeldrand (**`gameBorderCharacter`**) oder des Schlangenkörpers (**`snakeBodyCharacter`**) befindet.

Ist dies der Fall, wird der Variablen **`isBorderOrBodyHit`** der Wert **`true`** zugewiesen. Danach endet die Methode mit der Anweisung **`return isBorderOrBodyHit;`**

Der zurück gegebene Wert zeigt mit **`true`** also an, ob Rand oder Körper getroffen wurde. Dann soll das Spiel enden. Für die Laufbedingung der Spielschleife wird die Variable **`isGameActive`** verwendet. Um das Spiel zu beenden, muss diese Variable aber auf **`false`** gesetzt werden!

Somit wird die Negation des zurückgegeben Wertes der Variable **`isGameActive`** zugewiesen und die Spielschleife endet dann.

```
// Falls kein Futterhappen gefressen wurde:
// Wurde der Rand oder der Schlangenkörper getroffen?
isBorderOrBodyHit = checkCollisionWithBorderOrSnakeBody(positionsOfSnake,
                                                        gameArea);
// Spiel stoppt, wenn Rand getroffen wurde
isGameActive = !isBorderOrBodyHit;
```

Die Grundfunktionalitäten des Spiels sind jetzt vollständig implementiert, so dass sich das Spiel jetzt tatsächlich spielen lässt. Allerdings fehlt noch die Anzeige der Spielinformationen und das Aussehen des Spiels lässt sich noch etwas pimpen :-)

Damit geht es hinter den Aufgaben weiter!

Aufgaben

1. Implementieren Sie die Methode **`checkForSnakeFood`** und fügen Sie den Aufruf der Methode an der notwendigen Stelle in der Spielschleife ein.
2. Implementieren Sie in der Spielschleife die Entscheidungsstruktur, die überprüft ob ein Futterhappen gefressen wurde oder nicht.
3. Implementieren Sie die Methode **`addSnakePartToEnd`** für den Aufruf im **`if`**-Teil der Entscheidungsstruktur.
4. Implementieren Sie die Methode **`checkCollisionWithBorderOrSnakeBody`**, die im **`else`**-Teil der Entscheidungsstruktur aus Aufgabe 4 benötigt wird.
5. Ergänzen Sie im **`else`**-Teil den Aufruf der Methode und die Zuweisung des Ergebniswertes an die Variable **`isGameActive`**.

Jetzt ist das Spiel also grundlegend spielbar. Es fehlt aber noch die Anzeige der Spielinformationen und es sollte auch eine Wiederholung des Spiels möglich sein. Auch die optische Gestaltung ist noch zu verbessern.

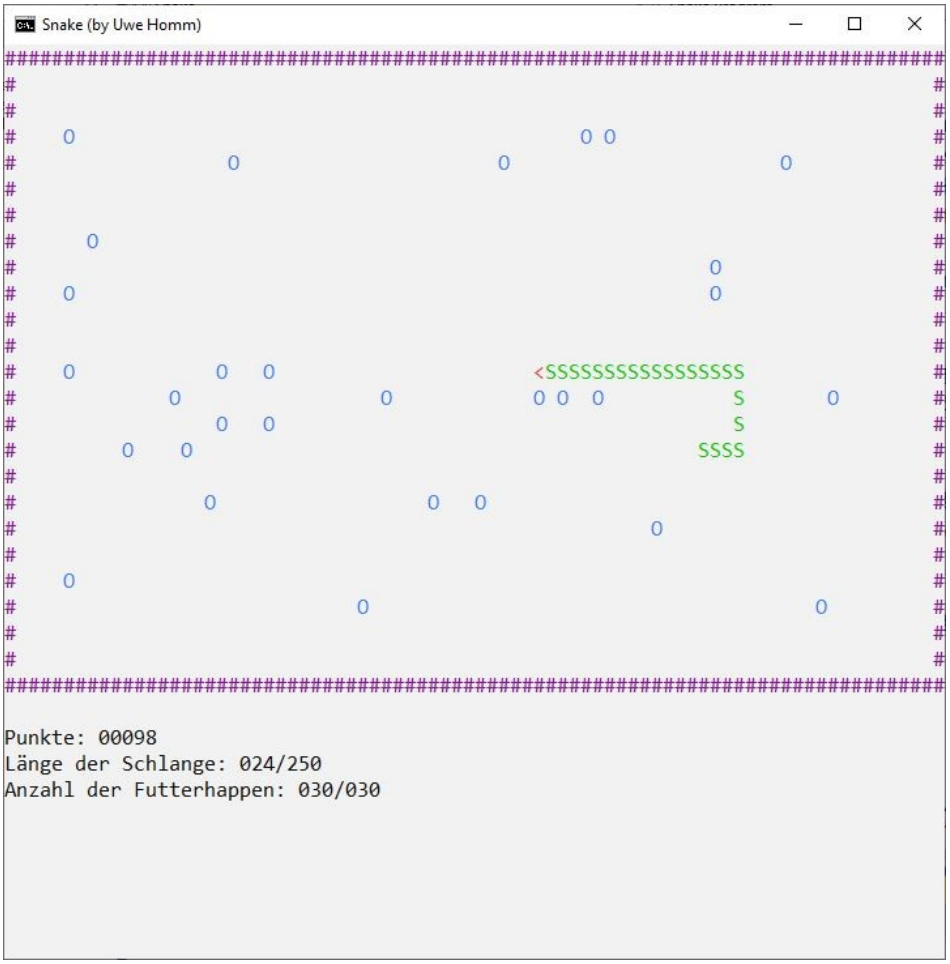
Dies soll auf den folgenden Seiten bearbeitet werden!

Anzeigen der Spielinformationen

Für die Anzeige der Spielinformationen wurde ja von Anfang an ein Bereich von Zeilen freigehalten, die durch die Konstante **`bottomBorder`** (Default: 10 Zeilen) definiert wurde.

In diesem Bereich sollen nun einige Spielinformationen dargestellt werden. Die Anzeige erfolgt ebenfalls durch eine Methode.

Als Bezeichner für diese Methode habe ich **`showGameInformations`** gewählt. Da sich die darzustellenden Informationen in Variablen der **`Main`**-Methode befinden, müssen diese als Parameter an diese Methode übergeben werden. Welche Informationen angezeigt werden sollen, ist letztendlich eine Entscheidung des jeweiligen Programmierers. In meinem Projekt soll die Anzeige der Spielinformationen wie dargestellt erfolgen.



Die erste Information ist die erreichten Spielpunkte, die zweite Information ist die momentane Länge der Schlange sowie die maximale Länge der Schlange. Und die dritte Information ist die aktuelle Anzahl an Futterhappen sowie die maximale Anzahl an Futterhappen.

Die beiden Maximalwerte sind ja in den global deklarierten Konstanten hinterlegt und können somit von der Methode zugegriffen werden.

Die drei Variablen **`points`**, **`lengthOfSnake`** und **`amountOfSnakeFood`** hingegen, müssen an die Methode übergeben werden.

Die Methode erhält aber noch eine vierte Information, nämlich ob das Spiel noch läuft oder aufgrund einer Kollision beendet wurde.

Im letzteren Fall soll noch die Meldung „Spiel beendet!“ angezeigt werden. Und für die noch zu programmierende Wiederholung des Spiels, soll dort auch noch die Frage angezeigt werden, ob das Spiel wiederholt werden soll.

Da in dieser Methode lediglich Textausgaben vorgenommen werden, ist kein Ergebniswert nötig, so dass der Rückgabetyp **`void`** lautet. Der vollständige Methodenkopf sieht also wie neben dargestellt aus.

```
// Zeigt die Spielinformationen an
static void showGameInformations(bool isBorderHit,
                                 int points,
                                 int lengthOfSnake,
                                 int amountOfSnakeFood)
```

Die Definition der Methode erfolgt nach eigenem Gusto[1]. Deshalb hier meine Version der Anzeige der Spielinformationen. Da kann man seine eigenen Vorstellungen durch Änderung der Farben zur Ausgabe etc. ausleben :-)

```
// Zeigt die Spielinformationen an
static void showGameInformations(bool isBorderHit,
                                 int points,
                                 int lengthOfSnake,
                                 int amountOfSnakeFood)
{
    Console.ForegroundColor = gameForegroundColor;

    // Ausgabe der Punkte
    Console.SetCursorPosition(0, windowHeight - bottomBorder + 1);
    Console.Write("Punkte: {0}", points.ToString("00000"));

    // Ausgabe Länge der Schlange
    Console.SetCursorPosition(0, windowHeight - bottomBorder + 2);
    Console.Write("Länge der Schlange: {0}/{1}", lengthOfSnake.ToString("000"),
                                                 maxLengthOfSnake.ToString("000"));

    // Ausgabe Anzahl der Futterhappen
    Console.SetCursorPosition(0, windowHeight - bottomBorder + 3);
    Console.Write("Anzahl der Futterhappen: {0}/{1}", amountOfSnakeFood.ToString("000"),
                                                      maxAmountOfSnakeFood.ToString("000"));

    // Ausgabe bei Spielende
    if (isBorderHit)
    {
        Console.SetCursorPosition(0, windowHeight - bottomBorder + 4);
        Console.WriteLine("Spiel beendet!");
    }
}
```

Die Textausgaben erfolgen mit Hilfe von Formatstrings, bei denen durch Platzhalter die einzufügenden Werte dargestellt sind. Die numerischen Werte werden durch die **`ToString`**-Methoden in Zeichenketten umgewandelt. Vier der fünf **`ToString`**-Methoden enthalten als Parameter die Zeichenfolge **`"000"`**, die bewirkt, dass die Zahlen prinzipiell dreistellig angezeigt werden sollen. Außer bei den Punkten, die werden fünfstellig angezeigt.

Die Position der Ausgabe wird durch die Methode **`SetCursorPosition`** gesteuert; diese Methode erhält die Zeilen- und Spaltennummer für die Ausgabe. Hierzu werden die bereits erwähnten Konstanten verwendet.

Diese Methode muss jetzt noch an den „richtigen“ Stellen in der **`Main`**-Methode aufgerufen werden. Damit bereits direkt nach der Initialisierung die Spielinformationen angezeigt werden, erfolgt der erste Aufruf noch vor der **`do...while`**-Spielschleife. Weiterhin soll nach jedem Zeitschritt, wenn also die neue Spielsituation durch Aufruf der Methode **`drawScreen`** gezeichnet wurde, die Ausgabe der aktuellen Spielinformationen erfolgen.

Wiederholung des Spielschleife

Damit das Spiel nich sang- und klanglos endet und sich das Fenster sofort schließt, soll der Benutzer die Möglichkeit haben, das Spiel zu wiederholen, ohne dass das Programm neu gestartet werden muss.

Dies ist recht einfach zu implementieren! Wie man unschwer erkennt, wird hierfür eine Wiederholgsstruktur benötigt. Eine Zählschleife (**`for`**) kann das nicht sein, ergo muss es eine offene Schleife, entweder eine **`while`**-Schleife oder eine **`do...while`**-Schleife sein.

1 Wer diesen Ausdruck nicht kennt: **`https://de.wikipedia.org/wiki/Gusto`** abgerufen am 11.062020

Und hier mach eigentlich nur die do...while-Schleife einen Sinn, da das Spiel ja mindestens einmal ablaufen soll und sich erst nach Spielende die Frage stellt, ob das Spiel wiederholt werden soll.

Somit „baut“ man um alle bisherigen Anweisungen in der **`Main`**-Methode eine weitere **`do...while`**-Schleife, die nach Ablauf der inneren Spielschleife per Textausgabe die Frage nach einer Wiederholung stellt.

Direkt im Anschluss wird die Antwort auf diese Frage in eine vor der äußeren Schleife definierten Variable (bei mir: **`playAgain`**) eingelesen und die bisherigen Anweisungen in der Main-Methode werden aufgrund dieser Eingabe erneut ausgeführt. Das Prinzip ist im Kasten dargestellt.

```
static void Main(string[] args)
{
    // Eingabe für Spielwiederholung
    string playAgain = "";
    do
    {
        // ============================================
        // Variablen
        // ============================================
        // Zum Ziehen von Zufallszahlen für Futterhappen
        Random randomNumbersGenerator = new Random();
        // und alle weiteren Anweisungen vor der Spielschleife
        // Ablauf des Spieles
        do
        {
            // Wartezeit
            Thread.Sleep(10);
            // und alles, was noch in der Spielschleife steht
        }
        while (isGameActive);

        // Spiel wiederholen
        Console.Write("Nochmal spielen (j/n)? > ");
        playAgain = Console.ReadLine();

    }
    while (playAgain == "j" || playAgain == "J");
```

Eine weitere Variante wäre auch, den gesamten Inhalt der **`Main`**-Methode in eine Methode mit dem Bezeichner z.B. **`startGame`** auszulagern.

In der nun leeren **`Main`**-Methode wird innerhalb der beschriebenen „äußeren“ **`do...while`**-Schleife diese Methode **`startGame`** aufgerufen.

Damit würde die „neue“ Main nur noch wie neben dargestellt aussehen; der Anfang der Methode **`startGame`** ist ebenfalls im Bild zu sehen.

Ist doch übersichtlicher, oder?

Okay...und jetzt soll die Ausgabe noch etwas „gepimpt“ werden :-)

```
0 Verweise
static void Main(string[] args)
{
    // Eingabe für Spielwiederholung
    string playAgain = "";
    do
    {
        // Start des Spiels
        startGame();

        // Spiel wiederholen
        Console.Write("Nochmal spielen (j/n)? > ");
        playAgain = Console.ReadLine();

    }
    while (playAgain == "j" || playAgain == "J");
}

1 Verweis
static void startGame()
{
    // ============================================
    // Variablen
    // ============================================
    // Zum Ziehen von Zufallszahlen für Futterhappen
    Random randomNumbersGenerator = new Random();
```

Aussehen der Schlange verbessern

Nun ja, was eine Verbesserung der Anzeige ist, ist ziemlich individuell. Da hat jeder seinen persönlichen Geschmack :-)

Bisher besteht die gesamte Schlange aus dem Zeichen **`'S'`** und die Farbgebung ist auch einfach die voreingestellte Vordergrundfarbe, die für alle Spielelemente gilt. Das soll nun etwas „aufgehübscht“ werden!

In meinem Projekt soll das Zeichen für den Spielfeldrand eine eigene Vordergrundfarbe haben. Auch die Futterhappen und der Schlangenkörper sollen jeweils eine eigene Vordergrundfarbe haben, wobei der Schlangenkopf ebenfalls eine eigene Vordergrundfarbe haben soll.

In dem Screenshot auf Seite 8 dieses PDFs kann man dies sehen: Der Spielfeldrand hat die Farbe **`DarkMagenta`**, die Futterhappen haben die Farbe **`Blue`**, der Schlangenkörper wird in der Farbe **`Green`** und der Schlangenkopf in der Farbe **`Red`** dargestellt.

Für alle diese Farbeinstellungen sind globale Konstanten definiert. Zusammen gefasst sind sie in dem Kasten dargestellt.

Auch das Aussehen des Schlangenkopfes soll geändert werden.

```
// Hintergrundfarbe Spielfeld
const ConsoleColor gameBackgroundColor = ConsoleColor.White;

// Vordergrundfarbe Spielfeld
const ConsoleColor gameForegroundColor = ConsoleColor.Black;

// Vordergrundfarbe Spielfeldrand
const ConsoleColor gameBorderColor = ConsoleColor.DarkMagenta;

// Vordergrundfarbe Futterhapen
const ConsoleColor snakeFoodCharacterColor = ConsoleColor.Blue;

// Vordergrundfarbe Schlange
const ConsoleColor snakeHeadCharacterColor = ConsoleColor.Red;
const ConsoleColor snakeBodyCharacterColor = ConsoleColor.Green;
```

Ich habe mich dazu entschieden, den Schlangenkopf durch ein Zeichen zu ersetzen, welches auch die Laufrichtung der Schlange widerspiegelt. Dies bedeutet, dass es nicht nur ein einzelnes Zeichen für den Schlangenkopf gibt, sondern vier verschiedene Zeichen, nämlich: **`<`**, **`^`**, **`>`** und **`v`**.

Auch diese vier Zeichen (deshalb der Datentyp **`char`**) sind jeweils mittels einer globalen Konstanten festgelegt.

```
// Vier Zeichen für den Schlangenkopf
const char snakeHeadLeftCharacter = '<';
const char snakeHeadUpCharacter = '^';
const char snakeHeadRightCharacter = '>';
const char snakeHeadDownCharacter = 'v';
```

Aufgrund der Modularisierung des Spiels, ist auch sofort klar, wo die Änderungen erfolgen müssen, damit die Ausgabe entsprechend umgestaltet wird. Beginnen wir mit dem Aussehen des Schlangenkopfs.

Aussehen des Schlangenkopfs

Die Schlange wird in der Methode **`drawSnake`** in das Array **`gameArea`** eingetragen. Da ja nun der Kopf der Schlange in einer neuen Form eingetragen werden soll, müssen also hier Änderungen vorgenommen werden.

In dieser Methode wird ja mittels einer for-Schleife das Array mit den Positionen der Schlange (**`positionsOfSnake`**) durchlaufen und an der betreffenden Zeilen- und Spalrenposition das bereits definierte Zeichen **`snakeBodyCharacter`** in das Spielfeld-Array **`gameArea`** eingetragen.

Um jetzt den neuen Schalngenkopf in das Spielfeld-Array **`gameArea`** einzutragen, muss diese Methode geändert werden: Erstens muss die Methode nun noch „wissen“, in welcher Richtung sich die Schlange momentan bewegt und zweitens muss das erste Element der Schlangenpositionen mit dem jeweils zugeordneten Zeichen für die Laufrichtung eingetragen werden.

Deshalb wird zuerst der Methodenkopf insofern geändert, dass die Laufrichtung der Schlange ebenfalls als Parameter übergeben wird. Das bedeutet aber, dass nun bei denAufrufen dieser Methode der Parameter mit dem Wert von **`directionOfSnake`** hinzugefügt werden muss!

In der Methode muss nun die erste Positionsangabe mit dem jeweiligen Zeichen für den Schlangenkopf gezeichnet werden. Die **`for`**-Schleife für das Zeichnen des Schlangenkörpers beginnt daher nun mit der zweiten Positionsangabe (Index 1) in dem Array **`positionsOfSnake`**.

```
// Überträgt die Position der Schlange in das Spielfeld
static void drawSnake(int[,] positionsOfSnake,
                              int lengthOfSnake,
                              char[,] gameArea,
                              int directionOfSnake)
{
    // Diese Reihenfolge, da sonst bei Kollision mit Körper
    // der Schlangenkopf nicht mehr zu sehen ist

    // Erst den Schlangenkörper eintragen
    for (int i = 1; i < lengthOfSnake; i++)
    {
        // Zeile und Spalte für Körperelement ermitteln
        int row = positionsOfSnake[i, 0];
        int column = positionsOfSnake[i, 1];
        // Zeichen für Schlangenkörper an der Position eintragen
        gameArea[row, column] = snakeBodyCharacter;
    }

    // Zeile und Spalte für Schlangenkopf ermitteln
    int rowHead = positionsOfSnake[0, 0];
    int columnHead = positionsOfSnake[0, 1];

    // Dann den Schlangenkopf in Abhängigkeit der Richtung eintragen
    switch (directionOfSnake)
    {
        // Links
        case 1:
            gameArea[rowHead, columnHead] = snakeHeadLeftCharacter;
            break;
        // Aufwärts
        case 2:
            gameArea[rowHead, columnHead] = snakeHeadUpCharacter;
            break;
        // Rechts
        case 3:
            gameArea[rowHead, columnHead] = snakeHeadRightCharacter;
            break;
        // Abwärts
        case 4:
            gameArea[rowHead, columnHead] = snakeHeadDownCharacter;
            break;
    }
}
```

Im Anschluss daran wird die Zeilen- und Spaltennummer des Schlangenkopfs bestimmt und mit dem hinzugefügten Parameter **`directionOfSnake`** mittels einer Auswahlstruktur (**`switch`**) das jeweilige Zeichen für den Schlangenkopf in das Spielfeld-Array **`gameArea`** eingetragen.

Allerdings muss zuerst der Schlangenkörper in das Spielfeld-Array **`gameArea`** übertragen werden und dann erst der Schlangenkopf!

Warum? Weil bei einer Kollision der Schlange mit sich selbst, sonst der Kopf durch das Körperelement mit dem der Kopf kollidiert ist, ersetzt wird und der Kopf dann nicht mehr sichtbar ist.

Zuletzt sollen jetzt noch die Farben der einzelnen Spielelemente geändert werden.

Neue Farben braucht das Spiel

Bisher wird alle Spielelemente (Spielfeldrand, Futterhappen und Schlange) in der gleichen Farbe angezeigt. Das soll sich nun ändern!

Auch hier unterstützt uns die bisherige Modularisierung des Spiels. Die Ausgabe einer Spielsituation, die im Array **`gameArea`** gespeichert ist, erfolgt ja in der Methode **`drawScreen`**.

Damit sind lediglich in der Methode **`drawScreen`** Änderungen vorzunehmen! Sehen wir uns die bisherige Methode daher nochmal an.

```
// Gibt den Inhalt des Spielfeldes im Fenster aus
static void drawScreen(char[,] gameArea)
{
    // Einstellen der Farben im Konsolenfenster
    Console.BackgroundColor = gameBackgroundColor;
    Console.ForegroundColor = gameBorderColor;

    // Hier wird der gesamte Inhalt des Array gameArea
    // tatsächlich in das Fenster geschrieben
    for (int row = 0; row < windowHeight - bottomBorder; row++)
    {
        for (int column = 0; column < windowWidth; column++)
        {
            Console.SetCursorPosition(column, row);
            Console.Write(gameArea[row, column]);
        }
    }
}
```

In zwei geschachtelten Zählschleifen wird nun der Inhalt des Arrays **`gameArea`** mittels **`Write`**-Methode Zeichen für Zeichen nach vorheriger Cursorpositionierung ausgegeben.

Die Farben für die Ausgabe werden zu Beginn der Methode festgelegt.

Letztlich wollen wir ja für die verschiedenen Spielelemente verschiedene Vordergrundfarben verwenden, die ja in den zuvor beschriebenen Konstanten definiert wurden.

```
// Gibt den Inhalt des Spielfeldes im Fenster aus
static void drawScreen(char[,] gameArea, int directionOfSnake)
{
    // Einstellen der Farben im Konsolenfenster
    Console.BackgroundColor = gameBackgroundColor;

    // Hier wird der gesamte Inhalt des Array gameArea
    // tatsächlich in das Fenster geschrieben
    for (int row = 0; row < windowHeight - bottomBorder; row++)
    {
        for (int column = 0; column < windowWidth; column++)
        {
            Console.SetCursorPosition(column, row);
            // Auswahl der Farbe für Zeichen
            switch (gameArea[row, column])
            {
                case gameBackgroundCharacter:
                    Console.ForegroundColor = gameForegroundColor;
                    break;
                case gameBorderCharacter:
                    Console.ForegroundColor = gameBorderColor;
                    break;
                case snakeHeadLeftCharacter:
                case snakeHeadUpCharacter:
                case snakeHeadRightCharacter:
                case snakeHeadDownCharacter:
                    Console.ForegroundColor = snakeHeadCharacterColor;
                    break;
                case snakeBodyCharacter:
                    Console.ForegroundColor = snakeBodyCharacterColor;
                    break;
                case snakeFoodCharacter:
                    Console.ForegroundColor = snakeFoodCharacterColor;
                    break;
            }
            Console.Write(gameArea[row, column]);
        }
    }
}
```

Es muss also eine Fallunterscheidung hinsichtlich des auszugebenden Zeichens vorgenommen werden; handelt es sich bei dem auszugebenden Zeichen um das Spielfeldrandzeichen, oder um das Zeichen für einen Futterhappen, um das Zeichen für den Schlangenkopf (eigentlich sind das vier verschiedene Zeichen!) oder um das Zeichen für den Schlangenkörper.

Ganz klar, für solch eine Fallunterscheidung verwenden wir wiederum die **`switch`**-Struktur.

Wir untersuchen das per **`Write`**-Methode auszugebende Zeichen und legen je nach Zeichen die entsprechende Vordergrundfarbe fest. Wenn für mehrere Fälle die selben Anweisungen auszuführen sind, kann man diese Fälle direkt untereinander auflisten, so wie eben bei den vier Zeichen für den Schlangenkopf, der ja immer die selbe Farbe haben soll.

Ausblick

Somit sind wir am Ende des Projekts angelangt. Das Spiel ist fertig und lässt sich spielen, ohne dass es zu Programmabstürzen kommt. Aber wie so oft, sind auch für diesen Stand des Spiels noch weitere „Features“ denkbar. Ich hatte zuvor im Zusammenhang mit dem Spielfeldrand von einem Designfehler gesprochen.

In dieser Version des Spiels ist der Rand im Hinblick auf seine Position quasi fest vorgegeben! Aber es wäre ja auch denkbar, das im bisherigen Aktionsbereich der Schlange weitere Begrenzungen einzubauen. Damit sind auch „schwierigere“ Spielfelder (unterschiedliche Level) möglich, weil die Schlange nicht mehr überall hin bewegt werden kann.

Dies würde sich leicht lösen lassen, in dem man die Positionen der Spielfeldbegrenzung nicht fest einprogrammiert, sondern auch für diese Begrenzungen ein zweidimensionales Array verwendet, in dem die Positionen der Spielfeldbegrenzungen gespeichert werden.

Diese Positionen würde man dann mit einer zusätzlichen Methode **`drawBorder`** in die jeweilige Spielsituation (**`gameArea`**) eintragen.

Denkbar wäre auch, dass einzelne Ränder entfallen und die Schlange dann das Spielfeld beispielsweise am rechten Rand verlassen kann und dann am linken Rand wieder auftaucht.

Eine weitere Modifikation könnte sein, dass die Geschwindigkeit der Schlange bei Erreichen von bestimmten Spielständen zunimmt, indem man der Anweisung **`Thread.Sleep()`** einen kleineren Wert übergibt, der die „Wartezeit“ verkürzt.

Das sind jetzt nur zwei Ideen, wie man das Spiel noch etwas dynamischer gestalten kann.

Sicherlich fällt da jedem noch etwas ein! Daher möchte sich Sie dazu anregen, ihre eigenen Ideen einfließen zu lassen und das nun „fertige“ Projekt zu modifizieren und zu erweitern.

Viel Spaß dabei!