- Maven
  - Ant verfolgt einen imperativen Ansatz: Vorgabe der Arbeitsschritte
    - Ermöglicht sehr individuelle Prozesse
  - Maven verfolgt einen deklarativen Ansatz: Beschreibung der Ziele
    - Erzwingt eine Standardisierung

  - Merkmale
    - Ansatz: *Convention over Configuration*
    - Standardisierte Verzeichnisstruktur
    - Standardisierter Bearbeitungszyklus (*Life Cycle*)
    - Neue Projekte können über Projektvorlagen (*Archetypes*) angelegt werden
    - Management von Abhängigkeiten
    - Erweiterungen über *Plug-ins*

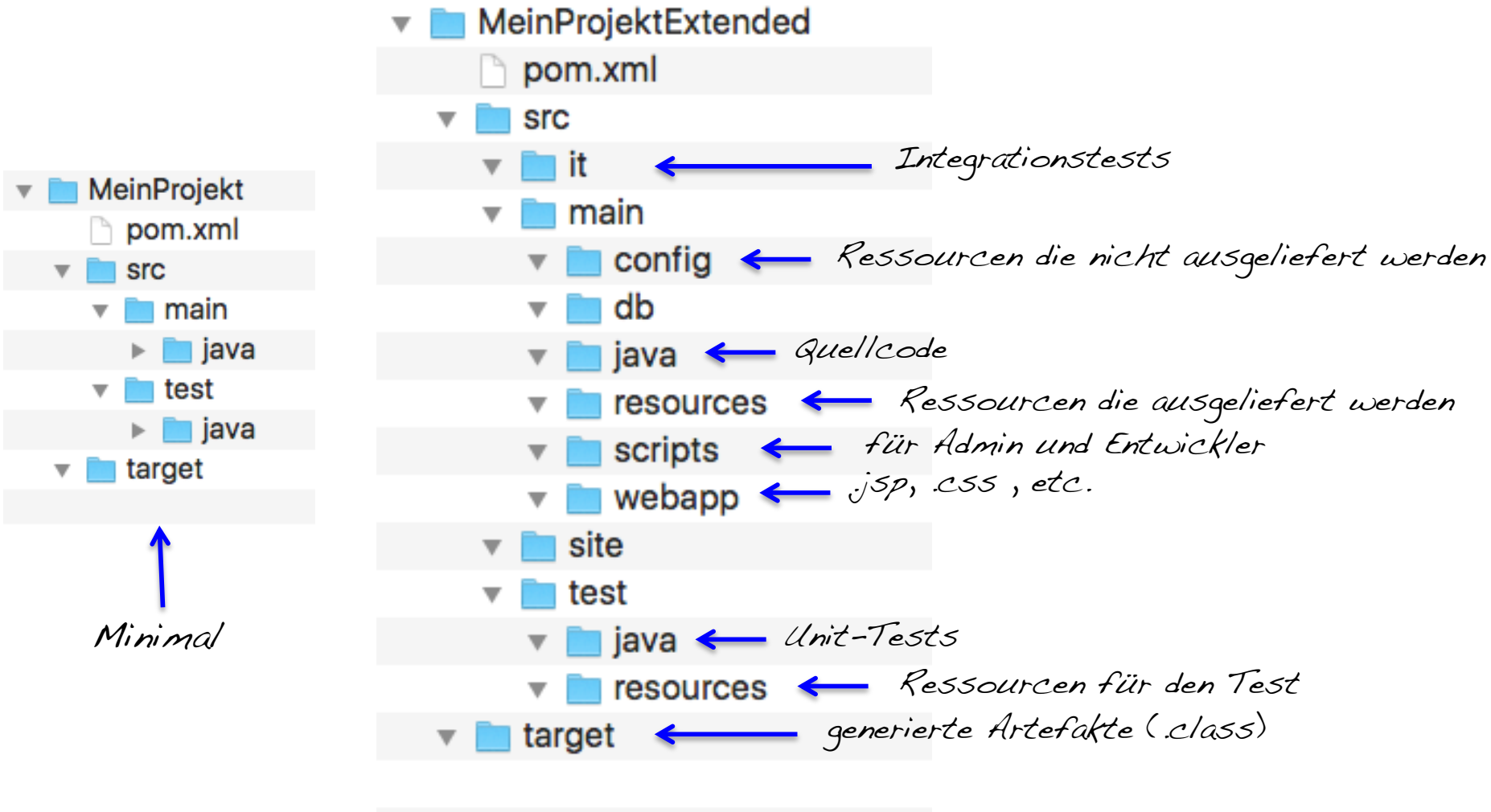
- Der Build-Prozess eines Projekts wird in der Datei `pom.xml` (pom : Project Object Model) konfiguriert

```
<project xmlns=http://maven.apache.org/POM/4.0.0
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>de.fhdortmund</groupId>
   <artifactId>Utility</artifactId>
   <packaging>jar</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>Utility</name>
   <url>http://www.fh-dortmund.de</url>
   <developers>
      <developer>
         <id>dwiesmann</id>
         <name>Dirk Wiesmann</name>
         <email>dirk.wiesmann@fh-dortmund.de</email>
         <properties><active>true</active></properties>
      </developer>
   </developers>
</project>
```

Ausgangsbasis

pom.xml

- Maven-Verzeichnisstruktur

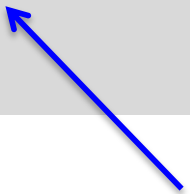


- Maven: Management von Abhängigkeiten
  - In einem Softwareprojekt bestehen in der Regel Abhängigkeiten zu weiteren (externen) Artefakten
  - Zum Beispiel zu Programmbibliotheken: `junit-jupiter-5.9.1.jar`, `log4j-api-2.19.0.jar`
  - Die Abhängigkeiten werden in Maven über die folgenden Attribute aufgelöst

Attribut	**Bedeutung**	**Beispiel**
`groupId`	Organisation, die für das Projekt/Produkt zuständig ist	`de.fh.dortmund`
`artifactId`	Identifizierung des generierten Artefakts	`myProg`
`version`	Versionsnummer	`1.0.0-SNAPSHOT`
`type`	Typ / Art der Paketierung	`WAR`

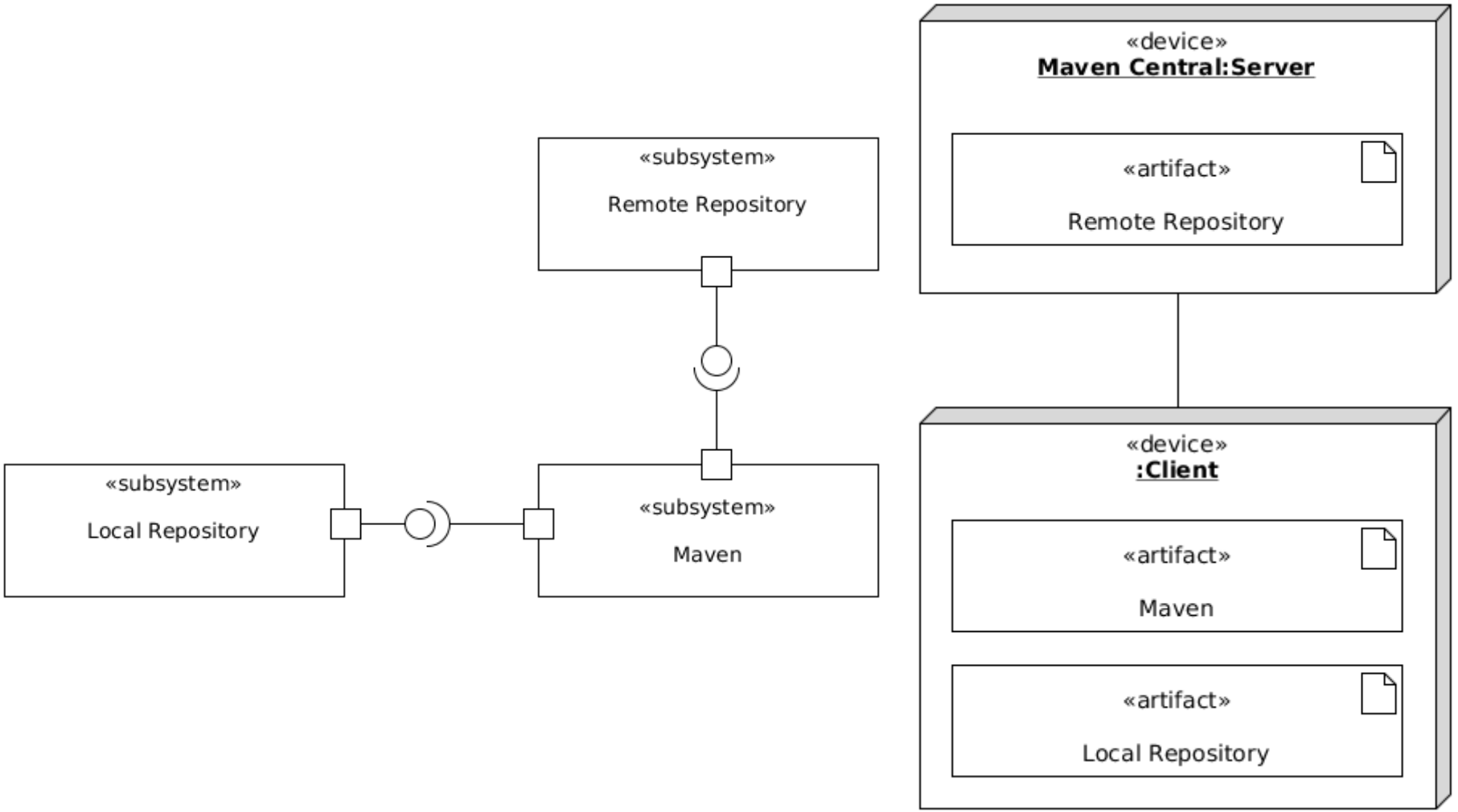
- Alle Abhängigkeiten werden unter dem Element `<dependencies>` aufgelistet
- Für jede Abhängigkeit wird ein `<dependency>`-Eintrag aufgenommen

```
<dependencies>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.9.1</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```



Zeitraum in der die Abhängigkeit zur Verfügung (im Klassenpfad) steht, z.B.
compile : immer (default)
test: während der Testaktivitäten

- Die erforderlichen Artefakte werden automatisch von einem zentralen Repository geladen und in einem lokalen Repository abgelegt



- Indirekte Abhängigkeiten werden automatisch aufgelöst

```
> mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Utility 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ Utility ---
[INFO] de.fhdortmund:Utility:jar:1.0-SNAPSHOT
[INFO] +- junit:junit:jar:4.11:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] \- log4j:log4j:jar:1.2.17:compile
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.808 s
[INFO] Finished at: 2015-11-19T13:32:03+01:00
[INFO] Final Memory: 12M/60M
[INFO] ------------------------------------------------------------------------
```

Anzeige der Abhängigkeiten in einem Projekt

- Der Einsatz eines eigenen Repository Managers ist möglich (z.B. Nexus)
  - Der eigene Repository Manager unterbindet den direkten Zugriff von Maven auf allgemeine Repositories im Internet (z.B. Maven Central)

- Maven: *Life Cycle* und Phasen
  - Standardmäßig werden drei verschiedene *Life Cycle* unterschieden
    a) `Default`: Übersetzen, Paketieren und Verteilen eines Projekts
    b) `Clean`: Löschen von temporären und generierten Artefakten
    c) `Site`: Erzeugung von Dokumentation
  - Jeder *Life Cycle* besteht aus mehreren Schritten, die als Phasen bezeichnet werden
  - Eine Phase kann gezielt ausgeführt werde
    - Alle Vorgängerphasen werden damit ebenfalls ausgeführt

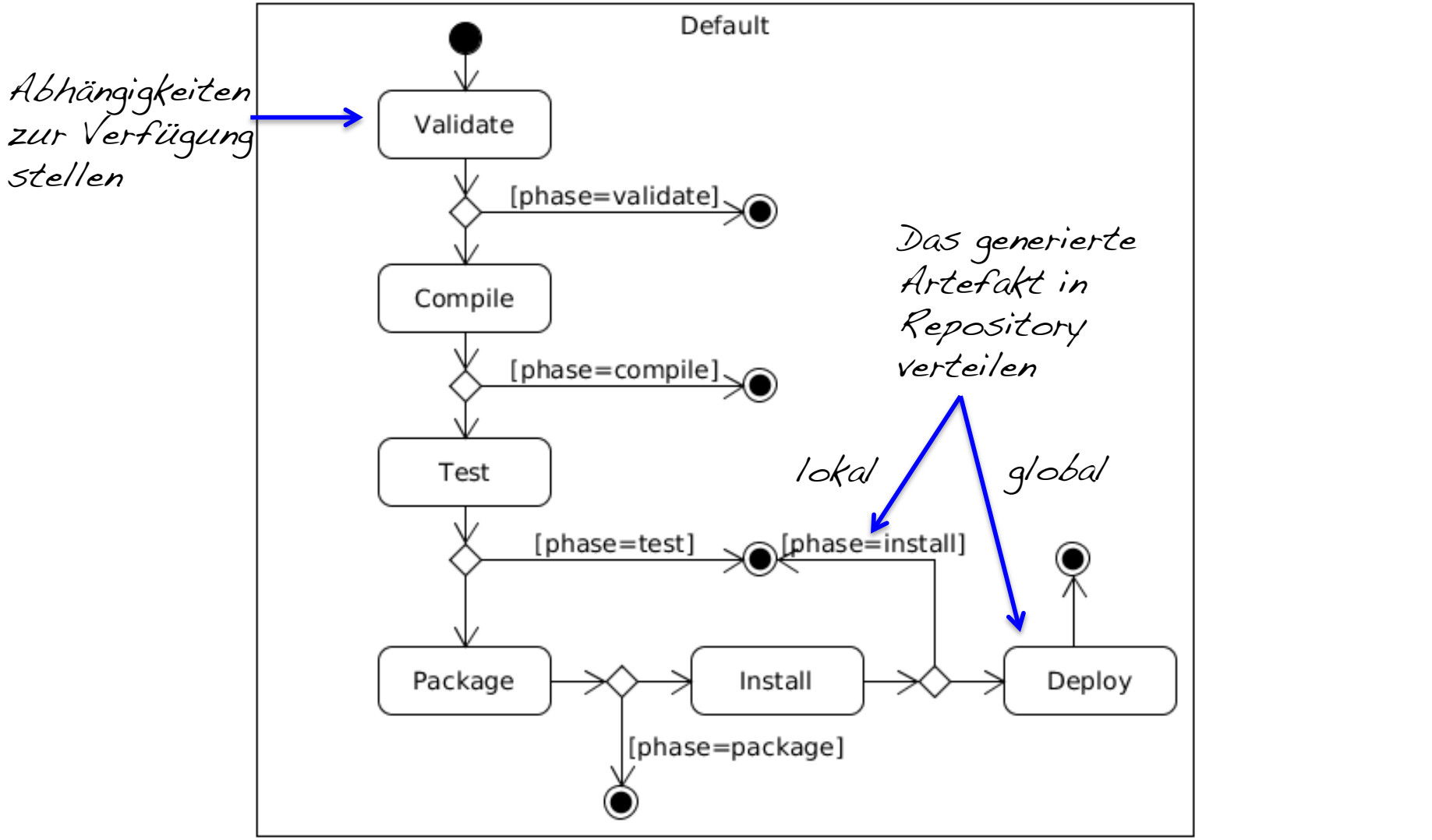
```
> mvn test
```

    - Eine Kombination von Phasen aus unterschiedlichen *Life Cycle* ist möglich

```
> mvn clean test
```

- Der `Default`-Life-Cycle

nur die zentralen Phasen sind dargestellt



- In jeder Phase werden spezielle Plug-Ins aufgerufen
  - In der Compile-Phase wird z.B. das Maven-Standard-Plug-In `Compiler` aufgerufen
- In den Plug-Ins werden verschiedene Ziele (*goals*) realisiert
  - Das Plug-In Compiler hat die beiden Ziele (*goals*) `compile` und `testCompile`
- Ein spezielles Ziel kann über die Notation *`Plug-In`*`:`*`Goal`* aufgerufen werden
- Beispiele:

```
> mvn compiler:compile
```

Die Quellcode-Dateien werden übersetzt (ohne Testcode)

```
> mvn compiler:testCompile
```

Nur der Testcode (Unit-Test) wird übersetzt

- Der Clean-Life-Cycle
  - In der zentralen Phase `clean` wird das Maven-Standard-Plugin `Clean` aufgerufen
  - Das `Clean`-Plug-In besitzt nur das Ziel `clean`
  - Dieses Ziel löscht alle temporären und automatisch generierten Dateien

```
> mvn clean
```

Phase

Standard Verbindung (lifecycle binding): Phase und Goal

```
> mvn clean:clean
```

Plug-In und Goal

- Der `Site`-Life-Cycle
  - In der zentralen Phase `site` wird eine Projektdokumentation erstellt.
  - Standardmäßig werden im Ordner `target/site` Web-Seiten angelegt, die Informationen aus der `pom.xml` aufbereiten (z.B. Abhängigkeiten und Plug-Ins)

einige Built-in Lifecycle Bindings

Phase	**Plug-In:Goal**
`clean`	`clean:clean`
`compile`	`compiler:compile`
`test`	`surefire:test`
`test-compile`	`compiler:testCompile`
`package`	`ejb:ejb, oder jar:jar, oder war:war`
`site`	`site:site`

- Maven: Projektvorlagen (*Archetypes*)
  - Die Verzeichnisstruktur und die `pom.xml` muss nicht manuell erstellt werden
  - Maven bietet eine Reihe von Projektvorlagen (*Archetypes*), aus denen die Grundstrukturen automatisch generiert werden können
  - Eine Katalog von Artefakten kann über das Plug-In `archetype` angezeigt werden

```
> mvn archetype:generate
```

    - Eine Vorlage kann dann interaktiv ausgewählt werden
    - Die Einrichtung des Projektes erfolgt über einen Dialog

  - Eine Vorlage kann auch direkt gewählt werden

```
> mvn archetype:generate
      -DarchetypeArtifactId=maven-archetype-quickstart
```

Basisstruktur

- Maven: Grundeinstellungen
  - Im Build-Bereich der `pom.xml` sind die folgenden Plugins einzutragen

```
<build>
   <plugins>
     …
   </plugins>
</build>
```

  - Aufgrund der schnell steigenden Anzahl an unterschiedlichen Java-Versionen ist es häufig notwendig, eine spezifische Version vorzugeben

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.8.1</version>
   <configuration>
      <source>14</source>
      <target>14</target>
   </configuration>
</plugin>
```

- Neben der `junit-jupiter`-Dependency wird von JUnit 5 eine aktuelle Version vom surefire-Plugin für die Ausführung der Unit-Tests benötigt

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <version>3.0.0-M5</version>
</plugin>
```

- Maven: Reports für die Qualitätssicherung
  - Unter dem Element `<reporting>` können Plug-Ins `<plugins>` für die Berichterstellung eingebunden werden

```
<reporting>
  <plugins>
  </plugins>
</reporting>
```

    - Erstellung einer HTML-Seite mit einer Verknüpfung zu den *Reports*

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-site-plugin</artifactId>
   <version>3.12.1</version>
</plugin>
```

    - Implementierungsdokumentation/Dokumentationsextraktion: Javadoc

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-javadoc-plugin</artifactId>
   <version>3.4.1</version>
</plugin>
```

- Unit-Test-Reports

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-report-plugin
   </artifactId>
   <version>3.0.0-M5</version>
</plugin>
```

pom.xml

surefire-report:report

-> erstellt ein Bericht der Testergebnisse als HTML-Seite

- Project-Info (Projektdaten)

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>
      maven-project-info-reports-plugin
   </artifactId>
   <version>2.9</version>
</plugin>
```

pom.xml

z.B. project-info-reports:summary
project-info-reports:dependencies

- Aus Kompatibilitätsgründen muss zurzeit das `project-info-reports-plugin` um den folgenden Eintrag ergänzt werden (innerhalb von `<plugin></plugin>`)

```
<dependencies>
   <dependency>
      <groupId>org.apache.maven.shared</groupId>
      <artifactId>maven-shared-jar</artifactId>
      <version>1.2</version>
      <exclusions>
         <exclusion>
            <groupId>com.google.code.findbugs</groupId>
            <artifactId>bcel-findbugs</artifactId>
         </exclusion>
      </exclusions>
   </dependency>
   <dependency>
      <groupId>org.apache.bcel</groupId>
      <artifactId>bcel</artifactId>
      <version>6.2</version>
   </dependency>
</dependencies>
```

Glass-Box-Test (White-Box-Test) / Strukturtest

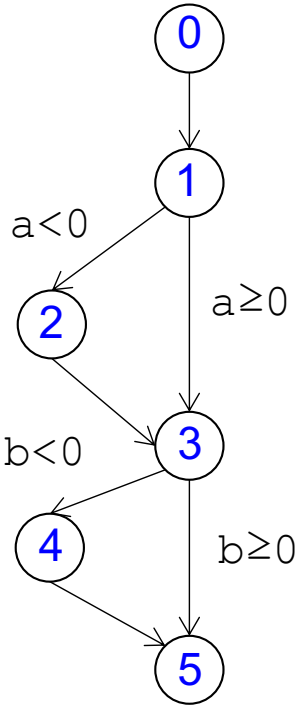
Der Black-Box-Test berücksichtigt nicht die innere Struktur des Programms
Über einen reinen Funktionstest (Black-Box-Test) können wir daher nicht sicherstellen, dass auch alle Teile des Programms getestet werden

Der Programmcode ist sichtbar. Es wird ein Strukturtest vorgenommen. Dabei wird die Überdeckung des Codes durch die Testfälle analysiert. Das Ziel ist es, eine vorgegebene Überdeckungsrate zu erreichen.

- Um die Überdeckung bestimmen zu können, muss das Programm mit zusätzlichen Anweisungen für die Messung instrumentiert werden
- Mit Hilfe der Instrumentierung kann z.B. gezählt werden, wie häufig eine Programmzeile ausgeführt wird
- Es werden verschiedene Überdeckungskriterien unterschieden:
  1. Anweisungsüberdeckung
  2. Zweigüberdeckung
  3. Bedingungsüberdeckung
  4. Datenflussbasierte-Überdeckung

- Kontrollflussgraph
  - Ein Glass-Box-Test orientiert sich am Kontrollflussgraphen einer Code-Sequenz
  - Anweisungen werden im Kontrollflussgraphen als Knoten dargestellt
  - Der Kontrollfluss zwischen den Anweisungen wird über die Kanten repräsentiert

```
0:  public static int manhatten(int a, int b){
1:     if (a < 0)
2:        a = -a;
3:     if (b < 0)
4:        b = -b;
5:     return a+b;
    }
```



Übungsaufgabe

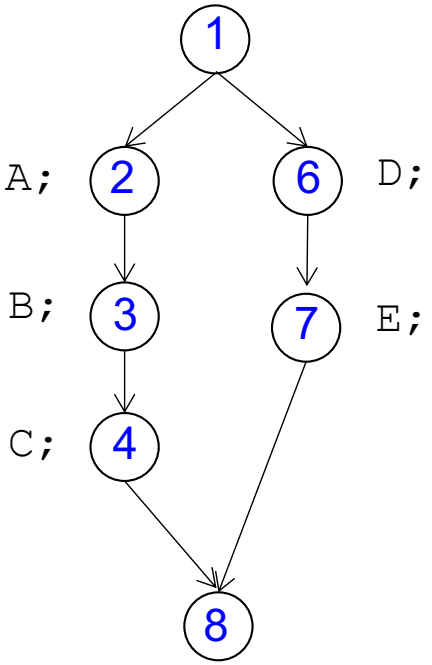
Erstellen Sie die Kontrollflussgraphen für die folgenden elementaren Verzweigungs- und Schleifenkonstrukte

- `if (B) X;`
- `if (B) X; else Y;`
- `while (B) X;`
- `do X; while (B);`

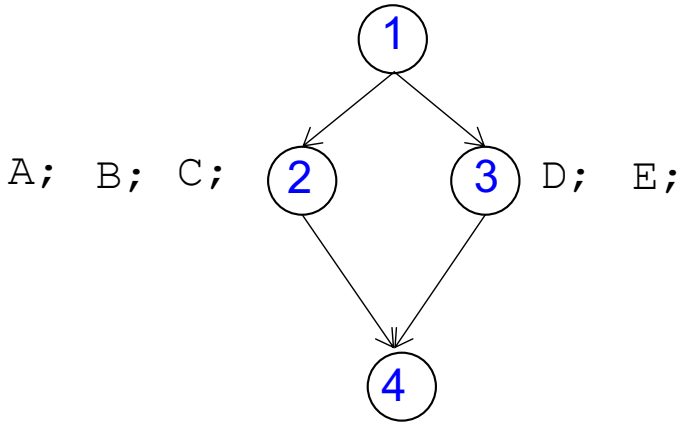
– Klassifikation von Kontrollflussgraphen

```
1:   if (B){
2:      A;
3:      B;
4:      C; }
5:   else  {
6:      D;
7:      E; }
```

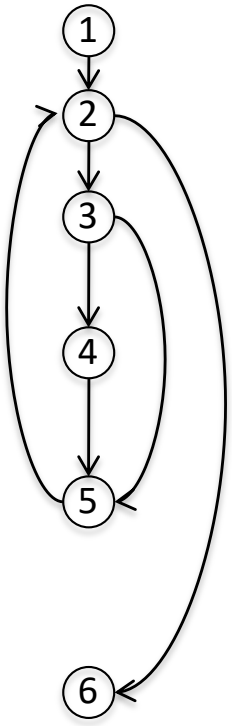
Expandiert



Kolabiert



– Als Beispiel betrachten wir den folgenden Programmausschnitt:



Kontrollflussgraph

```
// eingabe ist vom Typ String
int c=0, v=0;

while((c < eingabe.length()) && (eingabe.charAt(c) >= 'A')
                             && (eingabe.charAt(c) <= 'Z')){
  if((eingabe.charAt(c) == 'A') || (eingabe.charAt(c) == 'E') ||
     (eingabe.charAt(c) == 'I') || (eingabe.charAt(c) == 'O') ||
     (eingabe.charAt(c) == 'U')){
    v++;
  }

  c++;
}


System.out.println("Der Text enthält " + c + " Grossbuchstaben. "
                   + "Davon sind " + v + " Vokale.");
```

1. Anweisungsüberdeckung (*statement coverage*) / C_0-Test
   - Jede Anweisung im Programm muss im Test mindestens einmal ausgeführt werden (100% Anweisungsüberdeckung)
   - Eine Anweisungsüberdeckung > 80 % sollte angestrebt werden

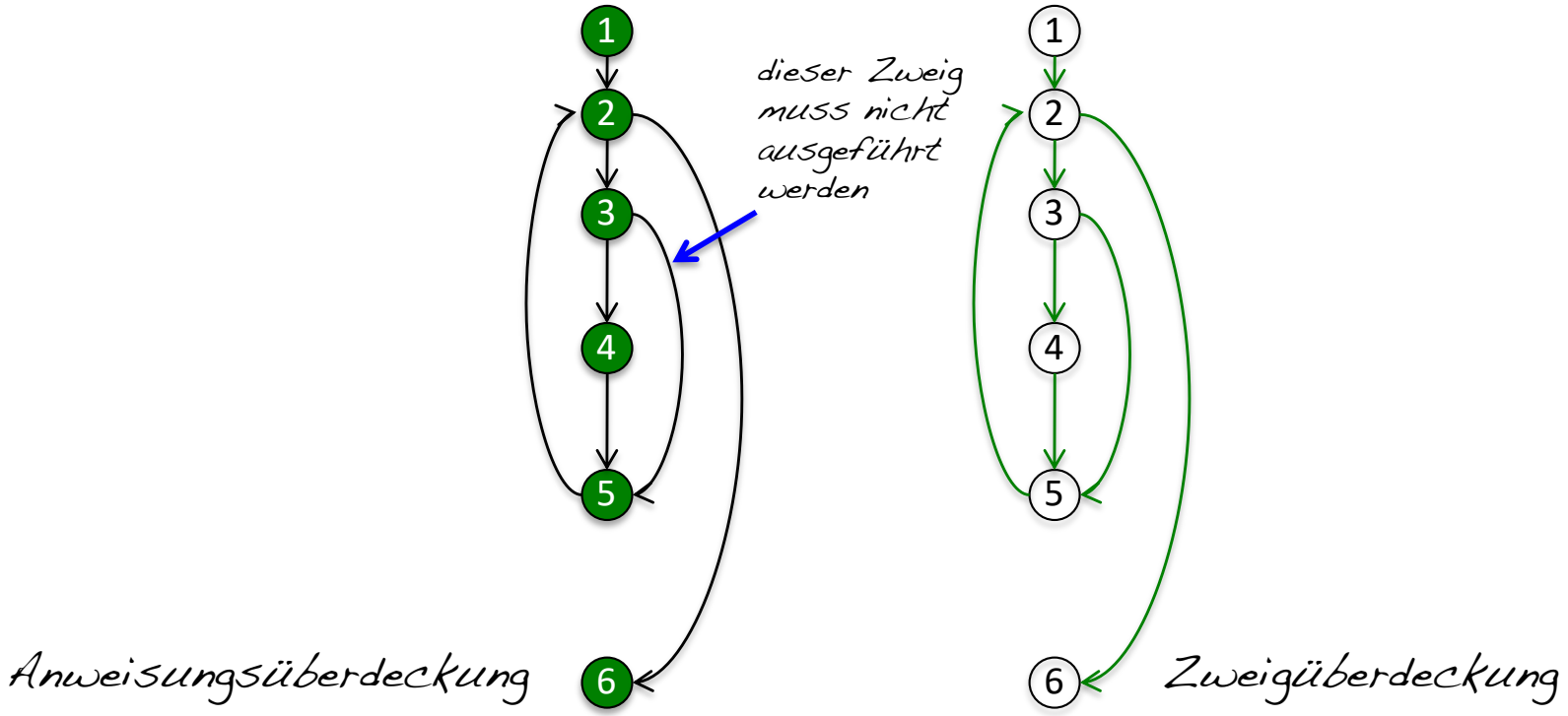
Übungsaufgabe

Kreuzen Sie die Anweisungen an, die für die jeweilige Testeingabe ausgeführt werden (siehe Kontrollflussgraph auf der vorherigen Seite)!

Eingabe	**1**	**2**	**3**	**4**	**5**	**6**
hallo						
FGH						
FOM						

2. Zweigüberdeckung (Entscheidungsüberdeckung) / C_1-Test

- Jeder Zweig im Programm wird im Test mindestens einmal ausgeführt (100% Zweigüberdeckung)
- Die Zweigüberdeckung ist strenger, als die Anweisungsüberdeckung, wenn das Programm leere Zweige enthält (z.B. `if-then` ohne `else`-Zweig)



- Automatisierung mit Maven
  - Codeüberdeckung: Cobertura

```
<plugin>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>cobertura-maven-plugin</artifactId>
   <version>2.7</version>
</plugin>
```

zurzeit Probleme mit neueren Java-Versionen

  - Codeüberdeckung: JaCoCo

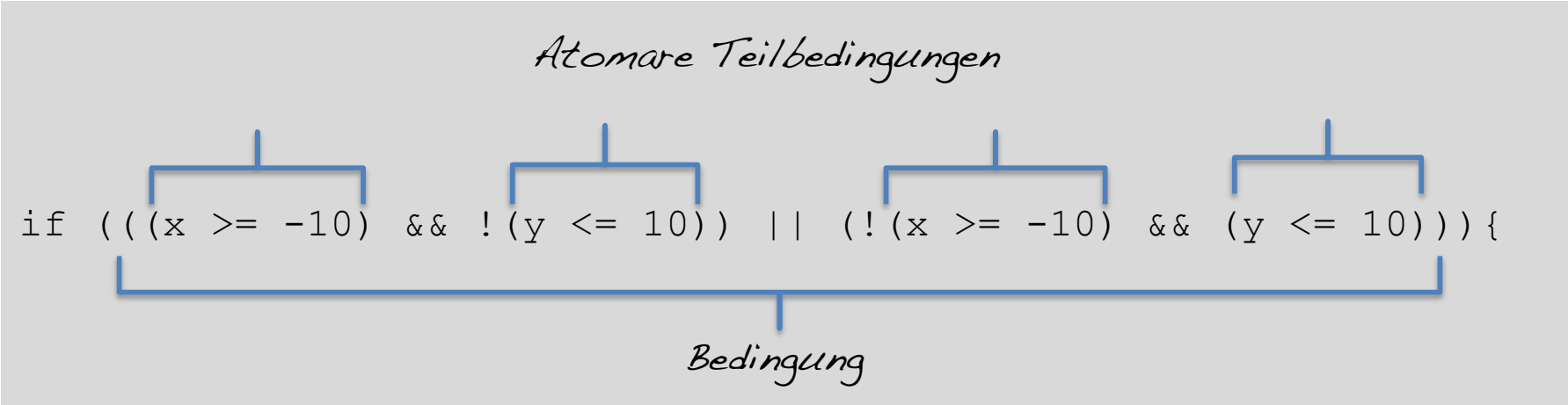
```
<build>
   <plugins>
      <plugin>
         <groupId>org.jacoco</groupId>
         <artifactId>jacoco-maven-plugin</artifactId>
         <version>0.8.6</version>
         <!-- weitere Konfiguration -->
      </plugin>
   </plugins>
</build>
```

siehe Praktikum

goal z.B. jacoco:report

3. Bedingungsüberdeckung

- Die Zweigüberdeckung stellt sicher, dass eine logische Bedingung während des Tests mindestens einmal `true` und einmal `false` ist
- Wenn die Bedingung zusammengesetzt ist, wird damit aber nicht sichergestellt, dass die Teilbedingungen korrekt formuliert sind
- Eine Bedingung ist eine Verknüpfung von Teilbedingungen mit den logischen Operatoren AND, OR und NOT
- Eine (atomare) Teilbedingung enthält keine logischen Operatoren, sondern höchstens Relationssymbole (<,>,==)



- Die Bedingungsüberdeckung bestimmt, wie genau die Bedingungen im Test überprüft werden
- Es werden dabei drei Arten unterschieden:

① **Einfache Bedingungsüberdeckung:** alle atomaren Teilbedingungen müssen während des Tests einmal `true` und einmal `false` ergeben

Testfälle (x=0, y=0), (x=-15, y=15)

② **Mehrfach-Bedingungsüberdeckung:** alle Variationen der Belegung der atomaren Teilbedingungen werden getestet. Achtung: 2^n Variationsmöglichkeiten bei *n* atomaren Bedingungen

Testfälle (x=0, y=15), (x=0, y=0), (x=-15, y=15), (x=-15, y=0)

③ **Minimale-Mehrfach-Bedingungsüberdeckung:** jede Teilbedingung muss mindestens einmal `true` und einmal `false` sein

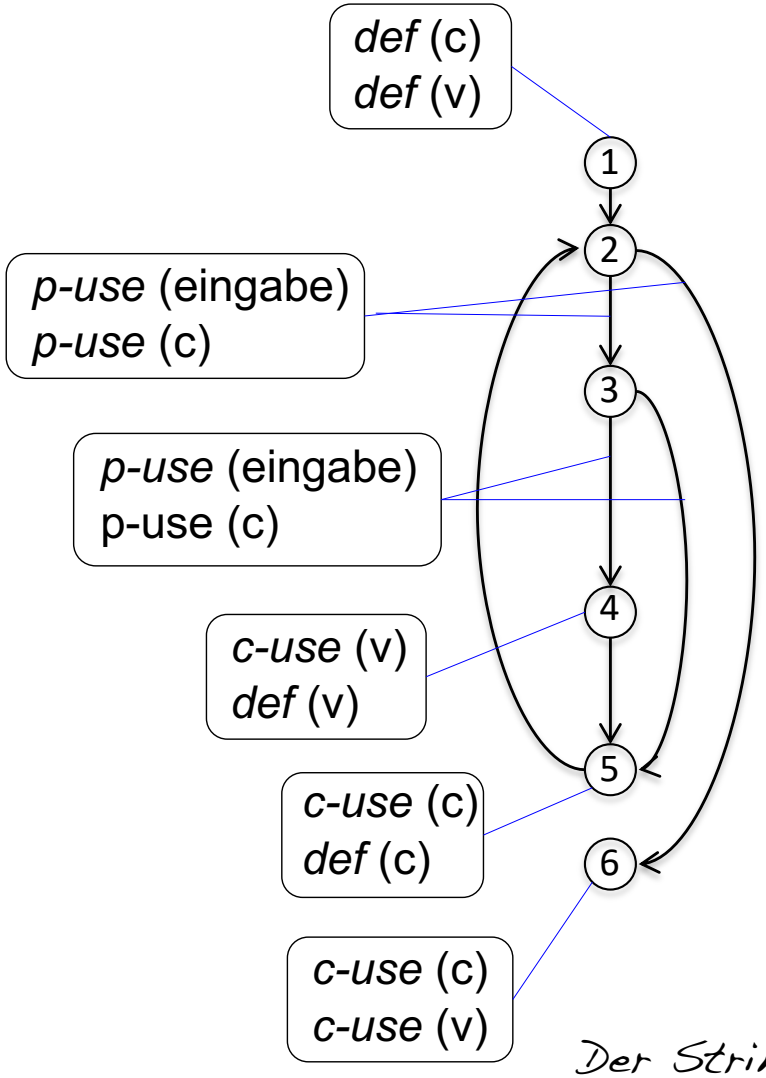
Testfälle (x=-15, y=15), (x=-15, y=0), (x=0, y=15)

- Eine einfache Bedingungsüberdeckung bietet weniger, als eine Zweig- und Anweisungsüberdeckung
- Die Mehrfach-Bedingungsüberdeckung enthält zwar die Zweigüberdeckung, sie ist aber sehr aufwendig. Zudem sind nicht alle Variationen realisierbar (z.B. können in dem Beispielprogramm niemals die Teilbedingungen `(eingabe.charAt(c) == 'A')` und `(eingabe.charAt(c) == 'E')` gleichzeitig den Wert `true` annehmen
- Die Minimale-Mehrfach-Bedingungsüberdeckung ist ein praktikabler Kompromiss

4. Datenflussbasierte Überdeckung (*Defs-Uses*-Überdeckung)

- Die Verwendung von Variablen wird analysiert
- Fragestellung: Führt der Wert einer Variablen, bei der Nutzung einer Variablen an weiteren Stellen im Programm zu einem Fehler?
- Es werden verschiedene Kategorien unterschieden
  - Die Definition (*def*) einer Variablen / Wertzuweisung
  - Die Verwendung einer Variablen in einem Ausdruck zur Berechnung eines Wertes (berechnende Benutzung, *computational-use*, *c-use*)
  - Die Verwendung einer Variablen in Bedingungen bzw. Prädikaten zur Berechnung von Wahrheitswerten (prädikative Benutzung, *predicate-use*, *p-use*)

- Kontrollflußgraph in der Datenflußdarstellung:




```
// eingabe ist vom Typ String
int c=0, v=0;

while((c < eingabe.length())
      && (eingabe.charAt(c) >= 'A')
      && (eingabe.charAt(c) <= 'Z')){
  if((eingabe.charAt(c) == 'A')
      || (eingabe.charAt(c) == 'E')
      || (eingabe.charAt(c) == 'I')
      || (eingabe.charAt(c) == 'O')
      || (eingabe.charAt(c) == 'U')){
    v++;
  }

  c++;
}
System.out.println("Der Text enthält "
    + c + " Grossbuchstaben. "
    + "Davon sind " + v + " Vokale.");
```

Der String eingabe wurde ausgeblendet

- Wir identifizieren nun bestimmte Kanten- und Kontenmengen zur späteren Definition von Überdeckungskriterien
- Sei $V=\{1,...,n\}$ die Menge der Knoten im Kontrollflussgraphen
- Dann ist $p=(s_0,...,s_k)$, mit $s_j \in V$ ein Pfad, wenn s_i mit s_{i+1} durch eine Kante verbunden ist (für $0 \leq i < k$)

- Definitionsfreier Pfad: Wird die Variable x im Knoten s_0 definiert, dann ist der Pfad $(s_0,...,s_k)$ genau dann definitionsfrei, wenn die Variable in den Knoten $s_1,...,s_k$ nicht erneut definiert wird

Schwaches Kriterium

- ***All Definitions*** (*All-defs*)
  - Die Testfälle durchlaufen für jede Definition einer Variablen einen definitionsfreien Pfad zu mindestens einem p-use oder c-use

- ***All c-uses***
  - Die Testfälle durchlaufen für jede Definition einer Variablen einen Pfad zu allen definitionsfrei erreichbaren c-uses

- ***All p-uses***
  - Die Testfälle durchlaufen für jede Definition einer Variablen einen Pfad zu allen definitionsfrei erreichbaren p-uses
- ***All uses***
  - Die Testfälle durchlaufen für jede Definition einer Variablen einen Pfad zu allen definitionsfrei erreichbaren p- und c-uses
- ***All c, some p***
  - Die Testfälle erfüllen das *All-c-uses*-Kriterium. Existiert zu einer Definition keine berechnende Nutzung, so wird zusätzlich mindestens ein definitionsfreier Pfad zu einer prädikativen Nutzung hinzugenommen
- ***All p, some c***
  - Analog zu *All-c-some-p*-Kriterium mit vertauschten Rollen

- Überblick über die Glas-Box-Testverfahren