



# **Verteilte Verarbeitung**

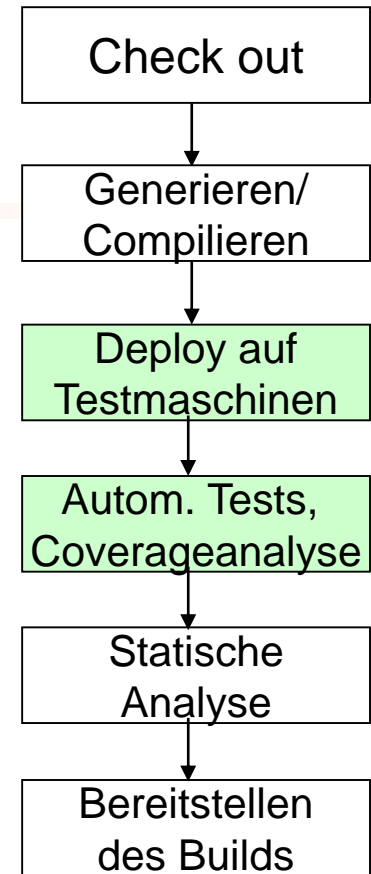
Steilkurs Gradle

# Build-Prozess

- Build der Software umfasst mehr als nur den Compiler-Aufruf
- Build-Prozess am Beispiel Java =
  - Beschaffen der Bibliotheken (z.T. Download)
  - Übersetzen der Quelltexte
  - JavaDoc
  - Junit-Tests ausführen und Überdeckung messen
  - Checkstyle, Findbugs, PMD etc. ausführen
  - Jar/War/Ear-Datei erstellen
  - Deployment auf Java EE Server
- Zusätzlich:
  - ggf. Datenbankschema erzeugen, Testdaten anlegen, Nutzungsrechte vergeben, ....
  - Ggf. Code-Generatoren, Transpiler, Obfuscatoren starten ...

# Ablauf eines Builds

1. Auschecken der Quelltexte aus der Versionsverwaltung
2. Übersetzen der Quelltexte
3. Deployment auf Testsystem
4. Automatisierte Tests ablaufen lassen  
= XUnit, Selenium, ... und Coverage Analyse
5. Statische Code-Analyse  
= Checkstyle, PMD, Findbugs, ...
6. Erstellen der Installationsroutinen / des Setups
7. Bereitstellen des Builds



# Automatisierung Build-Prozess: Warum?

- Wichtig: Software muss **unabhängig von der IDE** übersetzbar und lieferbar sein
- Tool für Automatisierung muss
  - die **Abhängigkeiten** zwischen den Dateien kennen
  - **Fremdwerkzeuge** steuern können: GIT/SVN-Zugriff, Compiler, XUnit, Statische Analyse, JavaDoc, Deployment, ...
  - auf verschiedenen Plattformen laufen
- Tools in Java: **ant (+ivy), Maven** und **Gradle**
  - flexibel konfigurierbar über Makefiles / Buildfiles / POM
    - Ant: ca. 2000, „Skript“ als XML Datei, gibt Schritt für Schritt vor was zu tun ist
    - Maven: ca. 2004, aktueller Marktführer, vieles implizit, Projekt-Eigenschaften werden deklariert in POM-Datei (= Project Object Model in XML)
    - Gradle: ca. 2009, setzt sich gerade durch, verwendet Ant und Maven Konzepte, Skript in Programmier-Sprache Groovy geschrieben
  - Für jedes Werkzeug: Integration in allen IDE (Eclipse, Netbeans, ...).

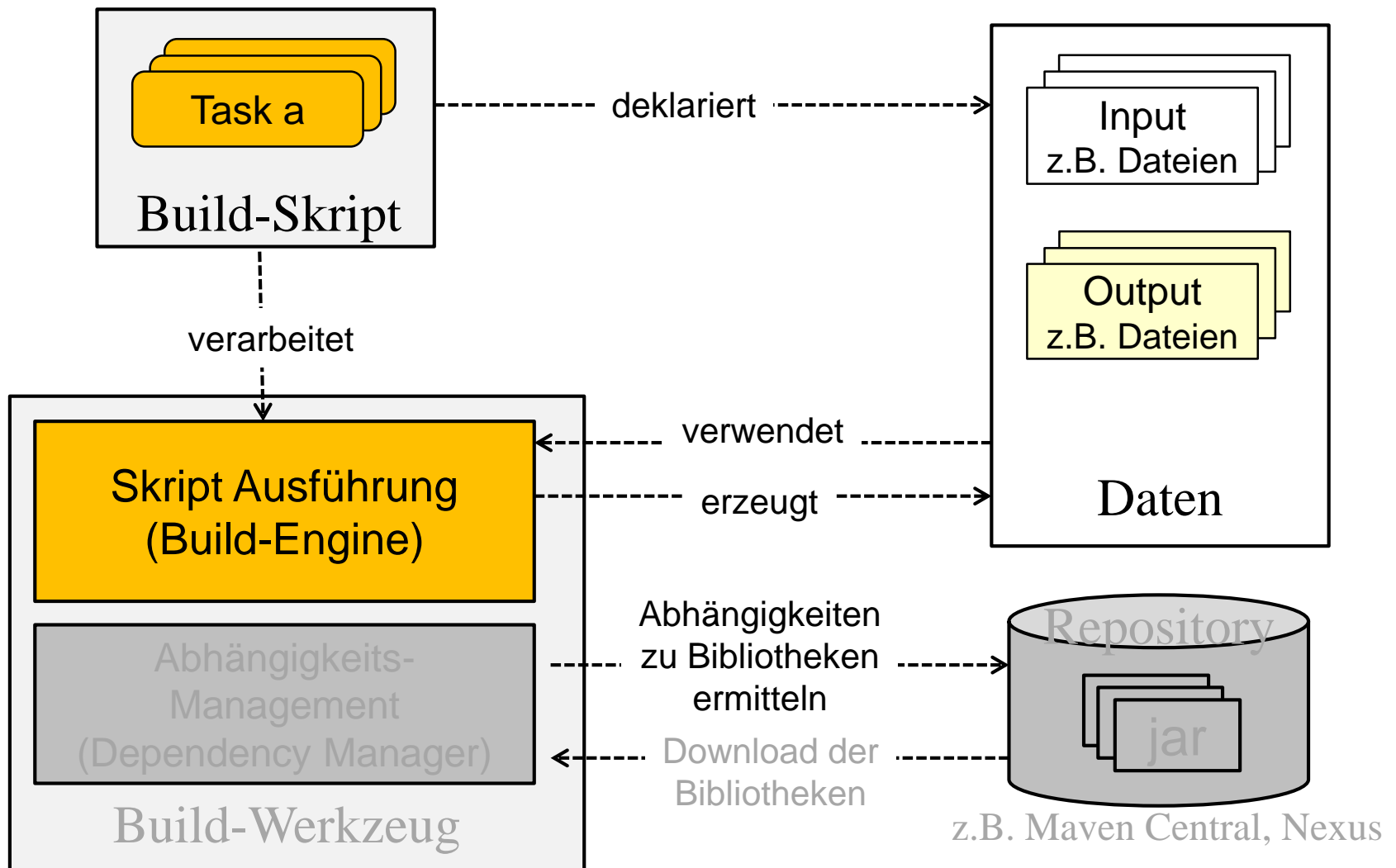


# Build-Werkzeug Gradle



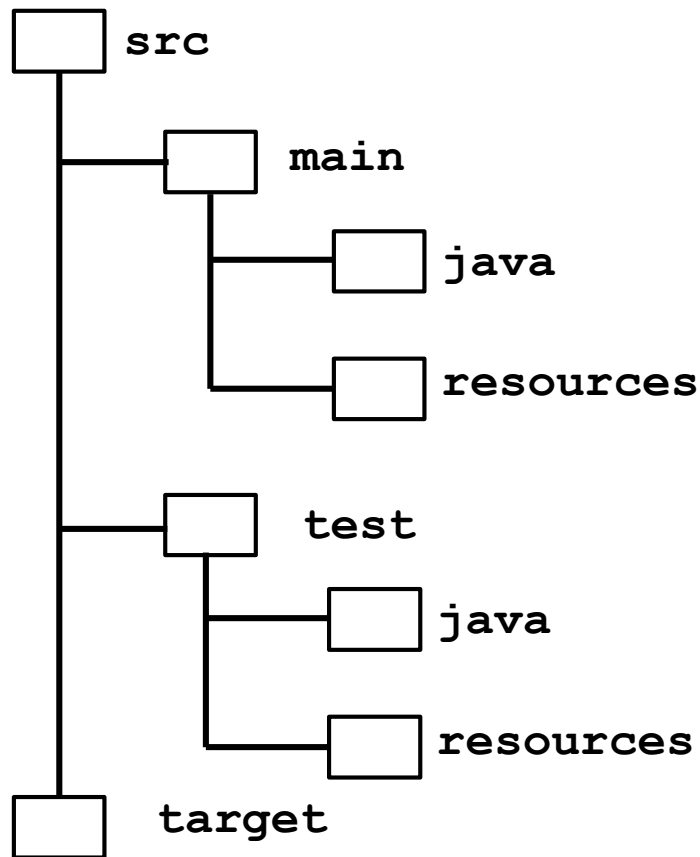
- = Werkzeug um Projekt unabhängig von der IDE (IntelliJ) übersetzen zu können, auch für
  - Continuous Integration (Jenkins) und
  - Continuous Delivery (u.a. docker)
- = Werkzeug zur Automatisierung des „Build-Prozesses“
  - Übersetzen, Javadoc erzeugen,
  - JUnit-Tests ausführen, statische Analyse
  - JAR/WAR/EAR-Datei erzeugen, Deployen, ...
- Scheint Maven abzulösen, verbreitet in Android Szene
- Über (eigene) Plugins gut erweiterbar
- Geschrieben in Groovy (= Skript / Programmiersprache)

# Build-Werkzeug Architekturübersicht



# Skript-Engine: Verzeichnisstruktur

## *Convention over Configuration*



- Gradle und Maven unterstellen bestimmte Verzeichnisstruktur
- Struktur kann im Skript angepasst werden
- Struktur unterstützt
  - Mehrere Programmiersprachen
  - Trennung von Test und Produktiv-Code



# Skripte in Gradle

- = Skript in Programmiersprache Groovy
  - Groovy = Skriptsprache ähnlich zu Python / Ruby mit enger Integration in Java, läuft auf JVM
  - Umstieg ggf. zu Kotlin?
- Bezeichnet als spezielle „DSL“ (Domänen spezifische Sprache) zum Bauen von Software

```
apply plugin: 'java'
```

```
version = 0.1  
sourceCompatibility = 1.7
```

```
repositories {  
    mavenCentral()  
}
```

```
dependencies {  
    testCompile 'junit:junit:4.12'  
}
```

Maven Bibliothek  
<http://repo1.maven.org/maven2/>

Infos wie bei Maven  
Gruppenname  
Name  
Versionsnummer

# Gradle = Skript in der Sprache Groovy

- **Groovy = Skriptsprache auf JVM-Basis**
- **Build-Skript ist ein Skript**
  - Kann programmiert werden: Definition von Methoden und Attributen möglich
  - Abläufe können mit if / while formuliert werden
  - Leicht erweiterbar

Objekt der Klasse **Project**

```
apply plugin: 'java'
```

Aufruf der Methode **apply()**

```
version = 0.1
```

```
sourceCompatibility = 1.7
```

Aufruf der set-Methoden für die Attribute **version** und **sourceCompatibility**

```
repositories {  
    mavenCentral()  
}
```

```
dependencies {  
    testCompile 'junit:junit:4.12'  
}
```

Aufruf der Methode **dependencies**

# Eigenschaften der Klasse Project

<https://docs.gradle.org/current/dsl/org.gradle.api.Project.html#org.gradle.api.Project:tasks>

## Properties

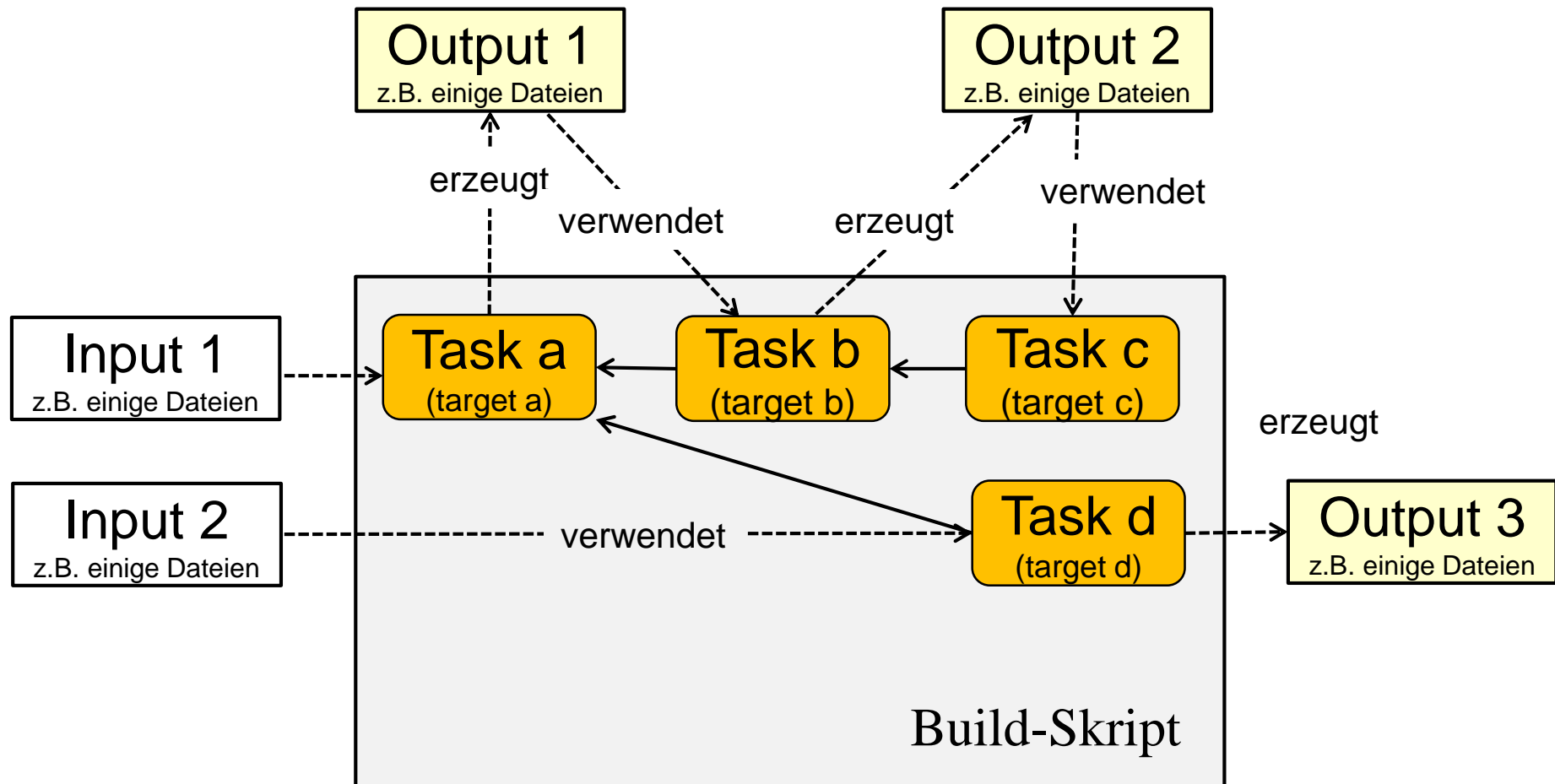
Property	Description
<code>allprojects</code>	The set containing this project and its subprojects.
<code>ant</code>	The <code>AntBuilder</code> for this project. You can use this in your build file to execute ant tasks. See example below.
<code>artifacts</code>	Returns a handler for assigning artifacts produced by the project to configurations.
<code>buildDir</code>	The build directory of this project. The build directory is the directory which all artifacts are generated into. The default value for the build directory is <code>projectDir/build</code>
<code>build</code>	

## Methods

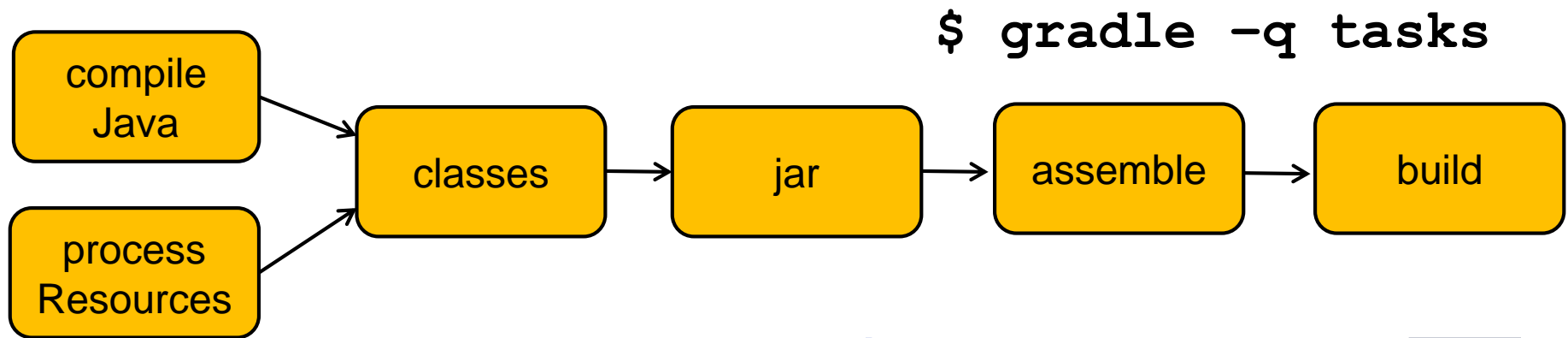
Method	Description
<code>absoluteProjectPath(path)</code>	Converts a name to an absolute project path, resolving names relative to this project.
<code>afterEvaluate(closure)</code>	Adds a closure to be called immediately after this project has been evaluated. The project is passed to the closure as a parameter. Such a listener gets notified when the build file belonging to this project has been executed. A parent project may for example add such a listener to its child project. Such a listener can further configure those child projects based on the state of the child projects after their build files have been run.
<code>afterEvaluate(action)</code>	Adds an action to execute immediately after this project is evaluated.
<code>allprojects(action)</code>	Configures this project and each of its sub-projects.
<code>apply(closure)</code>	Applies zero or more plugins or scripts.
<code>apply(options)</code>	Applies a plugin or script, using the given options provided as a map. Does nothing if the plugin has already been applied.

# Build-Prozess verarbeitet Dateien

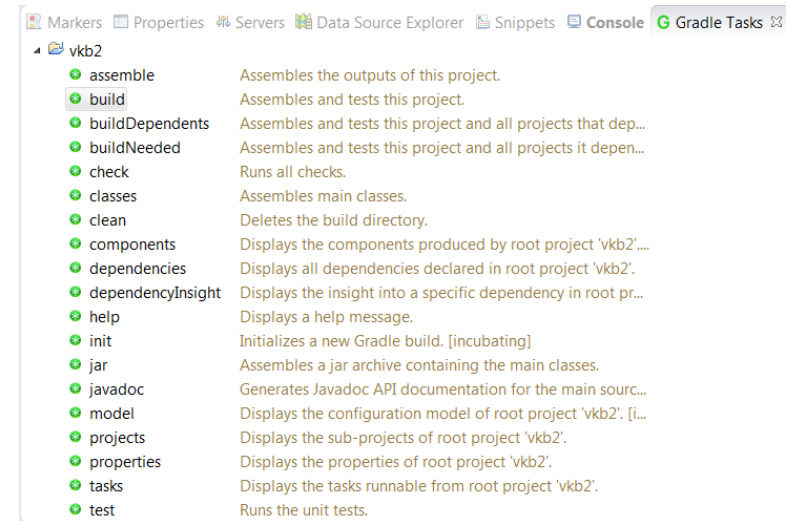
(Begriffsunterschied zwischen den Werkzeugen, was ein „Task“ ist)



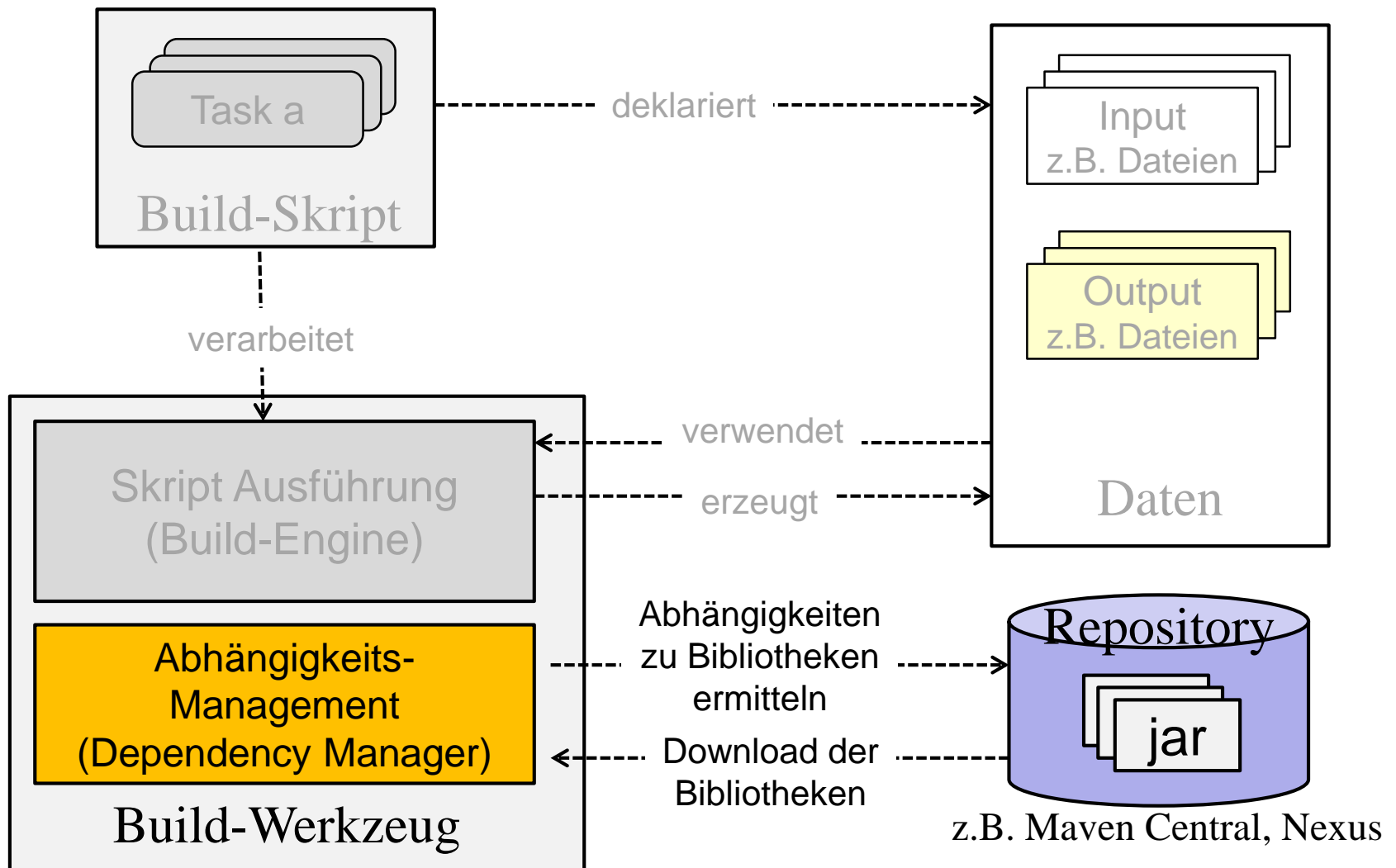
# Gradle: Java-Plugin Build-Lebenszyklus



- Java-Plugin bietet vorkonfigurierte Tasks an
- Diese können auf der Shell oder in IntelliJ ausgeführt werden und erledigen den Buildprozess bis zu dem angegebenen Schritt



# Abhängigkeitsmanagement




# Abhängigkeitsmanagement

- Beobachtung: Große Projekte haben sehr viele Abhängigkeiten zu fremden Bibliotheken, z.B.
  - JUnit, Hibernate, Apache-Commons ...
- Bibliotheken hängen ihrerseits von anderen Bibliotheken ab -> Große Abhängigkeitsgraphen
- Beispiel: Bibliothek Hibernate (Zugriffsschicht)

Abhängigkeiten

```
ult - Configuration for default artifacts.
org.hibernate:hibernate-core:3.6.7.Final
+--- antlr:antlr:2.7.6
+--- commons-collections:commons-collections:3.1
+--- dom4j:dom4j:1.6.1
+--- org.hibernate:hibernate-commons-annotations:3.2.0.Final
|    \--- org.slf4j:slf4j-api:1.5.8 -> 1.6.1
+--- org.hibernate.javax.persistence:hibernate-jpa-2.0-api:1.0.1.Final
+--- javax.transaction:jta:1.1
\--- org.slf4j:slf4j-api:1.6.1
```



# Abhängigkeitsmanagement

/2

- Abhängigkeiten zu fremden Bibliotheken werden im Build-Skript explizit angegeben

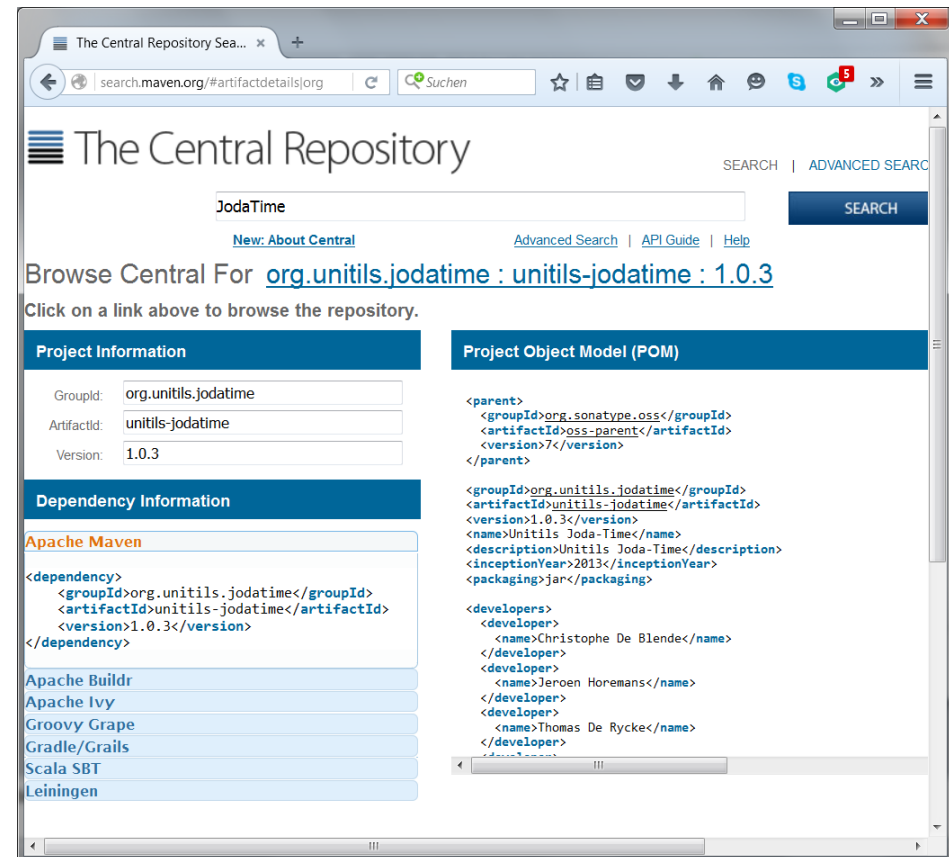
- Maven:

```
<dependency>
  <groupId>org.unitils.jodatime
</groupId>
  <artifactId>unitils-jodatime
</artifactId>
  <version>1.0.3</version>
</dependency>
```

- Gradle:

```
'org.unitils.jodatime:unitils-
jodatime:1.0.3,
```

- Build-Werkzeug lädt die notwendigen Bibliotheken bei Bedarf aus dem Internet, z.B. von <https://repo1.maven.org>





# Abhängigkeitsmanagement (Beispiel aus Platzgründen schon gradle)

```
apply plugin: 'java',

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.hibernate:hibernate-core:3.6.7.Final'
    testCompile 'junit:junit:4.+'
}
```

\$ gradle -q dependencies

```
default - Configuration for default artifacts.
\--- org.hibernate:hibernate-core:3.6.7.Final
      +--- antlr:antlr:2.7.6
      +--- commons-collections:commons-collections:3.1
      +--- dom4j:dom4j:1.6.1
      +--- org.hibernate:hibernate-commons-annotations:3.2.0.Final
           | \--- org.slf4j:slf4j-api:1.5.8 -> 1.6.1
      +--- org.hibernate.javax.persistence:hibernate-jpa-2.0-api:1.0.1.Final
      +--- javax.transaction:jta:1.1
      \--- org.slf4j:slf4j-api:1.6.1
```

Transitive  
Abhängigkeiten

# Literatur

