UNIKASSEL
VERSITÄT

SOFTWARE ENGINEERING
UNIVERSITÄT KASSEL

Software Tool Construction

Wintersemester 2023/24

Adrian Kunz

Vorlesung 12

Deployment mit GitHub Pages

- Starter Workflow: https://github.com/actions/starter-workflows/blob/main/pages/static.yml
- Pnpm Setup: https://github.com/pnpm/action-setup#use-cache-to-reduce-installation-time

IntelliJ

- LSP Support Dokumentation: https://plugins.jetbrains.com/docs/intellij/language-server-protocol.html
- Beispielprojekt mit LSP: https://github.com/fujaba/fulibFeedback/tree/master/apps/intellij-plugin
- TextMate Bundles Plugin: https://plugins.jetbrains.com/plugin/7221-textmate-bundles (leider nicht erweiterbar durch eigenes Plugin)

Debugging

JavaScript: SourceMaps

```ts
// helloWorld.ts
function greet(what: string) {
  console.log(`Hello ${what}`);
}
const world: string = "World";
greet(world);
```

```js
// helloWorld.js
function greet(what) {
  console.log("Hello ".concat(what));
}
var world = "World";
greet(world);
//# sourceMappingURL=helloWorld.js.map
```

```json
// helloWorld.js.map
{
  "version": 3,
  "file": "helloWorld.js",
  "sourceRoot": "",
  "sources": ["helloWorld.ts"],
  "names": [],
  "mappings":
"AAAA,SAAS,KAAK,CAAC,IAAY;IACvB,OAAO,CAAC,GAA
G,CAAC,gBAAS,IAAI,CAAE,CAAC,CAAC;AACjC,CAAC;A
ACD,IAAM,KAAK,GAAW,OAAO,CAAC;AAC9B,KAAK,CAAC,
KAAK,CAAC,CAAC"
,
  "sourcesContent": [
    "function greet(what: string) {\n
console.log(`Hello ${what}`);\n}\nconst
world: string = \"World\";\ngreet(world);\n"
  ]
}
```

https://tc39.es/source-map-spec/
https://evanw.github.io/source-map-visualization/#...

JavaScript: Debugger

```
41        const listItem = document.createElement('li');
42        const listText = document.createElement('span');
43        const listBtn = document.createElement('button');
```

Spalteninformation erlauben spaltenweise Breakpoints

```
37      button.onclick = function() {
38        let myItem = input.value;   myItem: "cheese"
39        input.value = '';
40
41        const listItem = document.createElement('li');   listItem: li
42        const listText = document.createElement('span');   listText: span
43        const listBtn = document.createElement('button');   listBtn: button
```

Variablennamen erlauben bessere Einsicht

https://firefox-source-docs.mozilla.org/devtools-user/debugger/how_to/set_a_breakpoint/index.html

Java: LineNumberTable und LocalVariableTable

```java
// HelloWorld.java
public class HelloWorld {
  public static void main(String[] args) {
    String world = "World";
    greet(world);
  }

  public static void greet(String what) {
    System.out.println("Hello " + what);
  }
}
```

```
public class HelloWorld
{
  public static void main(java.lang.String[]);
    Code:
      stack=1, locals=2, args_size=1
         0: ldc #7 // String World
         2: astore_1
         3: aload_1
         4: invokestatic  #9 // greet
         7: return
      LineNumberTable:
        line 3: 0
        line 4: 3
        line 5: 7
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       8     0  args   [Ljava/lang/String;
            3       5     1 world   Ljava/lang/String;
```

Java: LineNumberTable und LocalVariableTable

```java
// HelloWorld.java
public class HelloWorld {
  public static void main(String[] args) {
    String world = "World";
    greet(world);
  }

  public static void greet(String what) {
    System.out.println("Hello " + what);
  }
}
```

```
public static void greet(java.lang.String);
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #15 // Field System.out
       3: aload_0
       4: invokedynamic #21, 0 // concat
       9: invokevirtual #25 // PrintStream.println
      12: return
    LineNumberTable:
      line 8: 0
      line 9: 12
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      13     0  what   Ljava/lang/String;
}
```

Dateien und Inkrementelle Änderungen

Dateien und Inkrementelle Änderungen

```
// A.dyv
class A {
  func foo(b: B) {
    print(b.c.i);
  }
}
```

```
// B.dyv
class B {
  var c: C
}
```

```
// C.dyv
class C {
  var i: int

  func foo() {
    print(this.i);
  }
}
```

```
// D.dyv
class D {
  // ...
}
```

Abhängigkeiten:

- A -> B, C
- B -> C
- C
- D

Änderung an C: Attribut gelöscht

```
// A.dyv
class A {
  func foo(b: B) {
    print(b.c.i);
  }
}
```

unknown field 'i'

```
// B.dyv
class B {
  var c: C
}
```

```
// C.dyv*
class C {
  // var i: int

  func foo() {
    print(this.i);
  }
}
```

unknown field 'i'

```
// D.dyv
class D {
  // ...
}
```

Abhängigkeiten:

- **A -> B, C**
- **B -> C**
- **C**
- D

Änderung an C: Implementierung verändert

```
// A.dyv
class A {
  func foo(b: B) {
    print(b.c.i);
  }
}
```

unknown field 'i'

```
// B.dyv
class B {
  var c: C
}
```

```
// C.dyv*
class C {
  // var i: int

  func foo() {
    print(new D);
  }
}
```

```
// D.dyv
class D {
  // ...
}
```

Abhängigkeiten:

- A -> B, C
- B -> C
- **C -> D**
- D

Änderung an C: Neues private Attribut

```
// A.dyv
class A {
  func foo(b: B) {
    print(b.c.i);
  }
}
```

unknown field 'i'

```
// B.dyv
class B {
  var c: C
}
```

```
// C.dyv*
class C {
  private var i: int

  func foo() {
    print(new D);
  }
}
```

```
// D.dyv
class D {
  // ...
}
```

Abhängigkeiten:

- A -> B, C
- B -> C
- **C -> D**
- D

Compiler 1: "Performant"

Änderung an C: Neues private Attribut

```
// A.dyv
class A {
  func foo(b: B) {
    print(b.c.i);
  }
}
```

cannot access private field 'i'
🪄 Make 'i' public

```
// B.dyv
class B {
  var c: C
}
```

```
// C.dyv*
class C {
  private var i: int

  func foo() {
    print(new D);
  }
}
```

```
// D.dyv
class D {
  // ...
}
```

Abhängigkeiten:

- **A -> B, C**
- **B -> C**
- **C -> D**
- D

Compiler 2: "Benutzerfreundlich"

Signatur

- Umfasst alle nach außen sichtbaren Eigenschaften einer Datei
  - z.B. Klassen, Attribute, Methoden, Parameter, Typen
  - Genauer: Alles, was Einfluss auf die Kompilierung von abhängigen Dateien hat
- Beispiel
  A: "A(foo(B):void)"
  B: "B(c:C)"
  C: "C(i:int,foo():void)"

⚠ Problem:
- Signatur kann sehr lang werden

Signatur

- Umfasst alle nach außen sichtbaren Eigenschaften einer Datei
  - z.B. Klassen, Attribute, Methoden, Parameter, Typen
  - Genauer: Alles, was Einfluss auf die Kompilierung von abhängigen Dateien hat
- Beispiel

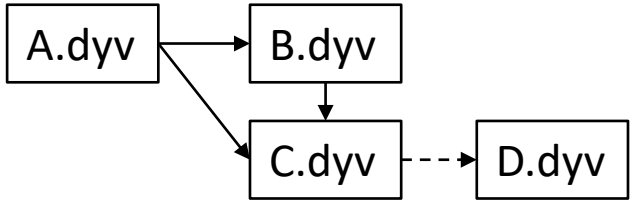
  A: "A(foo(B):void)".hashCode() => 0xfedd03ab
  B: "B(c:C)".hashCode() => 0x730167e3
  C: "C(i:int,foo():void)".hashCode() => 0xafabaf2f

✅
- Signatur hat begrenzte Länge

Besonderheiten

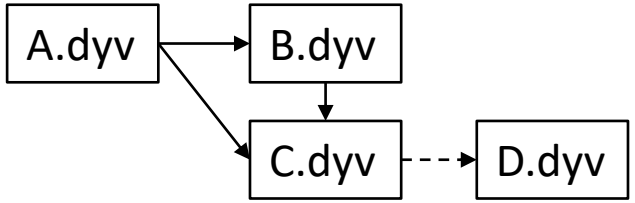
- Reihenfolge
  - `class A { var i: int; var j: int }`
    wird zu
    `class A { var j: int; var i: int }`
  - Signatur soll sich nicht verändern – Reihenfolge ist nach außen egal
  - Normalisierung/Sortieren notwendig: A(i:int,j:int)
- Hashcode
  - Naives hashCode() nicht perfekt (32 bit, Kollisionen…)
  - Java String.hashCode() gibt es nicht in JavaScript
  - Besser: SHA-(1|256|3) oder ähnliches

Implementierung: Daten



- Metadaten einer Datei
  - `Uri[] dependencies` – Abhängigkeiten der Datei inkl. nicht Signaturrelevante (⇢) (z.B. A.dependencies=[B, C], B.dependencies=[C], C.dependencies=[D])
  - `Signature signature` – Signatur der Datei (z.B. A.signature=0xfedd03ab, B.signature=0x730167e3)
- `Signature` – z.B. ein String oder (ausreichend kollisionsresistenter) Hash

Implementierung: Verhalten



- Bei einer Dateiänderung:
  - Parse AST neu
  - Aktualisiere Datei
- Bei einer Dateiaktualisierung:
  - Löse den AST neu auf
  - Berechne `signature` und `dependencies`
  - Wenn sich `signature` geändert hat:
    - Aktualisiere jede abhängige Datei

⚠ Ohne Signature:

- Änderungen wirken sich auf alle Knoten in isolierten Subgraphen aus, z.B. D->C->B->A
- In großen Libraries mit vielen unabhängigen Komponenten akzeptabel
- In Anwendungen meist die gesamte Codebase

Compiler vs Language Server

Compiler

- Einmaliger Prozess
- Dateien zu Beginn bekannt
- Kein Inmemory Cache
- Dateiänderungen müssen über Timestamps oder Checksums ermittelt werden

Language Server

- Langläufiger Prozess
- Dateien zu Beginn unbekannt (1)
- Inmemory Cache
- Watcher für Dateiänderungen

(1) können aber auch umständlich geladen werden:
https://stackoverflow.com/questions/49402283#49529838

Inkrementeller Compiler

- Vorteil: Schnellere Kompilierung von großen Projekten
- Lösung: Speichern von Buildcache als Datei in Output, z.B.:

```
{"files": [
  {"path": "A.dyv",
   "hash": "ab34fe18", // Bestimmt, ob sich der Dateiinhalt geändert hat (schnell, z.B. SHA)
   "signature": "fedd03ab",
   "dependencies": ["B.dyv", "C.dyv"] // Direkte Abhängigkeiten
  }, …
],
"version": …, "options": {…} // Sonstige Build-Parameter, die das Ergebnis beeinflussen
}
```

Mehr Infos

- Inkrementelle Kompilierung von Teilen des ASTs: https://langdev.stackexchange.com/questions/2876/what-are-some-techniques-for-faster-fine-grained-incremental-compilation-and-st
- Implementierung im Rust Compiler: https://blog.rust-lang.org/2016/09/08/incremental.html
- Gradle Java Compilation Avoidance (ABI-kompatible Änderungen): https://docs.gradle.org/current/userguide/java_plugin.html#sec:java_compile_avoidance
- Gradle Incremental Builds (Datei-Hashes/"Fingerprints"): https://docs.gradle.org/current/userguide/incremental_build.html#sec:how_does_it_work
- Gradle Incremental Java Compilation (Ermitteln von Dependencies): https://docs.gradle.org/current/userguide/java_plugin.html#sec:incremental_compile
- SBT (Scala) API Hashing: https://www.scala-sbt.org/1.x/docs/Understanding-Recompilation.html#hashing-an-api-representation
- TypeScript tsbuildinfo Optimierung: https://devblogs.microsoft.com/typescript/announcing-typescript-4-3/#tsbuildinfo-is-smol