Election: Implementierung von Übung 2 in Node (JavaScript)

Felix Blechschmitt

Dieses Dokument enthält Informationen über den Aufbau und die Funktionen von election.

Getting Started

Das Projekt wurde in der Programmiersprache JavaScript nach dem Standard EcmaScript 6 (ES6) entwickelt. Dabei wurde der Interpreter node in der Version v4.4.4 verwendet.

Installation

Zunächst müssen alle notwendigen Abhängigkeiten installiert werden:

```
$ npm install
```

Usage

Zum Übersetzen der Anwendung sowie zum Starten der einzelnen Applikationen werden NPM-Skripte verwendet, die in der Datei package.json definiert werden.

Ein solches Skript wird über folgendes Kommando ausgeführt. Dabei können optional Parameter übergeben werden:

```
$ npm run <SkriptName> [-- <Parameter>]
```

Ein Netzwerkknoten ist ein Node.js-Skript, das als einzelner Prozess über ein NPM-Skript gestartet werden kann. Zum Starten einer Testsuite, die anhand verschiedener Parameter eine Netzwerktopologie erstellt und automatisiert alle notwendigen Netzwerkknoten startet, dient ein weiteres Node.js-Skript.

Build-Prozess

Node ist nicht in der Lage JavaScript-Dateien, die im ES6-Standard entwickelt wurden, direkt auszuführen. Diese müssen zunächst in den ES5-Standard übersetzt (transpiliert) werden. Hierzu wird babel verwendet.

Der Übersetzungsvorgang kann über das NPM-Skript "build" angestoßen werden. Dabei wird zunächst das möglicherweise bereits vorhandene Verzeichnis "build" geleert bzw. ein solches Verzeichnis erstellt. Anschließend werden alle JavaScript-Dateien von babel in den Standard ES5 übersetzt und inklusive Source Map im Verzeichnis "build" abgelegt.

```
$ npm run build
```

Starten eines Netzwerkknotens

Die Hauptanwendung des Projekts ist ein Netzwerkknoten, der einen Kandidaten bzw. einen Wähler in einem Netzwerk an Wähler- und Kandidatenknoten repräsentiert. Das Ziel der Kandidaten ist es, möglichst viele Wähler von sich zu überzeugen. Sie machen dies, indem sie über den Echo-Algorithmus Kampagnen verbreiten oder über sog. Flooding "Wähl-Mich"-Nachrichten verbreiten. Solche Nachrichten werden wie

ein Gerücht im Netzwerk verbreitet, wobei ein Knoten die Nachricht nur dann weiterleitet, wenn er dem Kandidaten anhand des Confidence-Levels zustimmt.

Ein solcher Netzwerkknoten wird über das NPM-Skript "start-node" gestartet. Die Konfiguration eines Knotens kann als Parameter übergeben werden. Werden keine Parameter angegeben, so werden die notwendigen Parameter abgefragt und für den Rest Default-Werte verwendet:

```
$ npm run start-node
```

Alternativ können die notwendigen Parameter direkt beim Start übergeben werden, sodass die Anwendung ohne weitere Interkation mit dem Nutzer ausgeführt werden kann. Der Parameter "-h" listet alle möglichen Parameter auf:

```
$ npm run start-node -- -h
Usage: node run-node.js

      --endpointFilename=[ARG]  path to the endpoints file, leave blank to map ids to local ports
  -g, --graphFilename=[ARG]     path to the graph file defining the network node topology
      --id=[ARG]                ID of this endpoint
  -r, --receive=[ARG]           number of receives until a candidate starts a new call (e.g. campaign)
      --observer                start observer node
  -h, --help                    Display this help
```

So kann zum Beispiel ein Netzwerkknoten mit der ID 5 wie folgt gestartet werden:

```
npm run start-node -- -g ./config/graphElection.dot --id 5
```

Dieser Knoten ist dann auf localhost:4005 erreichbar. Der Knoten geht von einer Netzwerktopologie wie in der graphviz-Datei config/graph.dot definiert aus. Bei diesem Knoten handelt es sich um einen Wählerknoten, da die ID weder 1 noch 2 ist (was den beiden IDs für die Kandidatenknoten entspricht). Wird keine endpoint-Datei angegeben (wie in diesem Beispiel), so werden die Endpunkte als Lokal angenommen und der Netzwerkknoten mit der kleinsten ID erhält den Port 4001. Für jeden weiteren Knoten wird die Portnummer entsprechend hochgezählt. Die minimale Portnummer (default: 4000) kann über die Konstante MIN_PORT im EndpointManager angepasst werden. Die ID 0 ist für den Beobachter-Prozess reserviert und wird automatisch dem Port 4000 zugewiesen, was ebenso im EndpointManager definiert ist.

Starten des Init-Tools

Analog zur ersten Übung wird auch in diesem Projekt das Init-Tool verwendet, um Kontrollanchrichten an die Prozesse zu senden. Diesmal wird – abgesehen vom Kommando "stop all" zum Beenden aller Prozesse – hauptsächlich das Kommando "msg" verwendet, welches es erlaubt, eine Nachricht des gewünschten Typs an einen oder mehrere Knoten zu senden.

Folgender Aufruf sendet beispielsweise eine INIT-Nachricht an beide Kandidaten, was den Wahlprozess startet und dafür sorgt, dass die Kandidaten damit beginnen, Kampagnen bzw. "Wähl-Mich"-Nachrichten zu verbreiten:

```
$ npm run init -- -c msg -t init --addresses "localhost:4001;localhost:4002" -m "empty"
```

Dabei ist es wichtig, dass ein Inhalt für die Nachricht gesetzt wird. Der Inhalt selbst ist bei einer INIT-Nachricht jeoch irrelevant. Andernfalls wechselt das INIT-Skript in den interaktiven Modus und fordert den Nutzer auf, einen Inhalt einzugeben.

Generieren eines Graphen

Zum Generieren der Netzwerktopologie wurde das Tool graphgen aus Übung 1, welches über das Skript "graphgen" ausgeführt werden kann, erweitert. Dabei wird zunächst mithilfe des Havel-Hakimi-Algorithmus

ein Graph generiert, welcher anschließend analog zu dem Verfahren in Übung 1 in eine graphviz-Datei gespeichert wird.

Auch hier können die benötigten Informationen direkt als Parameter übergeben werden. Eine Hilfe wird auch hier mit dem Parameter "-h" angefordert.

```
$ npm run graphgen -- -h
Usage: node graphgen.js

  -n                      Number of nodes
  -s, --supporter=[ARG]   Number of supporters
  -f, --friends=[ARG]     Number of friends
  -o, --out=[ARG]         Output filename
  -h, --help              Display this help
```

Über den Parameter "-o" bzw. "–out" (output) wird der Pfad angegeben, an dem der erzeugte Graph gespeichert werden soll.

Der Havel-Hakimi-Algorithmus dient ursprünglich dazu, anhand einer gegebenen Sequenz an Knotengraden zu überprüfen, ob es möglich ist, einen Graphen zu generieren, der genau die in der Sequenz angegebenen Knotengrade besitzt. Eine mögliche Sequenz wäre beispielsweise [4, 3, 3, 3, 1]. Sie gibt an, dass der Graph fünf Knoten besitzt, wobei ein Knoten vier Kanten hat, drei Knoten jeweils drei Kanten und ein Knoten lediglich eine Kante besitzt.

In dieser Anwendung wird der Algorithmus verwendet, um die Eingabeparameter zu verifizieren und damit sicher zu sein, dass ein Graph erzeugt werden kann, bei dem jeder Wähler f Nachbarknoten besitzt. Zusätzlich wurde der Algorithmus dahingehend erweitert, dass sich die Anwendung die im Ablauf verbundenen Kanten merkt, sodass nach einem Durchlauf des Algorithmus der Graph erzeugt wurde.

Auf diese Weise wird das Wählernetz erzeugt, sodass jeder Wähler exakt f Freunde hat. Um sicherzugehen, dass der resultierende Graph zusammenhängend ist, wird der erzeugte Graph anschließend mit der Node-Bibliothek "connected-components" überprüft. Stellt sich heraus, dass es sich um keinen zusammenhängenden Graphen handelt, wird die Eingabe verworfen und der Benutzer dazu aufgefordert, andere Eingabeparameter zu verwenden.

Nachdem das Wählernetz erzeugt wurde, werden die beiden Kandidatenknoten hinzugefügt und diesen ihre Parteifreunde zugewiesen. Dazu wird ein zufälliger Knoten ausgewählt, dessen ID größer als 0 ist und kleiner als die maximale Anzahl an Knoten abzüglich der Anzahl an Parteifreunde *s*. Dieser Knoten sowie die *2* s* Folgeknoten werden alternierend den beiden Kandidaten als Parteifreunde zugewiesen. Die Parteifreunde haben somit in der Summe einen Nachbarn mehr als normale Wählerknoten.

Ein Nachteil dieser Lösung ist jedoch, dass bei gleichen Eingabeparametern annähernd gleiche Graphen erzeugt werden.

Starten der Testsuite

Um eine Wahl durchführen zu können, ist es notwendig, dass alle Wählerknoten sowie die Kandidatenknoten und ein Observerprozess gestartet werden. Hierfür kann die Node-Anwendung "run-election.js" verwendet werden, welche über das NPM-Skript "start-election" ausgeführt wird. Als Parameter können u.A. die Anzahl an Knoten, die Anzahl an Freunde und Parteifreunde der Wählerknoten übergeben werden. Der Parameter "-h" zeigt eine Übersicht der möglichen Argumente:

```
$  npm run start-election -- -h
Usage: node run-election.js

  -n, --nodes=[ARG]          number of nodes (without the observer node)
  -s, --supporters=[ARG]     number of supporters per candidate
  -f, --friends=[ARG]        number of friends for each voter
```

```
-r, --receives=[ARG]       number of receives until a candidate starts a new call (e.g. campaign)
-g, --graphFilename=[ARG]  path to the graph file defining the network node topology
-d, --delay=[ARG]          delay between initiating the election process and taking the snapshot (in s
-h, --help                 Display this help
```

Hinweis: werden keine Parameter angegeben, so werden die in "run-election.js" definierten Default-Werte verwendet.

Die Testsuite startet zunächst das graphgen-Tool, um ein Wählernetz passend zur gegeben Konfiguration zu erzeugen. Anschließend werden die einzelnen Knoten als eigenständige Kindprozesse gestartet. Dabei wird die Ausgabe jedes einzelnen Prozesses überwacht und auf die Standardausgabe des Hauptprozesses umgelegt. Sobald durch das Überwachen der Ausgabe erkannt wird, dass alle Knoten bereit sind, wird der Beobachter-Prozess gestartet und danach mithilfe des INIT-Skripts eine INIT-Nachricht an die beiden Kandidaten gesendet. Nach einer konfigurierbaren Verzögerung wird der Beobachter-Prozess angewiesen einen konsistenten Schnappschuss durchzuführen. Sobald ein Wahlergebnis verfügbar ist, werden alle Prozesse beendet und der Vorgang ist abgeschlossen.

Bei einem Durchlauf werden sehr viele Nachrichten zwischen den einzelnen Knoten ausgetauscht. Da jeder Knoten alle ein- sowie ausgehenden Nachrichten auf der Ausgabe protokolliert, wird die Ausgabe der Testsuite schnell sehr unübersichtlich. Aus diesem Grund wird empfohlen, die Ausgabe in eine Datei umzulenken, bzw. zum Beispiel mithilfe des *tee*-Kommandos die Ausgabe sowohl anzuzeigen als auch gleichzeitig in eine Datei zu schreiben. Bei der Ausgabe einer Log-Nachricht schreibt jeder Knoten seine eigene ID in runde Klammern. Dies kann man ausnutzen, um nur die Nachrichten eines bestimmten Knotens auszugeben. Folgendes Listing zeigt, wie die Testsuite gestartet wird und anschließend ledigliche die Ausgabe des Beobachter-Prozess angezeigt wird, wobei sich die gesamte Ausgabe in der Datei log/out.log befindet:

```
$ npm run start-election -- -n 8 -s 2 -f 3 -r 3 -g ./config/graphElection.dot > ./log/out.log 2> ./log/
$ tail -f ./log/out.log | grep "(0)"
```

Aufbau

Das Projekt wurde nach einem stark modularisierten Konzept entwickelt. Im Folgenden werden die Verzeichnisstruktur, die einzelnen Komponenten sowie das verwendete Nachrichtenprotokoll beschrieben.

Verzeichnisstruktur

Die Source-Dateien befinden sich alle im Verzeichnis "src" im Hauptordner des Projekts. Prinzipiell kann eine gesamte Node-Anwendung in einer einzelnen JavaScript-Datei entwickelt werden. Dies ist jedoch aufgrund fehlender Übersicht nicht empfehlenswert. Daher wurde das Projekt in verschiedene Komponenten unterteilt, welche in einzelnen Dateien entwickelt wurden. Ähnliche Module wurden dabei in Unterverzeichnissen gruppiert:

- /election: enhält die verschiedenen Klassen, die bei dem Wahlprozess miteinander interagieren (Wähler: voter.js, Kandidat: candidate.js usw.)
- /lib: enthält wiederverwendbare Module, die anwendungsübergreifend verwendet werden können
- /lib/algorithm: enthält die Implementierungen der einzelnen Algorithmen als wiederverwendbare Module
- /network-core: enthält die Grundmodule der Server-Client-Architektur
- /parser: enthält Komponenten zum Einlesen von Dateien bestimmter Formate
- /process-helpers: enthält Komponenten zum Starten von Node-Prozessen
- /tools: enthält zusätzliche Skripte, die als eigenständige Node-Anwendungen gestartet werden können, sowie deren Module

Die Datei "run-node.js" ist der Einstiegspunkt der Anwendung. Darin werden die einzelnen Komponenten miteinander verknüpft und der Netzwerkknoten gestartet.

Der Einstiegspunkt der Testsuite befindet sich in der Datei “run-election.js”. Dort werden – wie im vorherigen Abschnitt beschrieben – mehrere Netzwerkknoten gestartet, der Wahlablauf initialisiert und der konsistente Schnapschuss durchgeführt.

Komponenten

Das folgende UML-Klassendiagramm zeigt einen Überblick über die einzelnen Komponenten der Anwendung.

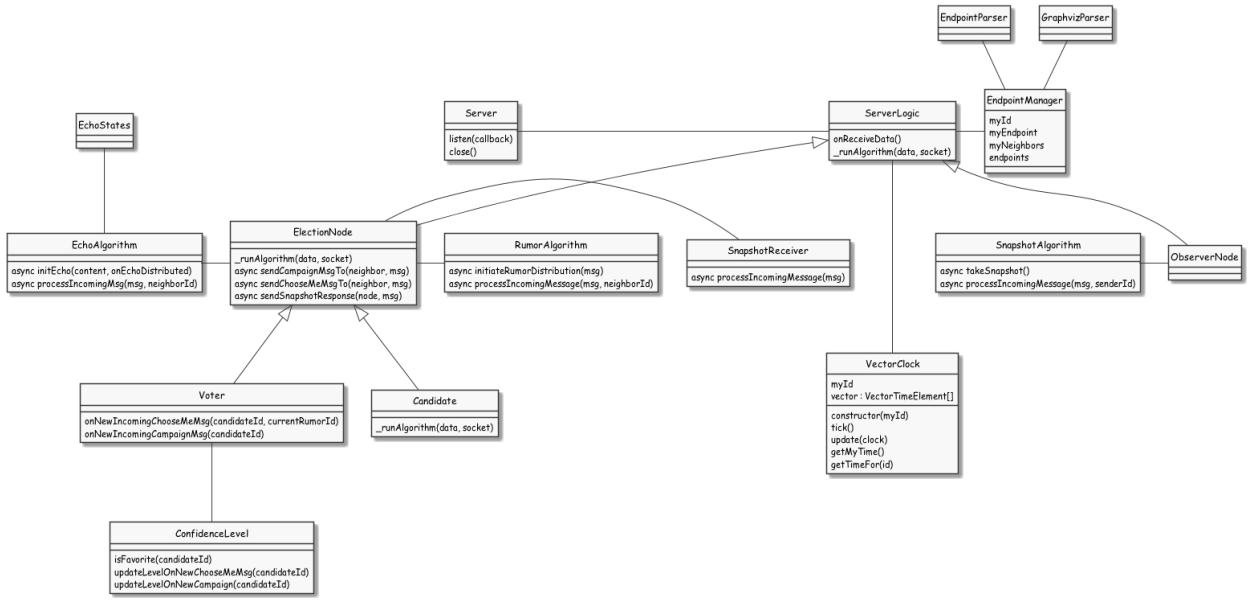

Figure 1: Komponenten der Anwendung

Die Grundkomponenten zum Ausführen des TCP Servers wurden aus der ersten Übung übernommen. Auf Grundlage der Klasse “ServerLogic” wurde für den Observer-Knoten die Erweiterung “ObserverNode” und für die Kandidaten- und Wählerknoten die Erweiterung “ElectionNode” entwickelt.

Die zu implementierenden Algorithmen wurden jeweils als einzelne Module entwickelt, welche zum Beispiel in eine Server-Implementierung “eingehängt” werden können. Ein solches Modul ist dadurch lediglich für den Ablauf des zu implementierenden Algorithmus verantwortlich und besitzt keinerlei Logik zum Austauschen von Nachrichten. Hierfür stellt das jeweilige Modul eine Schnittstelle zur Verfügung, die von dem Modul, das den Algorithmus verwenden möchte, implementiert werden muss. Dies ermöglicht eine Entkopplung zwischen dem Algorithmus und der konkret verwendeten Variante zum Austausch von Nachrichten. Derjenige, der ein solches Modul verwendet, kann also entscheiden, ob die zum Ablauf des Algorithmus ausgetauschten Nachrichten z.B. über TCP, UDP oder gar durch einen einfachen Methodenaufruf beim Empfänger (falls beide Knoten durch Threading innerhalb eines Prozesses ausgeführt würden) übertragen werden. Eine solche Entkopllung ist gerade bei der Verwendung von automatisierten Tests enorm von Vorteil, da so die einzelnen Knoten durch verschiedene Objekte simuliert und der Nachrichtenaustausch in den Testfällen durch entsprechende Methodenaufrufe ausgeführt werden konnte.

Vektorzeit

Damit in der Ausgabe ersichtlich ist, in welcher Reihenfolge die – teilweise parallel abgesetzen – Nachrichten gesendet wurden, wird eine logische Zeit eingeführt: Die Vektorzeit. Diese Zeiteinheit besteht aus n-Zählern, wobei n gerade der Anzahl an Netzwerkknoten entspricht. Die Vektorzeit ist Teil einer jeden Nachricht, die versendet wird. Der eigene Zähler wird genau dann erhöht, wenn ein lokales Ereignis passiert (beispielsweise

das Senden oder Empfangen einer Nachricht). Beim Empfangen einer Nachricht werden außerdem die Werte aller Zähler verglichen und das Maximum übernommen.

Implementiert wurde die Vektorzeit als eigenes Modul. Die Klasse VectorClock beinhaltet ein assoziatives Array, welches die Zähler für verschiedene Netzwerkknoten, die anhand einer eindeutigen ID identifiziert werden, speichert. Tritt ein lokales Ereignis auf, so wird der eigene Zähler mittels der Methode "tick(()" erhöht. Beim Empfangen einer Nachricht wird der darin übermittelte Vektor an die Methode "update(clock)" übergeben, was dafür sorgt, dass das Maximum der jeweiligen Zähler übernommen wird. Im initialen Zustand kennt das Modul lediglich den eigenen Zählerstand. Das Vektorfeld wird mit jeder ankommenden Nachricht von einem vorher noch unbekannten Netzwerkknoten entsprechend angepasst und erweitert.

Protokoll

Zum Austausch der Nachrichten zwischen den einzelnen Netzwerkknoten wird das JSON-Format verwendet. Eine Nachricht hat dabei folgende Komponenten:

- type: Typ der Nachricht (control, snapshot, campaign, choose-me, not-you, keep-it-up)
- msg: Inhalt der Nachricht
- from: ID des Absenderknotens (optional)
- time: Vektorzeit des Absenderknotens

Handelt es sich um eine Kontrollnachricht, so enthält das Messagefeld die Aktion ("STOP" zum Beenden des Knotens bzw. "STOP ALL" zum Senden der "STOP ALL" Nachricht an alle Nachbarn und zum anschließenden Beenden des Knotens).

Alle Nachrichten werden über TCP übertragen. Auch wenn dies – aufrund der TCP-Übertragung – nicht explizit notwendig wäre, werden alle empfangenen Nachrichten nach dem Erhalt mit einem leeren JSON-Objekt bestätigt.

Die einzelnen Algorithmen verwenden eine Nachricht eines bestimmten Typs. Alle Nachrichten, die der Echo-Algorithmus versendet, werden beispielsweise unter dem Nachrichten-Typ "campaign" versendet. Der Payload einer solchen Echo-Nachricht wird im Nachrichtenfeld "msg" übertragen. Dieses Feld ist je nach verwendetem Nachrichtentyp ebenfalls ein JSON-Objekt.

Beispiel einer EXPLORER-Nachricht des Echo Algorithmus

```
{"msg":{"id":"hn2CFNx","content":2,"type":"explorer"},"from":22,"type":"campaign": "time":{"myId":0,"ve
```

Beispiel einer Kontrollnachricht

```
{"msg":"STOP","type":"control"}
```

Tests

Die dynamische Typisierung von JavaScript birgt die Gefahr von häufigen Laufzeitfehlern. Aus diesem Grund ist es gerade bei der Verwendung einer solchen Programmiersprache besonders wichtig, die Komponenten der Anwendung mit automatisierten Tests auf korrekte Funktionalität zu überprüfen. Außerdem helfen gerade Unit-Tests bei der Entwicklung von einzelnen Komponenten. Mithilfe von Unit-Tests kann man sich von der korrekten Funktionalität einzelner Komponenten überzeugen, da diese dann gezielt ohne den Kontext der gesamten Anwendung ausgeführt werden können.

Automatisierte Tests

Einige Hilfsfunktionen, kleinere Module der Anwendung, wie zum Beispiel der Parser für die Endpoint- bzw. graphviz-Datei oder der Kantengenerator sowie die einzelnen Algorithmen wurden mit Unit-Tests versehen. Für den Echo-Algorithmus wurde darüber hinaus noch eine Test-Suite entwickelt, die verschiedene Knoten als JavaScript-Objekte darstellt und das Versenden der Nachrichten über Methodenaufrufe simuliert. Auf diese Weise kann die Funktionalität des Algorithmus getestet werden, ohne, dass verschiedene Prozesse gestartet werden müssen und eine funktionierende TCP-Verbindung vorausgesetzt werden muss.

Die Tests befinden sich im Verzeichnis "tests" des Projektordners. Damit die Quelldateien mit den zugehörigen Testdateien leicht verbunden werden können, ist der "tests"-Ordner nach der gleichen Verzeichnisstruktur aufgebaut wie der "src"- Ordner.

Die Tests können mithilfe des NPM-Skripts "test" ausgeführt werden:

```
$ npm run test
```

Test Coverage

Bei der Verwendung von automatisierten Tests ist es außerdem sinnvoll, ein Tool zu verwenden, das die Testabdeckung überprüft. Ein solches Tool überprüft beim Durchlaufen der Tests, welche Codezeilen, Funktionen, Statements und Verzweigungen durchlaufen werden und berechnet daraus wie groß die Testabdeckung ist. Schaut man sich den Bericht einer solchen Auswertung an, kann man ablesen, wo sich potentielle Fehler verstecken.

Betrachtet man lediglich den berechneten Wert der Testabdeckung, so kann dies unter Umständen täuschen, da lediglich die vom Test ausgeführten Module bei der Berechnung berücksichtigt werden. So kann es also vorkommen, dass eine sehr gute Testabdeckung angegeben wird, obwohl manche Module überhaupt gar nicht getestet werden.

Continuous Integration & Code Quality

Im Rahmen dieses Projekts wird TravisCI als Continuous Integration System verwendet. Dieses System ist mit dem github repository verknüpft und sorgt bei jeder Änderung der Daten im Repository dafür, dass die Anwendung erstellt wird und alle automatisierten Tests ausgeführt werden. Die Datei .travis.yml beschreibt die Konfiguration des Testservers und besagt, welcher Interpreter in welcher Version verwendet werden soll. Nachdem die automatisierten Tests von TravisCI durchgeführt wurden, wird das Ergebnis der Testabdeckung an einen Code Climate gesendet.

Code Climate bereitet den Bericht der Testabdeckung graphisch auf und ermöglicht es zu analysieren, welche Module nicht ausreichend mit Tests abgedeckt sind. Weiterhin lässt sich Code Climate selbst mit einem github repository verknüpfen und kann dann eine Überprüfung der Code-Qualität anhand verschiedener Kriterien durchführen und das Projekt danach bewerten. So werden beispielsweise Code-Duplikationen aufgedeckt oder vor zu komplexen Funktionen gewarnt.