

# Web Engineering II

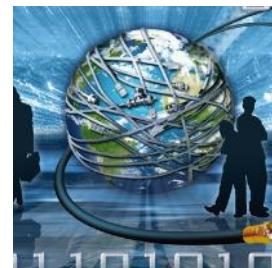
## 06 Debugging und Testen

Johannes Konert



BEUTH HOCHSCHULE  
FÜR TECHNIK  
BERLIN

University of Applied Sciences





## Agenda

- **Wiederholung und Bonusfrage**
  
- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**
  
- **Debug-Vorgehen: 6 Stufenplan**
  - 3. Minimierung
  - 4. Hypothesen
  - 5. Analysewerkzeuge  
Code editieren
  - 6. Testen und Testwerkzeuge
  
- **Abschließende kurze Übung**
- **Zusammenfassende Fragen**
- **Ausblick**



## Zusammenfassende Fragen

„Pick-and-Challenge“



1. Für welche zwei unterschiedlichen Programmier-Ziele werden `app.route(...)` und `express.Router()` verwendet?
2. Wann bietet es sich an `nodemon` statt `node` zu verwenden?
3. Was ist der Unterschied zwischen den Methoden `app.use(...)` und `app.all(...)`?
4. Wie können Teile der URL im `pattern` bei `app.all('pattern', function handler)` als Variable deklariert werden, um diese im Handler-Code dann zu nutzen als Variable im Code genutzt werden (in `request.params`)?
5. Welche Unterschiede bestehen zwischen `request.params` und `request.query`?
6. Wie wandeln Sie den HTTP Body, der bei einem POST an Ihren Server geschickt wird, wieder von JSON String nach JavaScript Object um?
7. Warum ist es wichtig, dass der Server-Start mittels `app.listen(..)` als letzter Aufruf in Ihrer `app.js` erfolgt?
8. Welche Module installieren Sie global, welche als devDependencies und welche als normale dependencies in Ihrem Projekt? (bspw. `nodemon`, `bodyparser` und `node-minify`)
9. Wozu dient der Aufruf von `next()` in einer Handler-Funktion? Welcher Unterschied besteht zwischen `next()` und `next(obj)` ?
10. Wozu dient `"use strict";` in einer JavaScript-Datei?
11. Warum ist es essentiell, dass Sie im Server auf jeden (auch ungültige) Request eine Antwort senden? Wie machen Sie das? Welchen Statuscode verwenden Sie bei ungültigen Anfragen?
12. **Bonusfrage:** Wie können Sie mehrere Handlerfunktionen in express für die gleiche Route-URL angeben (die sich dann nacheinander mit `next()` aufrufen)? Wie können Sie umgekehrt für mehrere Routes die gleiche Handlerfunktion angeben? (in beiden Fällen ohne copy&paste)



## Wiederholung/Bonusfrage

- Wie können Sie mehrere Handlerfunktionen in express für die gleiche Route-URL angeben (die sich dann nacheinander mit next()) aufrufen?)
- Wie können Sie umgekehrt für mehrere Routes die gleiche Handlerfunktion angeben? (in beiden Fällen ohne copy&paste)

```
app.get(['/time', '/servertime'], function (req, res, next) {  
    res.set('Content-Type', 'text/plain')  
    .send(new Date().toLocaleTimeString());  
});
```

oder

```
app.get('/time', sendServerTime);  
app.get('/servertime', sendServerTime);  
  
function sendServerTime(req, res, next) {  
    res.set('Content-Type', 'text/plain')  
    .send(new Date().toLocaleTimeString())  
}
```



## Wiederholung/Bonusfrage

- Wie können Sie mehrere Handlerfunktionen in express für die gleiche Route-URL angeben (die sich dann nacheinander mit `next()`) aufrufen?

```
app.get('/file.txt', startTime, appendFile, function(req, res, next) {  
    var timeNeeded = process.hrtime(res.startTime);  
    res.write('\n' + timeNeeded[1] + 'ns').end();  
});  
// Helper function (later in extra modules) *****  
function appendFile(req, res, next) {  
    fs.readFile('file.txt', function(err, data) {  
        if (err){  
            err.status = 500; // internal server error  
            next(err); // give err to our standard error response code...  
            return;  
        }  
        res.write(data);  
        next();  
    });  
}  
function startTime(req, res, next) {  
    res.locals.startTime = process.hrtime();  
    next();  
}  
}]}  
WE2 – 06 Debugging und Testen – Prof. Dr.-Ing. J. Konert – SoSe17
```



## Agenda

- **Wiederholung und Bonusfrage**

- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**

- **Debug-Vorgehen: 6 Stufenplan**
  - 3. Minimierung
  - 4. Hypothesen
  - 5. Analysewerkzeuge  
Code editieren
  - 6. Testen und Testwerkzeuge

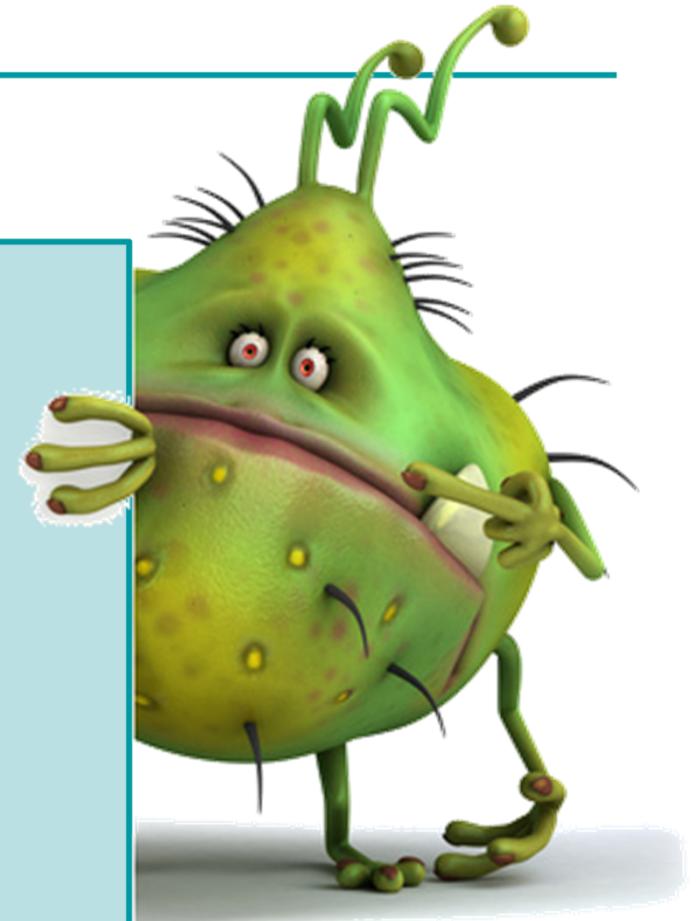
- **Abschließende kurze Übung**
- **Zusammenfassende Fragen**
- **Ausblick**



## Was ist ein Bug?

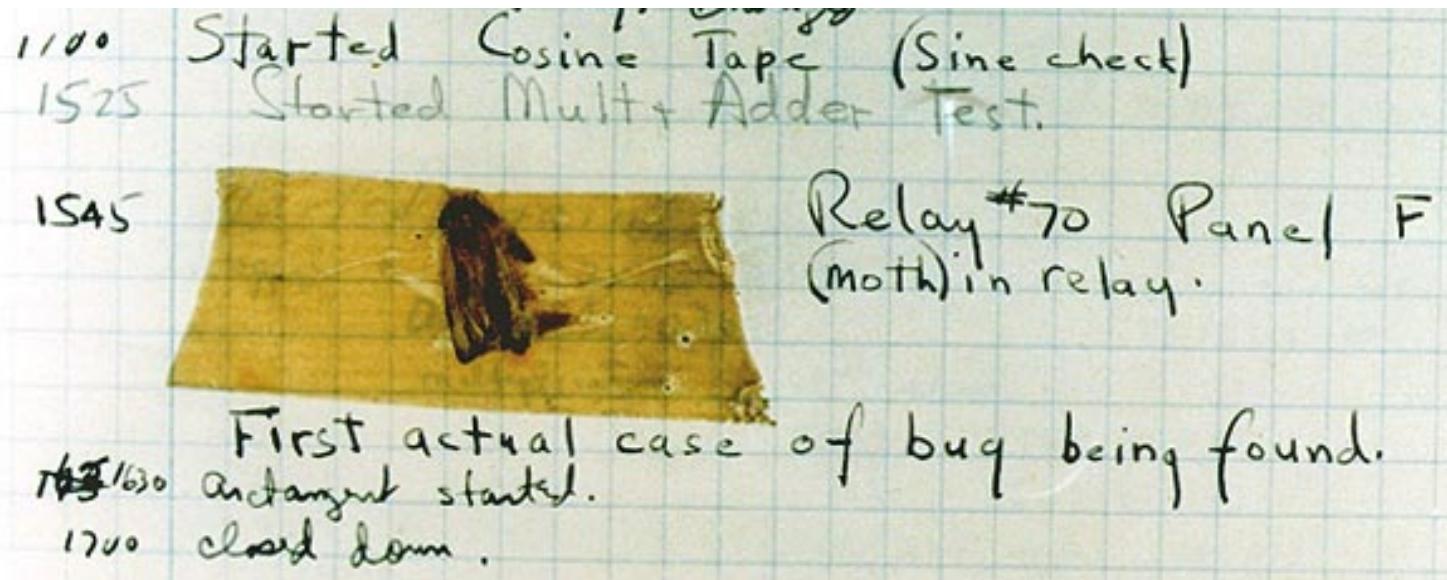
### Bug

- **Nichterfüllung einer Anforderung**  
(EN ISO 9000:2005)
- **Abweichung** eines tatsächlichen Verhaltens (**IST**) vom erwarteten (**SOLL**)





## Der erste Bug?



- 1947 eine Motte im Relay einer Rechenmaschine
- Wird Grace Hopper zugeschrieben

## Tatsächlich jedoch

- u.a. Thomas Edison (1878) schreibt Knackgeräusche im Telefon einem Käferknabbern zu



## Agenda

- **Wiederholung und Bonusfrage**

- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**

- **Debug-Vorgehen: 6 Stufenplan**
  - 3. Minimierung
  - 4. Hypothesen
  - 5. Analysewerkzeuge
  - Code editieren
  - 6. Testen und Testwerkzeuge

- **Abschließende kurze Übung**
- **Zusammenfassende Fragen**
- **Ausblick**



## Rollen bei der Softwareentwicklung als Informatiker/in

Aufgabe	Rolle
Design	Architekt/in
Coding	Ingenieur/in
Testen	Vandale/in
Debugging	Detektiv/in

## Zeitaufwand bei Erfüllen der Rollen

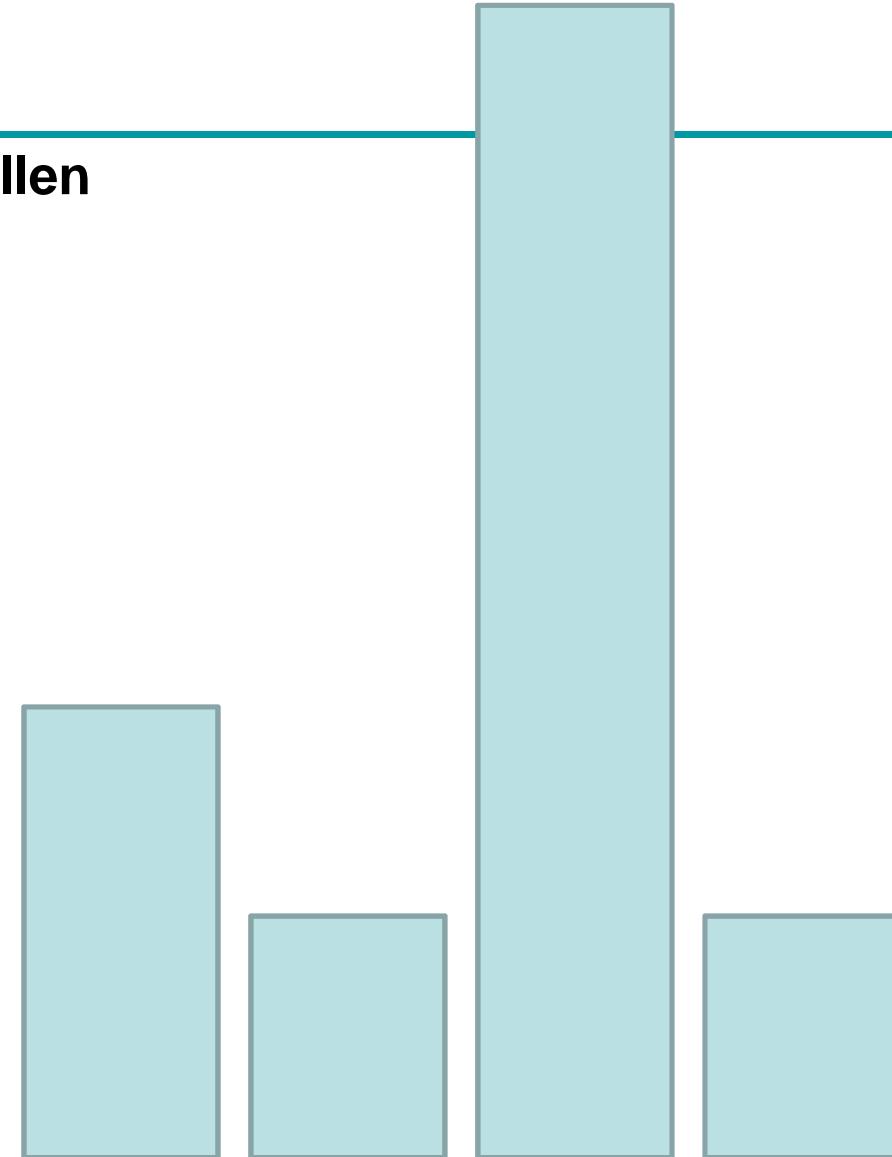


**Design**

**Coding**

**Testen**

**Debuggen**



## Zeitaufwand bei Erfüllen der Rollen

### Debuggen ist

- eine **wesentliche Fähigkeit** des/der Softwareentwicklers/in
- mindestens **doppelt so schwer** wie Programmieren
- Auch für professionelle Programmierer/innen eine **wesentliche Aufgabe**
- also Zeitaufwand kein Zeichen von schlechten Programmierfähigkeiten





## Agenda

- **Wiederholung und Bonusfrage**

- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**

- **Debug-Vorgehen: 6 Stufenplan**

3. **Minimierung**
4. **Hypothesen**
5. **Analysewerkzeuge**  
**Code editieren**
6. **Testen und Testwerkzeuge**

- **Abschließende kurze Übung**

- **Zusammenfassende Fragen**

- **Ausblick**

## 5 Teile einer Bug-Beschreibung

**Beschreiben:**

- a. Ziel**
- b. Annahmen (Eingabe, Kontext)**
- c. Code**
- d. Erwartetes Verhalten (SOLL)**
- e. Tatsächliches Verhalten (IST)**

**Effektiv für Lösungsfindung:**

Bug (**a-e**) sich selbst  
oder anderen beschreiben  
(Mirror Talking)





## Ziel heute

- **Finden des Fehlers in der REST-API „miniTwitter“**

The screenshot shows the Postman interface. At the top, there is a search bar with the URL `http://localhost:300...` and a plus icon for adding new environments. To the right, it says "No environment". Below the search bar, there are tabs for "GET", "http://localhost:3000/tweets", "Params", and a blue "Send" button with a dropdown arrow. Underneath these tabs, there are five categories: "Authorization", "Headers (1)", "Body", "Pre-request script", and "Tests". The "Headers (1)" tab is currently selected, indicated by a thick black underline. Below it, there is a table with one row. The first column has a checked checkbox next to "Accept" and the value "application/json" in the "Value" column. There is also a small edit icon in the "Value" column. The "Header" column contains the header name "Accept".

## Could not get any response

This seems to be like an error connecting to <http://localhost:3000/tweets>.



## A. Zielbeschreibung (in einem Satz)

- Der Node.js Server soll die falsche Angabe eines Accept-Version Headers erkennen und einen Fehler ausgeben



## B. Annahmen

- Der Request kommt korrekt bei `node.js` an
- Der neue Middlewarehandler wird beim Request immer aufgerufen (`app.use(..)`)
- Im Request-Objekt unter `req.get('Accept-Version')` steht der Wert des Headerfeldes
- Der Code prüft auf `!== 1.0` und setzt beim Response-Objekt mit `res.sendStatus(406)` einen Fehlercode
- Ansonsten wird über `andere Handler` der REST-Request bearbeitet
- Die gesetzten `Response-Werte` des Codes sind `korrekt`
- `Node.js` liefert die Antwort aus
- ...



## B. Annahmen

**Typische fehlerhafte Annahmen** beim Debuggen sind

- Ich habe keine Tippfehler im Code
- Bedingungen (if, ..) sind korrekt und tun, was sie sollen
- Eingabewerte (Parameter) sind wie erwartet
- Meine Ausgabewerte sind wie vom Rest des Codes erwartet
- Die von mir benutzen Methoden funktionieren wie erwartet
- Die eingesetzten Libraries, Frameworks, Betriebssysteme, Hardware funktioniert wie erwartet (Komponenten-Stack)



## C. Code

```
"use strict";

// node module imports
var path = require('path');
var express = require('express');
var bodyParser = require('body-parser');

// own modules imports
var store = require('./blackbox/store.js');

// creating the server application
var app = express();

// Middleware ****
app.use(express.static(path.join(__dirname, 'public')));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

// logging
app.use(function(req, res, next) {
  console.log('Request of type ' + req.method + ' to URL ' + req.originalUrl);
  next();
});

// API-Version control. We use HTTP Header field Accept-Version instead of URL-part /v1
app.use(function(req, res, next){
  // expect the Accept-Version 1.0
  var versionWanted = req.get('Accept-Version');
  if (versionWanted === undefined || versionWanted !== '1.0') {
    // 406 Accept-* header cannot be fulfilled.
    res.status(406);
  } else {
    next();
  }
});

// request type application/json check
app.use(function(req, res, next) {
  if ([ 'POST', 'PUT' ].indexOf(req.method) > -1 &&
    !( /application\/json/.test(req.get('Content-Type')) ) ) {
    // send error code 415: unsupported media type
    res.status(415).send('wrong Content-Type'); // user has SEND the wrong type
  } else if (!req.accepts('json')) {
    // send 406 that response will be application/json and request does not support it by i
    // user has REQUESTED the wrong type
    res.status(406).send('response of application/json only supported, please accept this')
  } else {
    next(); // let this request pass through as it is OK
  }
});

// Routes ****
app.get('/tweets', function(req,res,next) {
  res.json(store.select('tweets'));
});

app.post('/tweets', function(req,res,next) {
  var id = store.insert('tweets', req.body); // TODO check that the element is really a tweet
  // set code 201 "created" and send the item back
  res.status(201).json(store.select('tweets', id));
});

app.get('/tweets/:id', function(req,res,next) {
  res.json(store.select('tweets', req.params.id));
});

app.delete('/tweets/:id', function(req,res,next) {
  store.remove('tweets', req.params.id);
  res.status(200).end();
});

app.put('/tweets/:id', function(req,res,next) {
  store.replace('tweets', req.params.id, req.body);
  res.status(200).end();
});

// CatchAll for the rest (unfound routes/resources ****)
// catch 404 and forward to error handler
app.use(function(req, res, next) {
  var err = new Error('Not Found');
  err.status = 404;
  next(err);
});

// error handlers (express recognizes it by 4 parameters!)
// development error handler
// will print stacktrace as JSON response
if (app.get('env') === 'development') {
  app.use(function(err, req, res, next) {
    console.log('Internal Error: ', err.stack);
    res.status(err.status || 500);
    res.json({
      error: {
        message: err.message,
        error: err.stack
      }
    });
  });
}

// production error handler
...
```



## D. Erwartetes Verhalten (SOLL) (Code-Verhalten, Alle Fälle)

- i. Wenn ich einen GET-Request mit gesetztem Header-Feld „Accept-Version: 1.0“ aus Postman heraus an localhost:3000/tweets absetze, dann kommen die tweets als JSON-Antwort zurück
- ii. Lasse ich das Feld aus oder sende eine andere Versionsnummer kommt kein BODY und im Response Header ist Status-Code 406 gesetzt.

## E. Tatsächliches Verhalten (IST)

- i. Fall i.) funktioniert
- ii. Fall ii.) liefert keine Antwort. Der Browser wartet ewig auf die Antwort.



## Agenda

- **Wiederholung und Bonusfrage**

- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**

- **Debug-Vorgehen: 6 Stufenplan**

3. Minimierung
4. Hypothesen
5. Analysewerkzeuge  
Code editieren
6. Testen und Testwerkzeuge

- **Abschließende kurze Übung**
- **Zusammenfassende Fragen**
- **Ausblick**

## Debug-Vorgehen

### Ein 6 Stufenplan:

- 1. Bug beschreiben** (u.a. SOLL und IST)
- 2. Stabile Reproduzierbarkeit herstellen**
- 3. Isolieren** und „Minimized Example“ erstellen
- 4. Hypothesen** formulieren zur Frage:  
„Wo könnte der Bug herkommen?“
  - a. Das wahrscheinlichste zuerst prüfen
  - b. Die einfache These bevorzugen vor der komplizierten
  - c. Mit jeder geprüften These 50% der Fehlerquellen ausschließen\*\*
- 5. Beheben durch Analysieren**  
Nicht geklappt? **4.** wiederholen.  
Wenn nichts mehr übrig,  
**3.** wiederholen, **4.** detaillierter (Annahmen hinterfragen!)
- 6. Doku und Tests:** **2.** und **3.** rückgängig machen und testen (ideal: Unit-Test)



\*\* Konzept des Divide und Conquer (Teile und (Be)herrsche)



## Stufenplan: 3. Isolieren und Minimieren

### ■ Code-Mini-Example (wo der Bug noch auftritt)

```
"use strict";
// module imports
var express = require('express');
var store = require('./blackbox/store.js');

var app = express();
// ...
// API-Version control. We use HTTP Header field Accept-Version instead of URL-part /v1/
app.use(function(req, res, next){
    // expect the Accept-Version 1.0
    var versionWanted = req.get('Accept-Version');
    if (versionWanted === undefined || versionWanted !== '1.0') {
        // 406 Accept-* header cannot be fulfilled.
        res.status(406);
    } else {
        next();
    }
});
// Routes ****
app.get('/tweets', function(req,res,next) {
    res.json(store.select('tweets'));
});

// Start server ****
app.listen(3000); // no callback
```



## Agenda

- **Wiederholung und Bonusfrage**
  
- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**
  
- **Debug-Vorgehen: 6 Stufenplan**
  - 3. Minimierung
  - Hypothesen**
  - 5. Analysewerkzeuge
  - Code editieren
  - 6. Testen und Testwerkzeuge
  
- **Abschließende kurze Übung**
- **Zusammenfassende Fragen**
- **Ausblick**



## Stufenplan: 4. Hypothese formulieren

### ■ Wo könnte der Bug liegen?

- a. Das wahrscheinlichste zuerst prüfen
- b. Die einfache These bevorzugen vor der komplizierten
- c. Mit jeder geprüften These 50% der Fehlerquellen ausschließen (idealerweise)



### Aufgabe:

- 1. Formulieren Sie in Abstimmung mit Ihrem Banknachbarn ZWEI wahrscheinliche Hypothesen, weswegen der Bug wohl auftreten könnte. (2min)**
- 2. Danach Tafelsammlung**

### ■ Beispiele:

- Hohe Wahrscheinlichkeit: „*Der Server läuft gar nicht und die erste Antwort kommt aus einem Cache*“
- Geringe Wahrscheinlichkeit: „*Es wird eine Antwort gesendet, aber Postman ‘kennt’ den Statuscode 406 nicht und hängt daher*“

## Stufenplan: 4. Hypothese formulieren // Code zur Aufgabe

```
"use strict";
// module imports
var express = require('express');
var store = require('./blackbox/store.js');

var app = express();
// API-Version control. We use HTTP Header field Accept-Version
app.use(function(req, res, next){
    // expect the Accept-Version 1.0
    var versionWanted = req.get('Accept-Version');
    if (versionWanted === undefined || versionWanted !== '1.0') {
        // 406 Accept-* header cannot be fulfilled.
        res.status(406);
    } else {
        next();
    }
});
// Routes ****
app.get('/tweets', function(req,res,next) {
    res.json(store.select('tweets'));
});
// Start server ****
app.listen(3000); // no callback
```



## Von der Hypothese zum Nachweis: Den Bug finden

### Ein 6 Stufenplan:

- 1. Bug beschreiben** (u.a. SOLL und IST)  

- 2. Stabile Reproduzierbarkeit herstellen**
- 3. Isolieren** und „Minimized Example“ erstellen
- 4. Hypothesen** formulieren zur Frage:  
„Wo könnte der Bug herkommen?“
  - a. Das wahrscheinlichste zuerst prüfen
  - b. Die einfache These bevorzugen vor der komplizierten
  - c. Mit jeder geprüften These 50% der Fehlerquellen ausschließen\*\*
- 5. Beheben durch Analysieren**  
Nicht geklappt? **4.** wiederholen.  
Wenn nichts mehr übrig,  
**3.** wiederholen, **4.** detaillierter (Annahmen hinterfragen!)
- 6. Doku und Tests:** **2.** und **3.** rückgängig machen und testen (ideal: Unit-Test)

\*\* Konzept des Divide und Conquer (Teile und (Be)herrsche)



## Agenda

- **Wiederholung und Bonusfrage**
  
- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**
  
- **Debug-Vorgehen: 6 Stufenplan**
  - 3. Minimierung
  - 4. Hypothesen
- Analysewerkzeuge**
- Code editieren**
- 6. Testen und Testwerkzeuge**
  
- **Abschließende kurze Übung**
- **Zusammenfassende Fragen**
- **Ausblick**

## Stufenplan: 5. Beheben durch Analysieren

### Tools für node-js/express

#### Inspektion

node-inspector  
IDE Debugger

#### Logging

debug  
morgan  
log4js

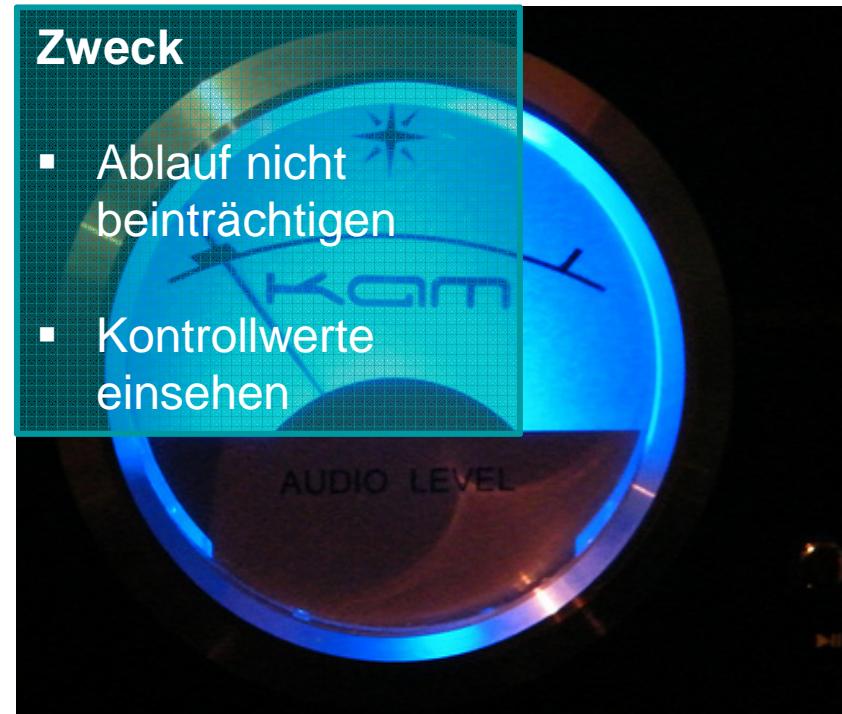


#### Zweck

- Ablauf  
nachvollziehen
- Systemzustand  
erfassen

#### Zweck

- Ablauf nicht  
beinträchtigen
- Kontrollwerte  
einsehen



## Stufenplan: 5. Beheben durch Analysieren

- **yarn add debug**

```
// node module imports
var express = require('express');
var debug = require('debug');

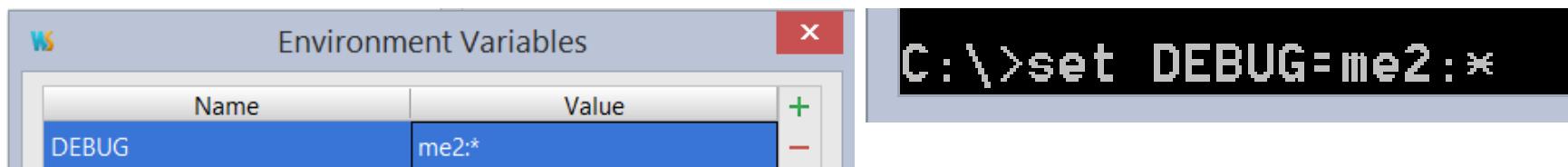
...
// creating the logger
var logger = debug('me2:su6app');

logger("Logger initialized");
```

### Logging

debug  
morgan  
log4js

- **debug filtert Ausgaben anhand der Umgebungsvariable DEBUG.  
(fehlt die, kommt keine Ausgabe)**





## Stufenplan: 5. Beheben durch Analysieren

### ■ **yarn add debug**

```
// node module imports
var express = require('express');
var debug = require('debug');

...
// creating the logger
var logger = debug('me2:su6app');

logger("Logger initialized");
```

### Logging

debug  
morgan  
log4js

### ■ **Express selbst nutzt debug!**

- **Setzen Sie z.B. `set DEBUG=me2:*`, `express:*`**

```
...
Tue, 09 May 2017 08:37:20 GMT express:router:layer new /
Tue, 09 May 2017 08:37:20 GMT express:router:route new /tweets
Tue, 09 May 2017 08:37:20 GMT express:router:layer new /tweets
Tue, 09 May 2017 08:37:20 GMT express:router:route get /tweets
Tue, 09 May 2017 08:37:20 GMT express:router:layer new /
```



## Stufenplan: 5. Beheben durch Analysieren

- **yarn add debug**

```
// node module imports
var express = require('express');
var debug = require('debug');

...
// creating the logger
var logger = debug('me2:su6app');

logger("Logger initialized");
```

### Logging

debug  
morgan  
log4js

```
console.log("Das bitte nicht mehr!");
```



## Stufenplan: 5. Beheben durch Analysieren

- **yarn add morgan**
  - ein HTTP Request/Response Logger
  - als Middleware

```
// node module imports
...
var morgan = require('morgan');
...
// Middleware *****
app.use(morgan('common'));
```

### Logging

debug  
morgan  
log4js

### Konsolenausgabe

```
::1 - - [09/May/2017:09:04:04 +0000] "GET /tweets HTTP/1.1" 200 190
::1 - - [09/May/2017:09:04:28 +0000] "GET /tweets/101 HTTP/1.1" 404 23
```

## Stufenplan: 5. Beheben durch Analysieren

- **yarn add morgan**
  - ein **HTTP Request/Response Logger**
  - als **Middleware**

```
// node module imports  
...  
var morgan = require('morgan');  
...  
// Middleware *****  
app.use(morgan('tiny'));
```

### Logging

debug  
morgan  
log4js

### Ausgabe-Format

common  
tiny  
combined  
dev  
short

### Konsolenausgabe

```
GET /tweets 200 190 - 3.738 ms  
GET /tweets/101 404 23 - 2.546 ms
```



## Stufenplan: 5. Beheben durch Analysieren

### ■ debug als Ausgabe von morgan

```
// node module imports
...
var debug = require('debug');
var morgan = require('morgan');

// creating the logger
var logger = debug('me2:su5app');

// Middleware *****
app.use(morgan('tiny', {
  "stream": { write: function(str) { logger(str); } }
}));
```

### Logging

debug  
morgan  
log4js

### Konsolenausgabe

```
Tue, 09 May 2017 09:12:23 GMT me2:su4app GET /tweets 200 190 - 3.341 ms
Tue, 09 May 2017 09:12:26 GMT me2:su4app GET /tweets/101 404 23 - 0.724 ms
```



## Stufenplan: 5. Beheben durch Analysieren

- **yarn add log4js**
  - Ein umfangreiches Logging Framework
  - Basiert auf browser-seitigem log4js
  - Portiert für node

### Logging

debug  
morgan  
log4js

```
var log4js = require('log4js');

...
// creating the logger
var logger = log4js.getLogger('we2:su6app');
logger.setLevel('TRACE');

...
logger.trace("Details about the sys state");
logger.debug("Logger initialized");
logger.info("System startup");
logger.warn("Hmm, should not be");
logger.error("An exceptional state");
logger.fatal("Ups!");
```



## Stufenplan: 5. Beheben durch Analysieren

- **yarn add log4js**
  - Ein umfangreiches Logging Framework
  - Basiert auf browser-seitigem log4js
  - Portiert für node

...

```
logger.trace("Details about the sys state");
logger.debug("Logger initialized");
logger.info("System startup");
logger.warn("Hmm, should not be");
logger.error("An exceptional state");
logger.fatal("Ups!");
```

### Logging

debug  
morgan  
log4js

## Konsolenausgabe

```
[2017-05-10 13:32:17.935] [TRACE] we2:su6app - Details about the sys state
[2017-05-10 13:32:17.938] [DEBUG] we2:su6app - Logger initialized
[2017-05-10 13:32:17.938] [INFO] we2:su6app - System startup
[2017-05-10 13:32:17.938] [WARN] we2:su6app - Hmm, should not be
[2017-05-10 13:32:17.938] [ERROR] we2:su6app - An exceptional state
[2017-05-10 13:32:17.938] [FATAL] we2:su6app - Ups!
```



## Stufenplan: 5. Beheben durch Analysieren

- **yarn global add node-inspector**
  - Debugger der im Chrome-Browser / Opera läuft
  - nutzt auf node-debug
  - Server-seitige Breakpoints
  - Achtung: Derzeit nur mit node v6.x kompatibel (nicht 7.x)
  - Code-Editierung im Browser → auf Server geändert
  
- Start des Servers dann mit  
**node-debug --save-live-edit app.js**
  - Ruft automatisch node mit --debug-brk auf (Anhalten nach Start)
  - Startet automatisch Chrome Browser mit node-inspector auf port 8080

### Inspektion

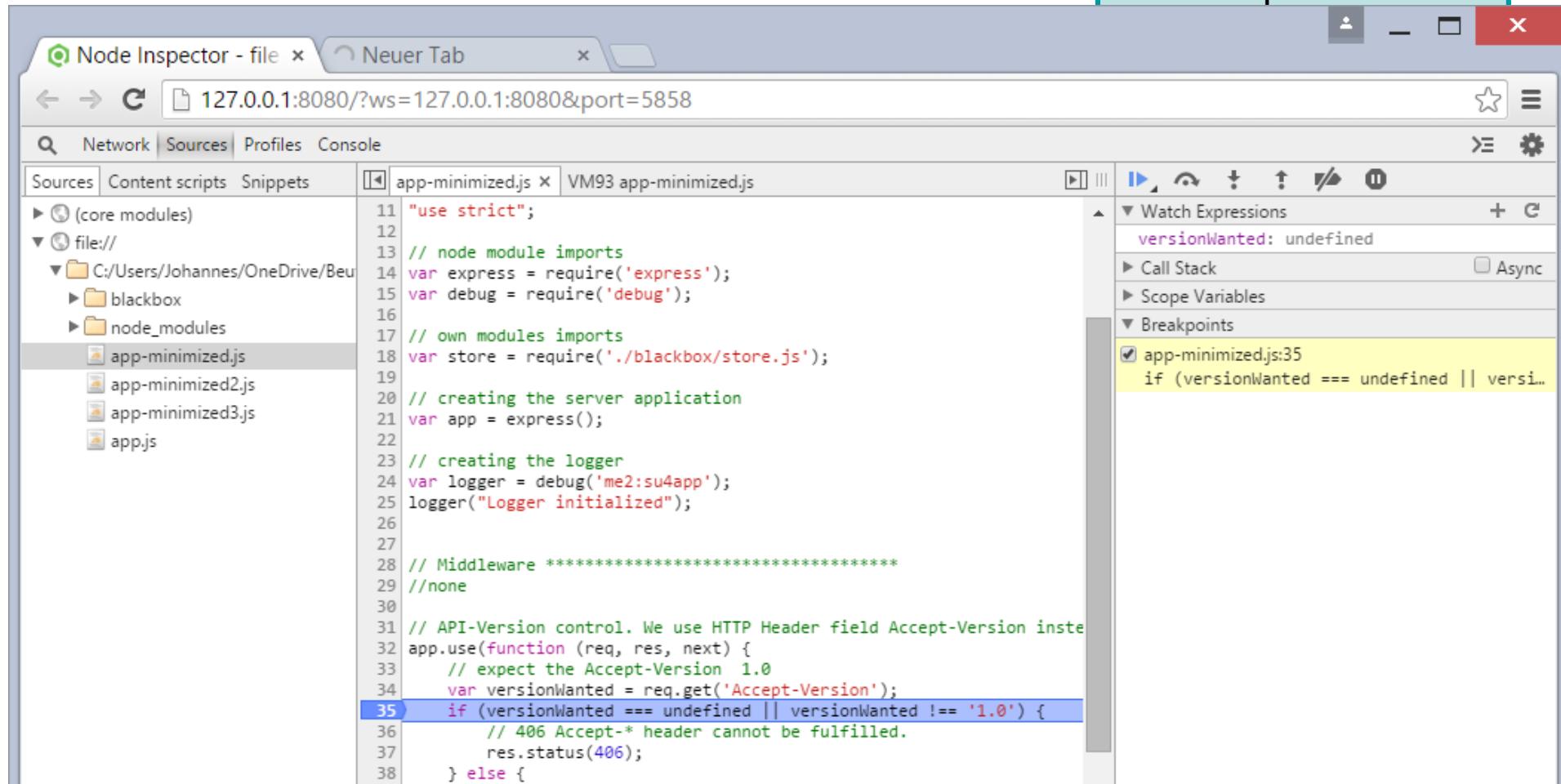
node-inspector  
IDE Debugger

## Stufenplan: 5. Beheben durch Analysieren

- **yarn global add node-inspector**

### Inspektion

node-inspector



The screenshot shows the Node Inspector interface with the following details:

- Network Tab:** Shows the URL `127.0.0.1:8080/?ws=127.0.0.1:8080&port=5858`.
- Sources Tab:** Displays the file `app-minimized.js`. The code is as follows:

```
11 "use strict";
12
13 // node module imports
14 var express = require('express');
15 var debug = require('debug');
16
17 // own modules imports
18 var store = require('./blackbox/store.js');
19
20 // creating the server application
21 var app = express();
22
23 // creating the logger
24 var logger = debug('me2:su4app');
25 logger("Logger initialized");
26
27
28 // Middleware ****
29 //none
30
31 // API-Version control. We use HTTP Header field Accept-Version instead
32 app.use(function (req, res, next) {
33     // expect the Accept-Version 1.0
34     var versionWanted = req.get('Accept-Version');
35     if (versionWanted === undefined || versionWanted !== '1.0') {
36         // 406 Accept-* header cannot be fulfilled.
37         res.status(406);
38     } else {
```

- Breakpoints Panel:** Shows a breakpoint at line 35 of `app-minimized.js`, which is highlighted in yellow. The condition is `if (versionWanted === undefined || versionWanted !== '1.0')`.

## Stufenplan: 5. Beheben durch Analysieren

### ■ WebStorm Debugger

- IDEs haben oft integrierte Debugger
- Starten node und Klicken eigenen Debugger ein
- Achtung: Derzeit nur mit node v6.x kompatibel (nicht 7.x)

### Inspektion

node-inspector  
IDE Debugger

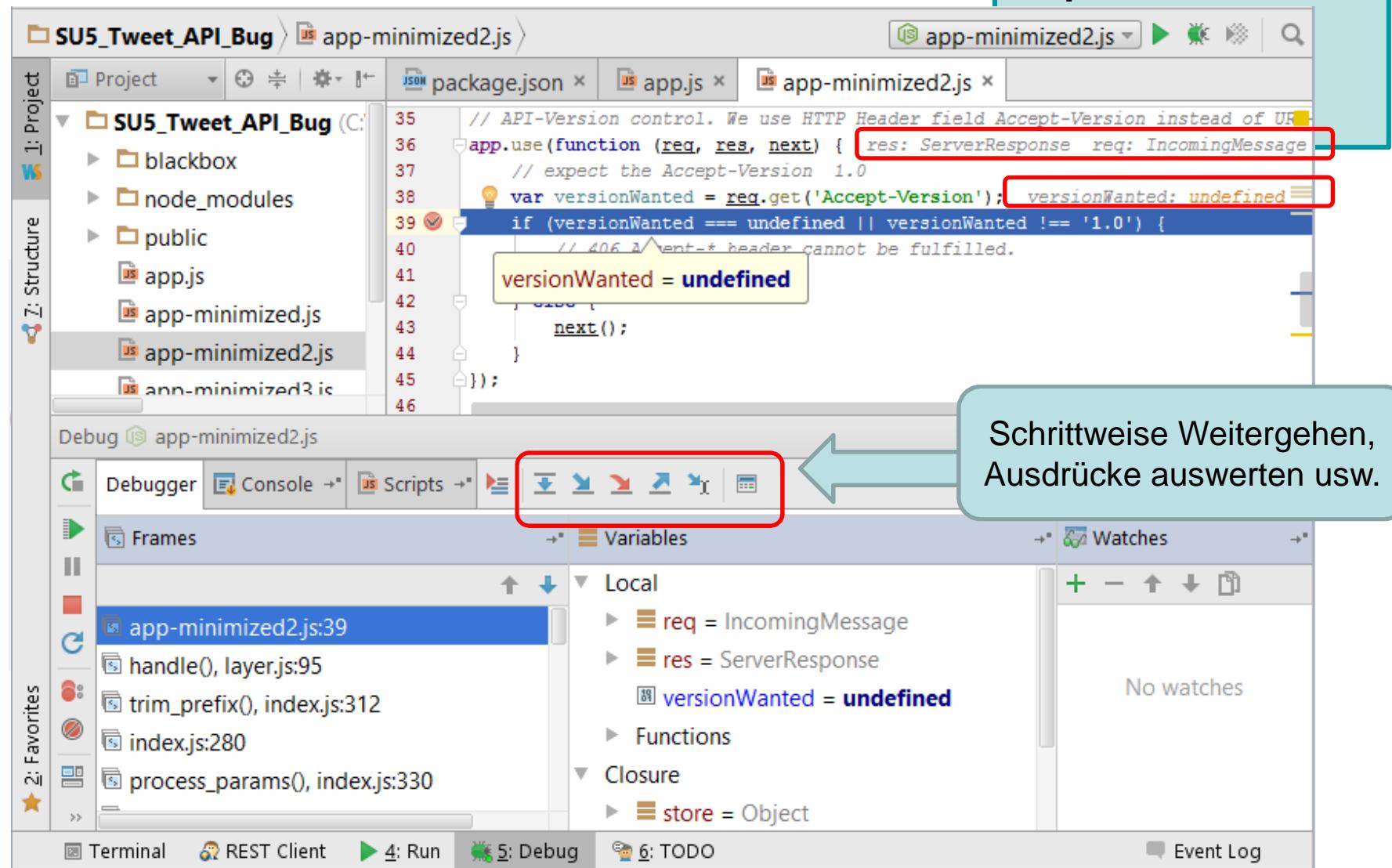


The screenshot shows the WebStorm IDE with the code editor displaying a file named 'app-minimized2.js'. The code handles API version control based on the 'Accept-Version' header. A red dot on the left margin indicates a breakpoint at the start of the conditional block. The code editor has syntax highlighting and code completion. To the right of the editor is a toolbar with various icons for debugging, including a green play button for running, a magnifying glass for search, and other debugger-related functions. A tooltip above the toolbar reads 'Debug 'app-minimized2.js' (Umschalt+F9)'. The status bar at the bottom shows the file name 'app-minimized2.js'.

```
// API-Version control. We use HTTP Header fi
app.use(function (req, res, next) {
    // expect the Accept-Version 1.0
    var versionWanted = req.get('Accept-Version');
    if (versionWanted === undefined || versionWanted !== '1.0') {
        // 406 Accept-* header cannot be fulfilled.
        res.status(406);
    } else {
        next();
    }
});
```

## Stufenplan: 5. Beheben durch Analysieren

### Inspektion



The screenshot shows an IDE interface with the following details:

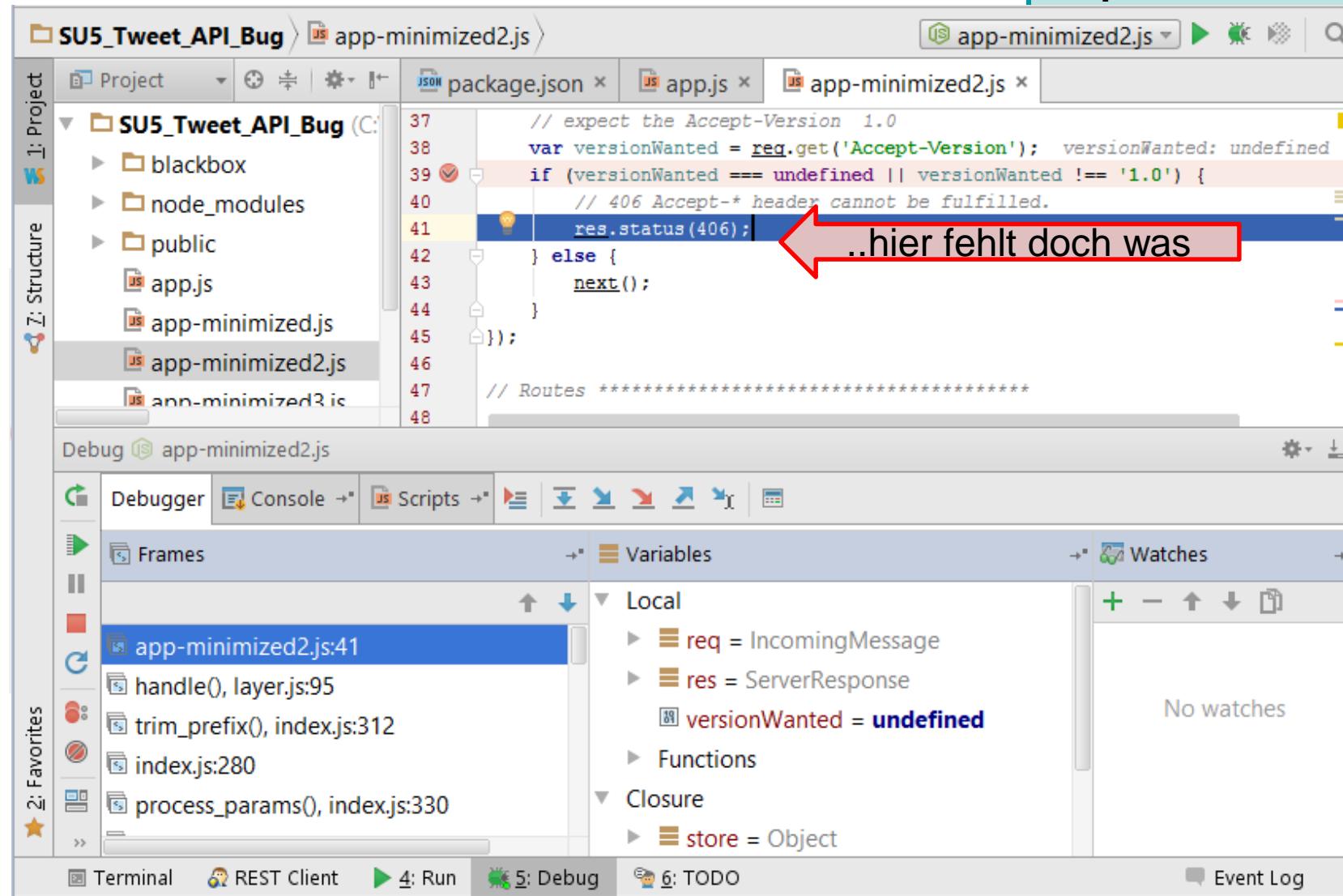
- Project Structure:** The project is named "SU5\_Tweet\_API\_Bug". It contains folders like "blackbox", "node\_modules", and "public", and files like "app.js", "app-minimized.js", "app-minimized2.js", and "app-minimized3.js".
- Code Editor:** The file "app-minimized2.js" is open. Line 39 contains the code: 

```
if (versionWanted === undefined || versionWanted !== '1.0') {
```

 A tooltip indicates that "versionWanted = undefined".
- Toolbars:** The top toolbar includes tabs for "Project", "package.json", "app.js", and "app-minimized2.js". The bottom toolbar includes buttons for "Terminal", "REST Client", "Run", "Debug", "TODO", and "Event Log".
- Frames:** The stack trace shows the current frame is at "app-minimized2.js:39". Other frames include "handle()", "layer.js:95", "trim\_prefix()", "index.js:312", "index.js:280", and "process\_params()", "index.js:330".
- Variables:** The "Variables" panel shows the "Local" scope with variables: "req = IncomingMessage", "res = ServerResponse", and "versionWanted = undefined".
- Watches:** The "Watches" panel shows "No watches".
- Callout Bubble:** A large callout bubble with a blue arrow points to the "Step Into" button in the debugger toolbar. The text inside the bubble reads: "Schrittweise Weitergehen, Ausdrücke auswerten usw." (Stepwise Continue, Evaluate Expressions etc.).

## Stufenplan: 5. Beheben durch Analysieren

### Inspektion



The screenshot shows an IDE interface with the following details:

- Project Structure:** The project is named "SU5\_Tweet\_API\_Bug". It contains folders "blackbox", "node\_modules", and "public", and files "app.js", "app-minimized.js", "app-minimized2.js", and "app-minimized3.js".
- Code Editor:** The file "app-minimized2.js" is open. The code is as follows:

```
// expect the Accept-Version 1.0
var versionWanted = req.get('Accept-Version'); versionWanted: undefined
if (versionWanted === undefined || versionWanted !== '1.0') {
    // 406 Accept-* header cannot be fulfilled.
    res.status(406);
} else {
    next();
}
// Routes ****
```

- Annotations:** Line 39 has a red circle with a checkmark. Line 41 has a yellow lightbulb icon.
- Variables View:** The "Variables" tab is selected. It shows the following local variables:
  - req = IncomingMessage
  - res = ServerResponse
  - versionWanted = undefined
  - Functions
  - Closure
  - store = Object
- Frames View:** The stack trace shows the current frame is at "app-minimized2.js:41". Other frames include "handle()", "layer.js:95", "trim\_prefix()", "index.js:312", "index.js:280", and "process\_params()", "index.js:330".
- Watches View:** No watches are present.
- Bottom Navigation:** The tabs include Terminal, REST Client, Run, Debug (selected), TODO, and Event Log.



## Agenda

- **Wiederholung und Bonusfrage**

- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**

- **Debug-Vorgehen: 6 Stufenplan**

3. Minimierung
4. Hypothesen
5. Analysewerkzeuge

Code editieren

6. Testen und Testwerkzeuge

- **Abschließende kurze Übung**

- **Zusammenfassende Fragen**

- **Ausblick**



## Stufenplan: 5. Beheben durch Analysieren

### ■ Code anpassen

```
// API-Version control. We use HTTP Header field Accept-Version
app.use(function (req, res, next) {
    // expect the Accept-Version 1.0
    var versionWanted = req.get('Accept-Version');
    if (versionWanted === undefined || versionWanted !== '1.0')
        // 406 Accept-* header cannot be fulfilled.
        res.status(406).end();
    } else {
        next();
    }
});
```



## Stufenplan: 5. Beheben durch Analysieren

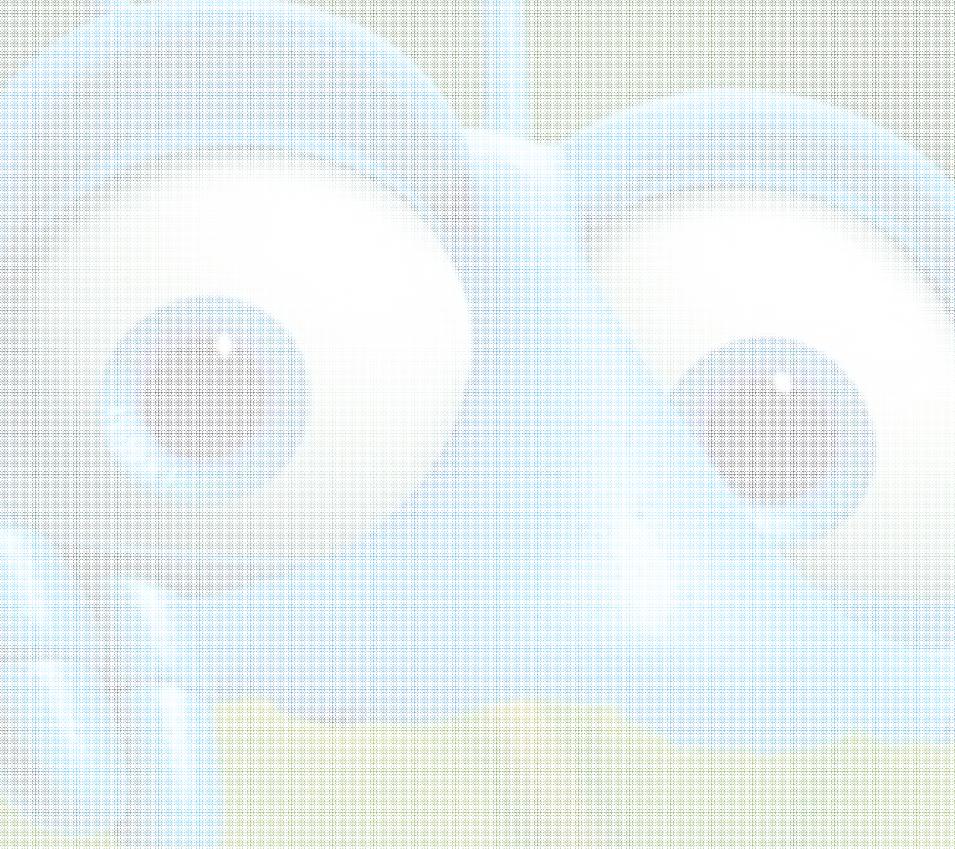
- Erneuter Aufruf
- Postman erhält nun eine Fehlermeldung

The screenshot shows the Postman interface with the following details:

- Request URL:** http://localhost:300...
- Method:** GET
- Endpoint:** http://localhost:3000/tweets
- Headers (1):**
  - Accept:** application/json
- Status:** 406 Not Acceptable
- Time:** 13 ms
- Body:** Connection → keep-alive  
Content-Length → 0



# Bug erfolgreich behoben



## Von der Hypothese zum Nachweis: Den Bug finden

### Ein 6 Stufenplan:

- 1. Bug beschreiben** (u.a. SOLL und IST)  

- 2. Stabile Reproduzierbarkeit herstellen**
- 3. Isolieren** und „Minimized Example“ erstellen
- 4. Hypothesen** formulieren zur Frage:  
„Wo könnte der Bug herkommen?“
  - a. Das wahrscheinlichste zuerst prüfen
  - b. Die einfache These bevorzugen vor der komplizierten
  - c. Mit jeder geprüften These 50% der Fehlerquellen ausschließen\*\*
- 5. Beheben durch Analysieren**  
Nicht geklappt? **4.** wiederholen.  
Wenn nichts mehr übrig,  
**3.** wiederholen, **4.** detaillierter (Annahmen hinterfragen!)
- 6. Doku und Tests:** **2.** und **3.** rückgängig machen und testen (ideal: Unit-Test)

\*\* Konzept des Divide und Conquer (Teile und (Be)herrsche)



## Agenda

- **Wiederholung und Bonusfrage**
- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**
- **Debug-Vorgehen: 6 Stufenplan**
  - 3. Minimierung
  - 4. Hypothesen
  - 5. Analysewerkzeuge
  - Code editieren
- Testen und Testwerkzeuge**
- **Abschließende kurze Übung**
- **Zusammenfassende Fragen**
- **Ausblick**



## Stufenplan: 6. Testwerkzeuge

### Testen

jasmine + frisby  
Mocha + should-http



## Stufenplan: 6. Testwerkzeuge

### Achtung: Rollenwechsel

Von Detektiv/in

Zu Vandalen/in („think evil“)

### Testen

jasmine + frisby  
Mocha + should-http

### Was testen

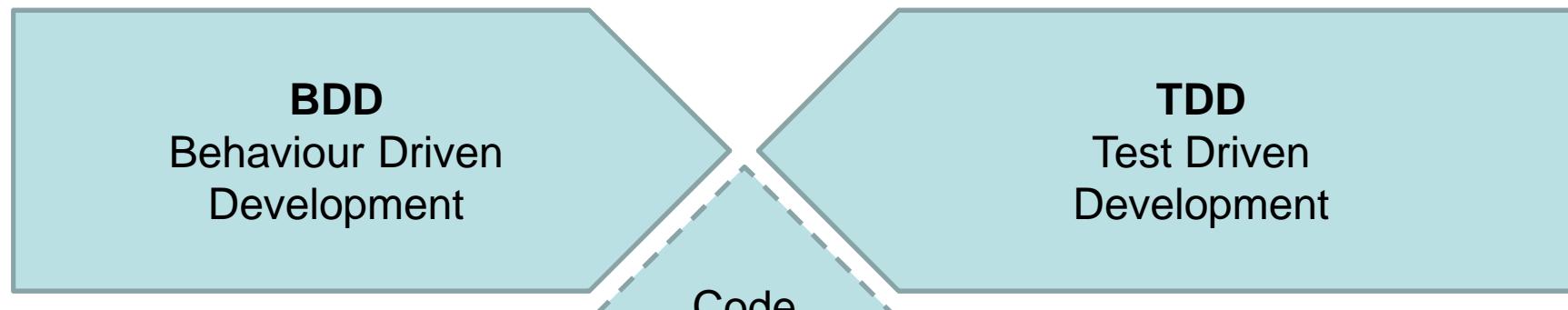
- **Normale Anfragen**
- **Extremfälle**
- **Ungültige Fälle**

### Warum testen

- **Code-Coverage**  
Abdecken von idealerweise 100% des Codes durch Tests
- **Definition of Done (DoD)**  
Spezifikation der Tests als DoD vor Implementierung. Legt fest, wann ein Task fertig implementiert ist
- **Continuous Integration**  
Weiterentwickelter Code kommt erst auf den Staging Server, wenn alle Tests durchlaufen.

## Stufenplan: 6. Testwerkzeuge

### ■ Zunächst: 2 Ansätze



- Fasst zusammen mit `describe(...)`
- Beschreibt erwartetes Verhalten mit `it(...)` und `should(...)` oder `expect(...)`.
- Fasst zusammen mit `suite(...)`
- Beschreibt erwartetes Verhalten mit `test(...)` und `assert(...)`.



## Stufenplan: 6. Testwerkzeuge

- **Frisby für REST-APIs**
- **Basierend auf allg. Test-Suite Jasmine**
  - **yarn add jasmine-node --dev**

**BDD Testen**  
jasmine + frisby  
Mocha + should-http

### Datei: mini\_test\_spec.js

```
// Tests
describe('Tweet REST API system', function() {
    it('delivers two tweets with id 101 and 102', function() {
        // do some GET
        //...
        expect(contentType).toContain('application/json');
        // and so on
    });
});
```

- **Starten mit jasmine-node**  
**(sucht alle Dateien mit \*\_spec.js, üblicher Order ist \spec)**
  - **Achtung: node.js Server muss unabhängig vorab gestartet sein!**



## Stufenplan: 6. Testwerkzeuge

- + **frisby**
  - **yarn add frisby --dev**

**BDD Testen**  
jasmine + frisby  
Mocha + should-http

### Datei: mini\_test\_spec.js

```
var frisby = require('frisby');

// Tests
describe('Tweet REST API system', function() {
  frisby.create('will send status 200 on GET')
    .get('http://localhost:3000/tweets')
    .expectStatus(200)
    .toss();
  ...
});
```

- Weiterhin Start mit **jasmine-node**



## Stufenplan: 6. Testwerkzeuge

### ■ Konsolenausgabe

```
Finished in 0.045 seconds
1 test, 1 assertion, 0 failures, 0 skipped
```

**BDD Testen**  
jasmine + frisby  
Mocha + should-http

### ■ Dokumentation

- **Jasmine expect(..) usw.**  
<http://jasmine.github.io/2.3/introduction.html#section-Expectations>
- **Frisby REST-Methoden**  
<http://frisbyjs.com/docs/api/>

## Stufenplan: 6. Testwerkzeuge

### ■ Jasmine mit Frisby: Beispiel

```
describe('Tweet REST API', function() {
    frisby.create('will send status 200 on GET')
        .get('http://localhost:3000/tweets')
        .expectStatus(200)
        .toss();

    frisby.create('will send JSON data of two tweets 101,102')
        .get('http://localhost:3000/tweets')
        .expectHeaderContains('Content-Type', 'application/json')
        .expectJSON('?',
            {
                'id': 102
            })
        .afterJSON(function(jsonObj) {
            expect(jsonObj[0].id).toMatch(101);
            expect(jsonObj.length).toBe(2);
        })
        .toss();
});
```

**BDD Testen**  
jasmine + frisby  
Mocha + should-http



## Stufenplan: 6. Testwerkzeuge

- **Jasmine in node „integrieren“**

### 1. package.json anpassen

```
"scripts": {  
  "start": "node app.js",  
  "test": "jasmine-node ./spec/"}
```

### 2. Anschließend im Projektverzeichnis starten mit npm test

```
> me2-su5_tweet_bug@0.0.1 test C:\Users\Johannes\OneDrive\Beuth_Lehre\ME2\Code\S  
U5_Tweet_API_Bug  
> jasmine-node ./spec/  
  
.  
  
Finished in 0.061 seconds  
2 tests, 5 assertions, 0 failures, 0 skipped
```

**BDD Testen**  
jasmine + frisby  
Mocha + should-http



## Stufenplan: 6. Testwerkzeuge

**BDD Testen**  
jasmine + frisby  
Mocha + should-http





## Stufenplan: 6. Testwerkzeuge

### ■ Mocha

**yarn add mocha -dev**

- Testsuit für Client oder Server
  - Kann mit CLI Option **--ui tdd** auf „Test Driven“ umgestellt werden

**BDD Testen**  
jasmine + frisby  
Mocha + should-http

### 1. package.json anpassen

```
"scripts": {  
  "test": "mocha"
```

### 2. Anschließend im Projektverzeichnis wieder starten mit

**npm test**

- Mocha sucht im Unterordner .\test nach .js Dateien



## Stufenplan: 6. Testwerkzeuge

### ■ Mocha-Code (Zusatzmodule)

- **should / should-http:** Erweitern Object.prototype
- **supertest / superagent:** HTTP-Requests durchführen und Antworten prüfen.

**BDD Testen**  
jasmine + frisby  
Mocha + should-http

```
var should = require('should');
require('should-http');
var request = require('supertest');

describe('miniTwitter REST API', function() {

    it('should send status 200 on GET', function(done) {
        ...
        done();
    });
});

});
```

...



## Stufenplan: 6. Testwerkzeuge

### ■ Mocha-Code (mit `should.js`, `supertest.js`)

```
describe('miniTwitter REST API', function() {  
    var url = 'http://localhost:3000';
```

```
    it('will send 2 tweets 101,102 as JSON', function(done) {  
        request(url)  
            .get('/tweets')  
            .set('Accept-Version', '1.0')  
            .set('Accept', 'application/json')  
            .expect('Content-Type', /json/)  
            .expect(200)  
            .end(function(err, res) {  
                should.not.exist(err);  
                res.should.be.json();  
                res.body.should.have.lengthOf(2);  
                res.body[0].should.have.keys(['id', 'message', 'creator']);  
                done();  
            })  
    })  
});
```



## Stufenplan: 6. Testwerkzeuge

- Mocha-Code (mit `should.js`, `supertest.js`)

BDD Testen  
Mocha + `should-http`

```
C:\Users\Johannes\OneDrive\Beuth_Lehre\ME2\Code\SU5_Tweet_API_Bug>npm test

> me2-su5_tweet_bug@0.0.1 test C:\Users\Johannes\OneDrive\Beuth_Lehre\ME2\Code\SU5_Tweet_API_Bug
> mocha

miniTwitter REST API
  U should sends status 200 on GET
  U will send 2 tweets 101,102 with content type json

2 passing (56ms)
```

- Dokumentationen:
  - <https://github.com/shouldjs/should.js>
  - <https://www.npmjs.com/package/should-http>
  - <https://github.com/visionmedia/supertest>
  - <https://mochajs.org/#getting-started>
  - Vergleich Jasmine 2.0 vs. Mocha  
<http://thejsguy.com/2015/01/12/jasmine-vs-mocha-chai-and-sinon.html>

## Von der Hypothese zum Nachweis: Den Bug finden

### Ein 6 Stufenplan:

- 1. Bug beschreiben** (u.a. SOLL und IST)  

- 2. Stabile Reproduzierbarkeit herstellen**  

- 3. Isolieren** und „Minimized Example“ erstellen  

- 4. Hypothesen** formulieren zur Frage:  
„Wo könnte der Bug herkommen?“
  - a. Das wahrscheinlichste zuerst prüfen
  - b. Die einfache These bevorzugen vor der komplizierten
  - c. Mit jeder geprüften These 50% der Fehlerquellen ausschließen\*\*  

- 5. Beheben durch Analysieren**  
Nicht geklappt? **4.** wiederholen.  
Wenn nichts mehr übrig,  
    **3.** wiederholen, **4.** detaillierter (Annahmen hinterfragen!)  

- 6. Doku und Tests:** **2.** und **3.** rückgängig machen und testen (ideal: Unit-Test)  


\*\* Konzept des Divide und Conquer (Teile und (Be)herrsche)



## Agenda

- **Wiederholung und Bonusfrage**
  
- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**
  
- **Debug-Vorgehen: 6 Stufenplan**
  - 3. Minimierung
  - 4. Hypothesen
  - 5.
  - Code editieren
  - 6. Testen und Testwerkzeuge
  
- **Abschließende kurze Übung**
- **Zusammenfassende Fragen**
- **Ausblick**

## Ziele von Testing, Inspektion, Logging

(Unit)Tests

Debug-  
Inspektion

Logging

- Aufgabe: Ordnen Sie die Ziele zu den drei vorgestellten Werkzeugen zu. **Einzelarbeit** (2-3min)

Ablauf-Verfolgung

Code-Abdeckung

Änderung innerer  
Systemzustände

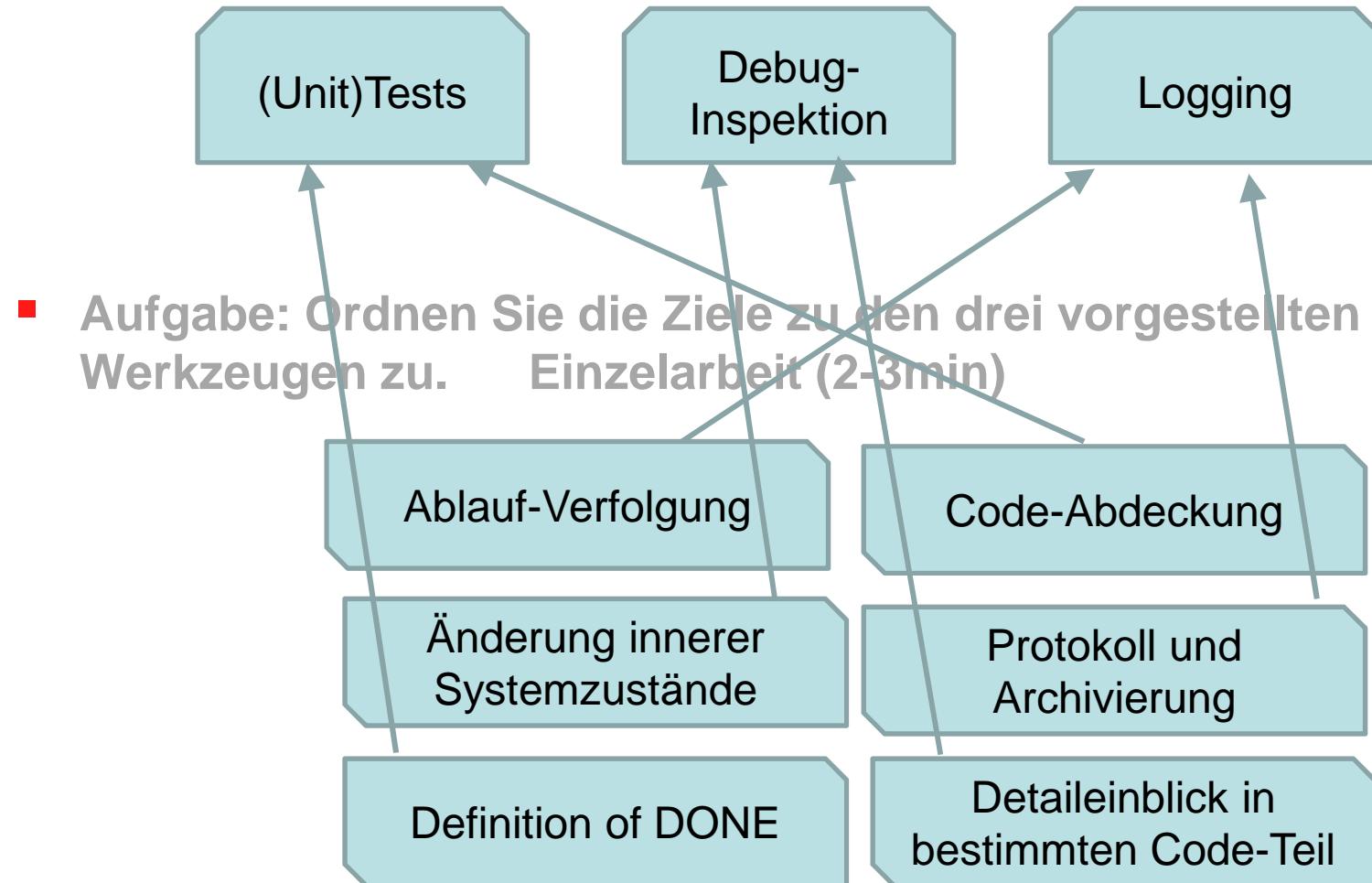
Protokoll und  
Archivierung

Definition of DONE

Detaileinblick in  
bestimmten Code-Teil



## Ziele von Testing, Inspektion, Logging



- Aufgabe: Ordnen Sie die Ziele zu den drei vorgestellten Werkzeugen zu. Einzelarbeit (2-3min)



## Ziele von Testing und Debugging

- **(Unit)Tests**
  - **Code-Abdeckung**
  - **Nutzbar für Definition of Done**
  - **Standardisierte, reproduzierbare Prüfung der Funktionalität**
  - **Nutzbar für agile Entwicklung und Test-driven Development**
  - **Achtung: Trügerische Schein-Sicherheit**
- **Debug-Inspektion**
  - **Detaileinblick in bestimmten Code-Teil**
  - **Einsicht und Änderung innerer Systemzustände**
- **Logging**
  - **Ablauf-Verfolgung**
  - **Protokoll und Archivierung**
  - **(Einsicht in Teile v. Systemzustand / Variablen)**

**Ziel für alle 3:  
Erkennen/Eingrenzen  
von Bugs!  
→ Qualität verbessern**



## Agenda

- **Wiederholung und Bonusfrage**
- **Was ist ein Bug?**
- **Rollen bei der Softwareentwicklung**
- **Beschreiben eines Bugs**
- **Debug-Vorgehen: 6 Stufenplan**
  - 3. Minimierung
  - 4. Hypothesen
  - 5.
  - Code editieren
  - 6. Testen und Testwerkzeuge
- **Abschließende kurze Übung**
- **Zusammenfassende Fragen**
- **Ausblick**

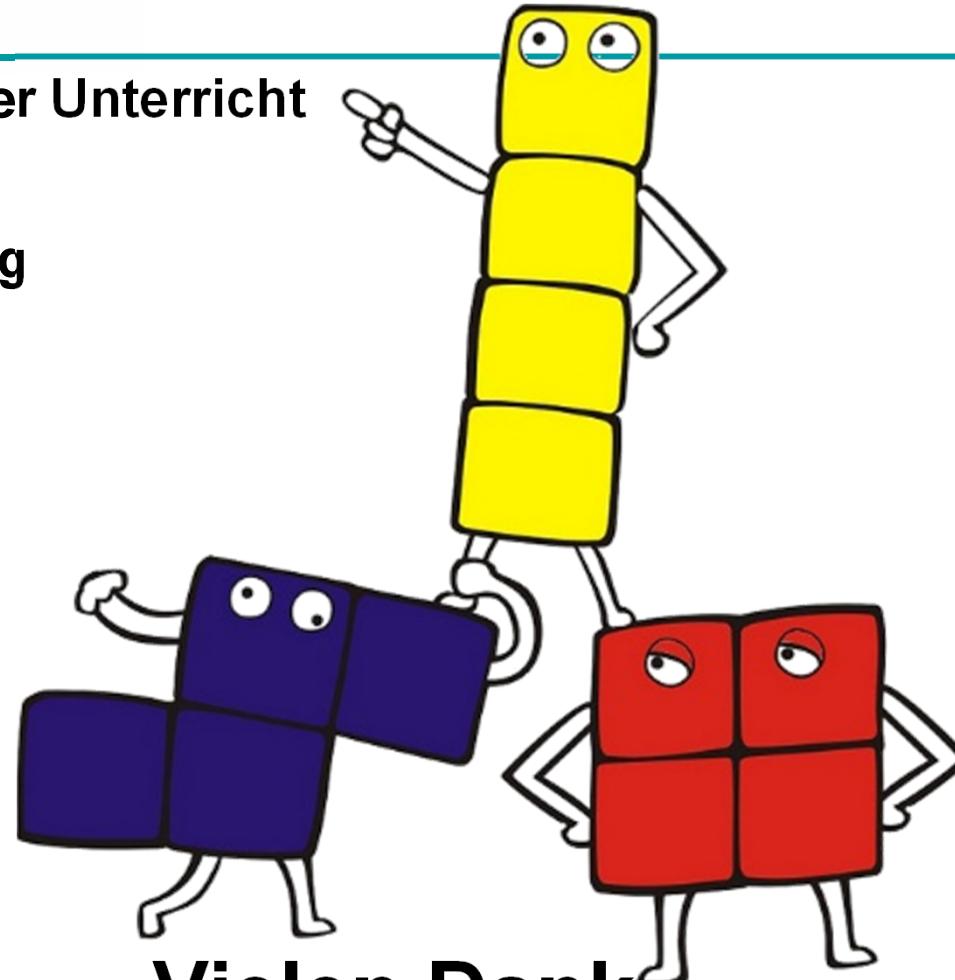


## Zusammenfassende Fragen

1. Wie können Sie mit express.js Werte von einer Handler-Funktion an die nächste weitergeben (obwohl next() ohne Parameter genutzt wird)?
2. Wie lautet die Definition des Begriffes Bug?
3. Welche 4 Rollen nehmen Sie als Softwareentwickler/in ein? Für welche Aufgaben?
4. Welche 5 Teile gehören zu einer Bugbeschreibung?
5. Was ist mit Schritt 2 Stabile Reproduzierbarkeit herstellen gemeint? Wie geht das?
6. Wieso ist es wichtig ein Minimized Example zu erstellen?
7. Welche drei Aspekte beachten Sie beim formulieren von Hypothesen, um zielgerichtet den Bug zu untersuchen (Detektiv)?
8. Für welche Ziele verwenden Sie Inspektion (Debugger) und wann besser Logging?
9. Wofür stehen die Logger-Level TRACE und FATAL? Wozu gibt es überhaupt verschiedene Level?
10. Welche drei Gruppen von Fällen sollten Sie stets testen (manuell oder mit UnitTests)?
11. Welche Ziele werden mit dem Unit-Testing verfolgt?
12. Wie funktioniert eine BDD oder TDD-basierte Test-Beschreibung?
13. Wozu dient das Modul should? Was wird damit vereinfacht in UnitTests?
14. Im Unterschied zu jasmine/frisby ist bei mocha die Testausführung asynchron. Auf welchen Code-Aufruf müssen Sie achten, um mocha mitzuteilen, dass ein Test fertig ist?
15. Was sagt eine Code-Coverage von 100% über die Qualität des Programmcodes aus?
16. Bonus-Frage: Wäre es sinnvoll, dass eine Programmiererin, die nicht Teil des Entwicklungsteams war, die Tests schreibt? Wenn ja, wie sollte Sie das tun? Wenn nein, warum nicht?

## Ausblick / Nächster Unterricht

- **Strukturierung,  
Modularisierung**
  - requireJS
  - AMD



**Vielen Dank  
und bis  
zum nächsten Mal**