

Podstawy Maven

v. 1.0.0 22-03-2020

(C) Przemysław Kruglej
2020

<https://przemyslawkruglej.com>
<https://craftsmanshipof.software>
<https://kursjava.com>

przemyslaw.kruglej@gmail.com

Spis treści

1 Wstęp.....	3
1.1 Założenia.....	3
1.2 Przykłady oraz wersja HTML i PDF dokumentu.....	3
1.3 Rozdziały.....	3
2 Czym jest Maven i do czego służy?.....	4
2.1 Dlaczego warto poznać Maven?.....	4
3 Instalacja Maven w Windows.....	6
3.1 Sprawdzanie poprawności instalacji Mavena.....	6
4 Pierwsze kroki z Maven.....	8
4.1 Pierwszy projekt.....	9
4.1.1 Wstępna struktura projektu.....	9
4.1.2 Klasa Java w projekcie.....	9
4.1.3 Podstawowa konfiguracja Maven w pom.xml.....	10
4.1.4 Struktura projektu z klasą i pom.xml.....	11
4.1.5 Budujemy projekt.....	11
4.1.6 Efekt mvn install i lokalne repozytorium .m2.....	14
4.1.7 Uruchomienie aplikacji za pomocą pluginu Exec Maven.....	15
4.1.8 Dodawanie zależności do projektu i pierwszy test jednostkowy.....	16
4.1.9 Pomijanie testów.....	18
4.1.10 Gdzie szukać zależności?.....	19
4.1.11 Czyszczenie projektu – mvn clean.....	20
4.2 Generator archetypów.....	21
4.3 Podsumowanie.....	24
4.4 Zadania.....	27
5 Fazy, pluginy, zadania, testy.....	28
5.1 Fazy budowy projektu i pluginy.....	29
5.2 Generowanie pliku JAR z zależnościami.....	32
5.3 Korzystanie ze zmiennych w pliku pom.xml.....	36
5.4 Podłączanie zadania pluginu do fazy budowy projektu.....	37
5.5 Konfiguracja i uruchamianie testów jednostkowych.....	39
5.6 Dodawanie testów integracyjnych do fazy verify.....	42
5.7 Podsumowanie.....	44
5.8 Zadania.....	47
6 Zależności i projekty wielomodułowe.....	48
6.1 Parametr scope zależności.....	49
6.2 Zależności przechodnie (transitive dependencies).....	50
6.3 Zależność do lokalnego projektu.....	52
6.4 Projekty wielomodułowe.....	53
6.4.1 Plugin management i dependency management.....	56
6.5 Efektywny POM.....	60
6.6 Podsumowanie.....	62
7 Podsumowanie Podstaw Maven.....	64
8 Dodatek – przydatne informacje i komendy.....	65
8.1 Informacje.....	65
8.2 Przydatne komendy.....	66

1 Wstęp

Ten artykuł ma na celu zwięzłe przedstawienie podstaw korzystania z Mavena. W artykule zawarłem minimum informacji, które są moim zdaniem wymagane, aby zrozumieć co i jak można osiągnąć za pomocą Mavena.

Będę wdzięczny za wszelkie uwagi i sugestie dotyczące tego artykułu – mój adres kontaktowy to przemyslaw.kruglej@gmail.com.

1.1 Założenia

Zakładam, że znasz podstawy języka Java, gdyż będziemy używali Mavena do pracy z projektami napisanymi w tym języku, oraz podstawy obsługi linii poleceń systemu Windows.

Języka Java możesz nauczyć się z mojego darmowego kursu [Nauka programowania w języku Java od podstaw](#).

Napisałem także artykuł [Podstawy linii poleceń](#) dla użytkowników systemu Windows. Znajdziesz tam informacje o korzystaniu z linii poleceń, przydatne komendy, skróty, i ustawienia, dowiesz się czym są standardowe wejście i wyjście, przekierowanie komend, i wiele więcej.

1.2 Przykłady oraz wersja HTML i PDF dokumentu

Użyte przykłady można znaleźć na Githubie:

<https://github.com/przemyslaw-kruglej/kurs-maven>

Ponadto, w powyższym repozytorium znajdziesz także aktualną wersję PDF tego dokumentu, natomiast wersja HTML dostępna jest pod adresem:

<https://kursjava.com/podstawy-maven>

1.3 Rozdziały

Ten dokument podzielony jest na kilka rozdziałów:

- Najpierw dowiemy się, czym jest Maven i do czego służy.
- Następnie, zainstalujemy i skonfigurujemy Maven w systemie Windows.
- W rozdziale czwartym zaczniemy używać Maven – utworzymy nasz pierwszy projekt i zobaczymy, jaką ma strukturę, a także dodamy do niego zależność i uruchomimy testy jednostkowe.
- W kolejnym rozdziale dowiemy się więcej o fazach budowy projektów w Maven, a także skonfigurujemy dodatkowe pluginy.
- Następnie, dokładniej porozmawiamy o zależnościach i zobaczymy jak skonfigurować projekt wielomodułowy w Maven.
- W ostatnim rozdziale znajdziesz spis przydatnych informacji i komend, które pojawiły się w poprzedzających rozdziałach.

Niektóre rozdziały zakończone są podsumowaniem, w którym znajdziesz skondensowane informacje z danego rozdziału.

2 Czym jest Maven i do czego służy?

Maven to popularne, darmowe narzędzie wspomagające programistów Java. Maven ułatwia:

- dodawanie do projektu zależności do danej biblioteki bądź frameworku,
- kompilowanie i budowanie projektu,
- przeprowadzanie testów jednostkowych i integracyjnych,
- generowanie raportów z testów oraz stron informacyjnych o projekcie,
- i wiele więcej – w zależności od tego, co potrzebujesz.

Maven stosuje zasadę *konwencja ponad konfiguracją* (*convention over configuration*). Dzięki temu większość domyślnych ustawień jest wystarczająca i nie trzeba się nimi przejmować – do rozpoczęcia korzystania z Mavena wymagana jest bardzo mała ilość konfiguracji.

Wszystkie ustawienia Mavena znajdują się w pliku o nazwie `pom.xml` (*POM* – *Project Object Model*). Konfiguracja XML zawarta w tym pliku definiuje m. in.:

- jak nazywa się nasz projekt,
- czy wynikiem zbudowania projektu jest plik JAR czy WAR,
- jakie projekt ma zależności,
- z jakich pluginów Mavena możemy korzystać.

Maven sprawdza w `pom.xml` jakie zależności skonfigurowałeś dla projektu i automatycznie pobiera je z oficjalnego repozytorium. Zostają one umieszczone w Twoim lokalnym repozytorium w katalogu `.m2/repository` w folderze Twojego użytkownika. W ten sposób Maven buduje lokalną bazę bibliotek, z których możesz korzystać. Jest to bardzo wygodne z punktu widzenia programisty – wystarczy podać zależność, a Maven sam odnajdzie i pobierze ją z oficjalnego repozytorium, o ile nie zrobił już tego wcześniej.

Maven umieszcza w lokalnym repozytorium także pliki JAR wygenerowane w ramach budowy Twoich projektów, dzięki czemu możesz korzystać w projektach z zależności do swoich innych projektów.

Mavena używa się z linii poleceń, wywołując komendę `mvn` z odpowiednim parametrem. Środowiska programistyczne, takie jak IntelliJ, wspierają projekty Maven i rozumieją konfigurację zawartą w `pom.xml`. Pozwalają także generować projekty z niego korzystające i uruchamiać jego odpowiednie komendy.

Gdy każesz Mavenowi zbudować Twój projekt, Maven uruchomi szereg zależnych od siebie faz (*build lifecycle phases*), gdzie każda ma inne zadanie do wykonania – faza `compile` kompiluje Twój kod, `test` wykonuje testy jednostkowe, a `install` umieszcza w lokalnym repozytorium zbudowany projekt. Do tych faz podpięte są pluginy, które wykonują związane z nimi zadania – dla przykładu, kompilacją zajmuje się plugin o nazwie *Maven Compiler*. Nie musisz jednak sam konfigurować procesu budowania projektu – większość domyślnych ustawień będzie wystarczająca. Możesz dodać do pliku `pom.xml` zależności do innych pluginów, jeżeli będziesz potrzebował konkretnej funkcjonalności, np. do wygenerowania pliku JAR zawierającego wszystkie zależności skonfigurujesz i użyjesz pluginu *Maven Assembly*.

2.1 Dlaczego warto poznać Maven?

Maven jest na tyle popularny i oferuje dużo równocześnie niewiele wymagając, że istnieje mała szansa, byś nie spotkał Mavena na projekcie, do którego dołączysz (ewentualnie będzie to *Gradle*,

alternatywne narzędzie dające podobną funkcjonalność).

Jeżeli opanujesz podstawy Mavena raz, to będziesz w stanie zbudować każdy korzystający z niego projekt za pomocą:

```
mvn install
```

Struktura katalogów i rozłożenie plików będą znajome – klasy Java znajdziesz w katalogu `src/main/java`, testy w `src/test/java`, wygenerowany JAR w katalogu `target` itd.

Jeśli będziesz chciał dodać zależność do nowej biblioteki, wystarczy, że do pliku konfiguracyjnego `pom.xml` dodasz odpowiedni wpis:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

Wszystkie projekty w Maven konfiguruje się w podobny sposób – jeśli poznasz format pliku konfiguracyjnego Mavena `pom.xml`, to będziesz w stanie się w nim odnaleźć na dowolnym projekcie.

Znajomość Mavena stanowczo zmniejsza próg wejścia w nowy projekt, a sam Maven nie tylko pozwala nam zaoszczędzić sporo czasu, ale także ułatwia nam pracę.

Twoje własne projekty także zyskają, jeżeli będziesz w nich wykorzystywał Mavena, ponieważ samo zarządzanie zależnościami już pozwala zaoszczędzić nam czas i ułatwić pracę, nie wymagając przy tym prawie żadnej konfiguracji.

3 Instalacja Maven w Windows

Aby korzystać z Mavena, musimy:

- pobrać go z oficjalnej strony i rozpakować,
- mieć zainstalowane JDK (Java Development Kit), by mógł z niego korzystać Maven,
- ustawić zmienną `JAVA_HOME` na katalog JDK – zmienna ta jest wymagana przez Maven,
- dodać katalog `bin` z folderu z rozpakowanym Mavenem do zmiennej środowiskowej `path`, aby Maven był dostępny z linii poleceń systemu Windows.

Aktualną wersję Mavena pobierzesz z oficjalnej strony:

<http://maven.apache.org/download.cgi>

Dla systemu Windows klikamy link poprzedzony etykietą *Binary zip archive* – na tą chwilę jest to `apache-maven-3.6.3-bin.zip`.

Po ściągnięciu należy wypakować zawartość archiwum do katalogu na dysku – najlepiej, by nie miał on w nazwie spacji – ja rozpakowałem archiwum do katalogu `C:\apps\apache-maven-3.6.3`

Maven wymaga do pracy zainstalowanego Java Development Kit (JDK) oraz ustawienia zmiennej systemowej `JAVA_HOME`, która powinna wskazywać na lokalizację zainstalowanej dystrybucji JDK.

Aby sprawdzić, czy masz JDK, wykonaj instrukcje opisane na mojej stronie (znajdziesz tam także informację, jak uruchomić linię poleceń w systemie Windows):

<https://kursjava.com/wstep-do-kursu/instalacja-java/#sprawdzeniePoprawnosciInstalacji>

Jeżeli w wyniku sprawdzenia zobaczysz na ekranie komunikat:

```
'javac' is not recognized as an internal or external command, operable program or batch file.
```

będzie to oznaczało, że nie masz zainstalowanego JDK. Przejdź w takim razie na stronę:

<https://kursjava.com/wstep-do-kursu/instalacja-java/#instalacjaJava>

gdzie znajdziesz instrukcję krok po kroku jak zainstalować JDK, a także w jaki sposób modyfikuje się zmienne systemowe Windows.

Po zainstalowaniu JDK, dodaj zmienną systemową o nazwie `JAVA_HOME`, która będzie wskazywała lokalizację zainstalowanego JDK. U mnie zmienna `JAVA_HOME` ma wartość:

```
C:\Program Files\Java\jdk-12
```

Dodatkowo, aby móc korzystać z Mavena z linii poleceń systemu Windows, należy dodać katalog `bin`, znajdujący się w rozpakowanym katalogu z Mavenem, do zmiennej środowiskowej `path`. Jak zmodyfikować tę zmienną znajdziesz na wskazanej powyżej stronie. Ja do zmiennej `path` dodałem taką lokalizację Mavena:

```
C:\apps\apache-maven-3.6.3\bin
```

3.1 Sprawdzanie poprawności instalacji Mavena

Pozostaje nam jeszcze sprawdzić, czy Maven jest dostępny z linii poleceń. Uruchom nowe okno linii poleceń (powinno to być nowe okno, ponieważ zmiany w zmiennych systemowych nie są widoczne w oknach linii poleceń, które zostały otwarte przed wykonaniem tych zmian) i wywołaj

komendę `mvn -version`:

```
> mvn -version
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: C:\apps\apache-maven-3.6.3\bin\..
Java version: 12, vendor: Oracle Corporation, runtime: C:\Program
Files\Java\jdk
-12
Default locale: pl_PL, platform encoding: Cp1250
OS name: "windows 8", version: "6.2", arch: "amd64", family: "windows"
```

Jeżeli zobaczysz na ekranie podobny komunikat do powyższego, to znaczy, że poprawnie zainstalowałeś i skonfigurowałeś Maven. Jeżeli nie, to wróć na początku tego rozdziału i sprawdź, czy ustawiłeś wszystkie zmienne poprawnie.

4 Pierwsze kroki z Maven

W tym rozdziale poznamy podstawowe cechy projektów tworzonych w Mavenie.

Dowiemy się m. in.:

- jak wygląda podstawowa struktura katalogów projektów Maven,
- jak skonfigurować prosty projekt Maven,
- czym są fazy podczas budowy projektu oraz do czego służą pluginy Maven,
- jak zbudować i uruchomić projekt,
- gdzie Maven przechowuje zależności,
- jak dodać zależność do naszego projektu,
- jak zlecić Mavenowi wykonanie testów jednostkowych,
- jak wygenerować szkielet projektu za pomocą generatora archetypów.

4.1 Pierwszy projekt

Na początku naszej przygody z Mavenem zobaczymy na prostym przykładzie jak wygląda struktura projektów Mavena, jak zbudować i uruchomić projekt oraz dodać do niego zależności, a także do czego służy lokalne repozytorium Mavena.

4.1.1 Wstępna struktura projektu

Maven zakłada, że nasze projekty będą miały konkretną strukturę katalogów (*konwencja przed konfiguracją*):

- `${basedir}/src/main/java` – tu umieszczamy nasz kod Java – jeżeli nasza klasa ma być w pakiecie `com.kursjava.maven`, to umieścimy ją w katalogu:
`${basedir}/src/main/java/com/kursjava/maven`
- `${basedir}/src/main/resources` – miejsce na zasoby wymagane przez naszą aplikację (pliki konfiguracyjne itp.),
- `${basedir}/src/test/java` – testy jednostkowe,
- `${basedir}/src/test/resources` – zasoby testów.

`${basedir}` to zmienna w Mavenie, która oznacza główny katalog projektu, w którym zawarte są wszystkie pozostałe pliki i katalogi projektu.

Stwórzmy nasz pierwszy projekt w Maven – będzie on zawierał na razie tylko jedną klasę (bez testów i zasobów), wystarczy więc stworzyć katalogi `src/main/java`. Cały projekt zawrzemy w nadrzędnym katalogu o nazwie `najprostszy-projekt` (`${basedir}` oznacza właśnie ten katalog):

```
najprostszy-projekt
|
|-- src
|   |
|   |-- main
|       |
|       |-- java
```

4.1.2 Klasa Java w projekcie

Do projektu dodamy jedną klasę o nazwie `com.kursjava.maven.HelloMaven`, która wypisze na ekran tekst `"Hello Maven!"`:

`najprostszy-projekt/src/main/java/com/kursjava/maven/HalloMaven.java`

```
package com.kursjava.maven;

public class HelloMaven {
    public static void main(String[] args) {
        System.out.println("Hello Maven!");
    }
}
```

Ponieważ nasza klasa jest w pakiecie `com.kursjava.maven`, musimy utworzyć katalogi odpowiadające pakietowi naszej klasy w przygotowanym wcześniej katalogu `najprostszy-projekt/src/main/java`.

4.1.3 Podstawowa konfiguracja Maven w pom.xml

Ostatnim elementem w tym projekcie jest plik konfiguracyjny Maven – `pom.xml`. Należy utworzyć go w katalogu głównym `najprostszy-projekt`. Treść minimalnego pliku `pom.xml` naszego projektu jest następująca:

najprostszy-projekt/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<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/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.kursjava.maven</groupId>
  <artifactId>hello-maven</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
</project>
```

Analiza powyższego pliku `pom.xml`:

- początek pliku informuje o wersji XML,
- główny element całej konfiguracji to `project`, którego atrybuty informują o przestrzeni nazw tagów używanych w tym pliku XML oraz o lokalizacji schematu XSD, który służy do walidacji formatu plików POM,
- `modelVersion` – oznacza wersję pliku POM – powinna być ustawiana na wartość `4.0.0`,
- `groupId`, `artifactId` oraz `version` to elementy identyfikujące nasz projekt:
 - `groupId` – unikalny identyfikator grupy, do której należy ten projekt – zazwyczaj jest to odwrócona domena autora bądź firmy odpowiedzialnej za projekt, z ewentualnym dodatkiem identyfikującym podgrupę projektów,
 - `artifactId` – nazwa tego konkretnego projektu,
 - `version` – aktualna wersja projektu – wykonując zmiany w naszym projekcie i dokonując release'ów powinniśmy tą wersję aktualizować,
- `properties` – ustawiamy dwa parametry wpływające na kompilację naszego kodu:
 - `maven.compiler.source` – źródła mają być traktowane jako kod Java w wersji 1.8,
 - `maven.compiler.target` – nasze klasy mają być kompilowane do wersji 1.8,
- dodatkowo, ustawiamy kodowanie czytanych i zapisywanych plików na UTF-8 korzystając z parametru `project.build.sourceEncoding`.

Starsza wersja pluginu `maven-compiler-plugin`, który kompiluje nasze klasy (pośrednio lub bezpośrednio korzystając z kompilatora `javac`), domyślnie zakładała wersję naszego

kodu w wersji Java 1.5. W nowszej odsłonie pluginu podniesiono tę wersję do 1.6.

Jeżeli korzystamy z kompilatora Java w wersji 12 lub wyższej, to minimalna obsługiwana wersja kodu Java to 1.7. Dlatego korzystamy z parametrów `maven.compiler.source` i `maven.compile.target`. Jeżeli byśmy tego nie zrobili, to podczas budowania projektu próba kompilacji zakończyłaby się błędami:

```
[ERROR] Source option 6 is no longer supported. Use 5 or later.  
[ERROR] Target option 6 is no longer supported. Use 7 or later.
```

(ewentualnie zamiast "option 6" zobaczylibyśmy "option 5")

4.1.4 Struktura projektu z klasą i pom.xml

Finalnie struktura naszego projektu wygląda następująco:

```
najprostszy-projekt  
|  
|-- pom.xml  
|  
|-- src  
|   |-- main  
|       |-- java  
|           |-- com  
|               |-- kursjava  
|                   |-- maven  
|                       |-- HelloMaven.java
```

4.1.5 Budujemy projekt

Projekt jest gotowy do zbudowania. Przechodzimy do linii komend i wywołujemy komendę `mvn install`:

```
> mvn install  
  
[INFO] Scanning for projects...  
[INFO]  
[INFO] -----< com.kursjava.maven:hello-maven >-----  
[INFO] Building hello-maven 1.0-SNAPSHOT  
[INFO] -----  
[ jar ]-----  
[INFO]  
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ hello-maven ---  
[INFO] Changes detected - recompiling the module!  
[INFO] Compiling 1 source file to D:\kurs_maven\przyklady\najprostszy-projekt\target\classes  
[INFO]  
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ hello-maven ---  
[INFO] No sources to compile
```

```
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello-maven
[INFO] No tests to run.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ hello-maven ---
[INFO] Building jar: D:\kurs_maven\przyklady\najprostszy-projekt\target\hello-maven-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ hello-maven
[INFO] Installing D:\kurs_maven\przyklady\najprostszy-projekt\target\hello-maven-1.0-SNAPSHOT.jar to
C:\Users\Przemek\.m2\repository\com\kursjava\maven\hello-maven\1.0-SNAPSHOT\hello-maven-1.0-SNAPSHOT.jar
[INFO] Installing D:\kurs_maven\przyklady\najprostszy-projekt\pom.xml to
C:\Users\Przemek\.m2\repository\com\kursjava\maven\hello-maven\1.0-SNAPSHOT\hello-maven-1.0-SNAPSHOT.pom
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 4.746 s
[INFO] Finished at: 2020-03-05T12:27:02+01:00
[INFO]
-----
```

Jedna komenda spowodowała szereg akcji, które zostały wykonane przez odpowiednie pluginy. Maven najpierw skompilował klasę `HelloMaven` za pomocą kompilatora `maven-compiler-plugin`:

```
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ hello-maven ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to D:\kurs_maven\przyklady\najprostszy-projekt\target\classes
```

Następnie, Maven próbował skompilować i uruchomić testy jednostkowe – te prace oddelegował do pluginów `maven-compiler-plugin` i `maven-surefire-plugin`. Na razie żadnych testów nie ma w naszym projekcie, więc Maven poinformował nas, że nie ma testów do skompilowania i uruchomienia:

```
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ hello-maven ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello-maven
[INFO] No tests to run.
```

Kolejnym etapem było wygenerowanie pliku JAR z naszą jedyną klasą. Ten JAR nazywamy *artefaktem*. Za to zadanie odpowiedzialny jest plugin `maven-jar-plugin`:

```
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ hello-maven
[INFO] Building jar: D:\kurs_maven\przyklady\najprostszy-projekt\target\hello-maven-1.0-SNAPSHOT.jar
```

Gdy wszystkie pozostałe fazy zakończyły się sukcesem, Maven wykonał ostatnią z nich – `install`. W tej fazie plugin `maven-install-plugin` przekopiował plik JAR, wygenerowany w poprzednim kroku przez plugin `maven-jar-plugin`, do lokalnego repozytorium `.m2/repository`:

```
[INFO] --- maven-install-plugin:2.4:install (default-install) @ hello-maven
[INFO] Installing D:\kurs_maven\przyklady\najprostszy-projekt\target\hello-
maven-1.0-SNAPSHOT.jar to
C:\Users\Przemek\.m2\repository\com\kursjava\mvn\hello-maven\1.0-
SNAPSHOT\hello-maven-1.0-SNAPSHOT.jar
```

Uruchomienie komendy `mvn install` spowodowało, że wykonanych zostało dużo operacji. Wynika to z faktu, że fazy są od siebie zależne – faza `install` wymaga pliku JAR generowanego w fazie `package`. Faza `package`, z kolei, wymaga przeprowadzenia testów w fazie `test`, a faza `test` – skompilowanego w fazie `compile` kodu.

Wszystkie pliki wygenerowane podczas budowania projektu umieszczane są w katalogu o nazwie `target` w katalogu głównym projektu. Katalog `target` zawiera m. in.:

- skompilowane klasy projektu w podkatalogu `classes`,
- listy plików, które brały udział w procesie kompilacji,
- wygenerowany plik JAR.

Dla użytkowników Gita: ponieważ katalog `target` zawiera pliki, które są generowane podczas budowania projektów, zazwyczaj katalog `target` dodawany jest do pliku `.gitignore`, by Git nie śledził zmian w tym katalogu.

Zajrzyj do katalogu `target`, by zaznajomić się z jego zawartością. Zauważ, jaką nazwę Maven nadał plikowi JAR:

```
najprostszy-projekt
|
|-- target
|   |-- hello-maven-1.0-SNAPSHOT.jar
```

Nazwa ta to połączenie wartości elementu `artifactId` i elementu `version` z pliku konfiguracyjnego `pom.xml`:

```
<groupId>com.kursjava.maven</groupId>
<artifactId>hello-maven</artifactId>
<version>1.0-SNAPSHOT</version>
```

Jeżeli pierwszy raz korzystasz z Mavena na Twoim komputerze, to po wykonaniu komendy `mvn install` zobaczysz długą listę komunikatów Mavena dotyczących pobierania pluginów wymaganych przez Maven do pracy:

```
Downloading from central:
https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-
resources-plugin/2.6/maven-resources-plugin-2.6.pom
Downloaded from central:
https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-
resources-plugin/2.6/maven-resources-plugin-2.6.pom (8.1 kB at 11 kB/s)
Downloading from central:
https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-
plugins/23/maven-plugins-23.pom
Downloaded from central:
https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-
plugins/23/maven-plugins-23.pom (9.2 kB at 70 kB/s)
```

Te konkretne komunikaty zobaczysz jednorazowo – Maven zapisze w lokalnym repozytorium wymagane przez niego pliki i będzie je od tej pory używał.

Zawsze, gdy będziesz korzystał z zależności bądź pluginów, których wcześniej nie używałeś (lub innych wersji tych zależności/pluginów), zostaną one jednorazowo pobrane, więc od czasu do czasu będziesz widywał komunikaty podobne do powyższych.

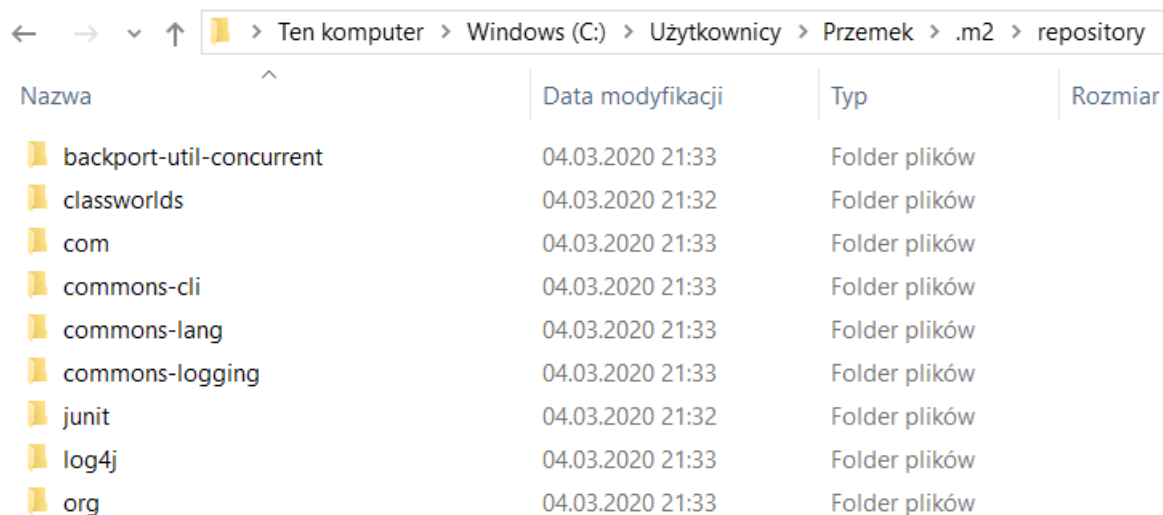
4.1.6 Efekt mvn install i lokalne repozytorium .m2

Gdy w swoim projekcie korzystasz z pluginów Mavena i dodajesz zależności do różnych bibliotek (takich jak JUnit, Spring, Hibernate itp.), Maven pobierze je automatycznie i umieści w Twoim lokalnym *repozytorium artefaktów*.

To repozytorium to katalog o nazwie `.m2` (kropka m2), znajdujący się domyślnie w katalogu Twojego użytkownika.

Maven, pobierając kolejne pluginy i biblioteki, które wykorzystujesz w swoim projekcie, buduje w ten sposób lokalne repozytorium tych zależności. Mogą one być używane pomiędzy wieloma projektami, dzięki czemu, gdy będziesz chciał z nich skorzystać w kolejnym projekcie, będą gotowe do użycia.

Spójrzmy, jak wygląda lokalne repozytorium artefaktów po niedawnej instalacji Mavena i wykonaniu `mvn install` na prostym projekcie z tego rozdziału:



Nazwa	Data modyfikacji	Typ	Rozmiar
backport-util-concurrent	04.03.2020 21:33	Folder plików	
classworlds	04.03.2020 21:32	Folder plików	
com	04.03.2020 21:33	Folder plików	
commons-cli	04.03.2020 21:33	Folder plików	
commons-lang	04.03.2020 21:33	Folder plików	
commons-logging	04.03.2020 21:33	Folder plików	
junit	04.03.2020 21:32	Folder plików	
log4j	04.03.2020 21:33	Folder plików	
org	04.03.2020 21:33	Folder plików	

W katalogu `.m2` znajduje się podkatalog `repository`, do którego Maven pobiera zarówno wymagane przez niego pluginy (jak `maven-compiler-plugin`), jak i zależności naszych projektów. Na razie nasze lokalne repozytorium nie jest zbyt duże, ale będzie się rozrastać, gdy będziemy wymagali do pracy coraz to nowych pluginów i zależności.

Jeżeli zajrzemy do np. `.m2\repository\org\apache\maven\plugins\maven-compiler-plugin\3.1`, to znajdziemy w nim plik `maven-compiler-plugin-3.1.jar`, czyli plugin służący do kompilacji klas, z którego korzysta Maven. W katalogu tym znajdziemy także plik o nazwie `maven-compiler-plugin-3.1.pom`, który jest plikiem POM tego pluginu. [Pluginy to także Javowe projekty tworzone z wykorzystaniem Mavena.](#)

Dzięki temu, że w repozytorium artefaktów przechowywane są nie tylko pliki JAR, ale także pliki POM, Maven jest w stanie sprawdzić, *jake zależności mają nasze zależności*. Często biblioteki, z których chcemy skorzystać, same mają zależności do innych bibliotek itd. – Maven śledzi te

W repozytorium `.m2` znajdują się nie tylko pluginy i zależności naszych projektów, ale także nasze projekty w postaci plików JAR. W poprzednim rozdziale komenda `mvn install` spowodowała przeniesienie wygenerowanego na podstawie naszego projektu pliku JAR do lokalnego repozytorium:

Plik POM naszego prostego projektu także został przeniesiony – jego rozszerzenie zostało zmienione z `.xml` na `.pom`. Zauważmy, że plik JAR został umieszczony w hierarchii katalogów, które są zgodne z polami `groupId`, `artifactId`, oraz `version`, z pliku `pom.xml` naszego projektu:

```
.m2\repository\com\kursjava\maven\hello-maven\1.0-SNAPSHOT
```

Możesz odpytać Maven o lokalizację lokalnego repozytorium `.m2` wykorzystując poniższą komendę:

4.1.7 Uruchomienie aplikacji za pomocą pluginu Exec Maven

```
[INFO] Building hello-maven 1.0-SNAPSHOT
[INFO] -----
[ jar ]-----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ hello-maven ---
Hello Maven!
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 1.136 s
[INFO] Finished at: 2020-03-05T20:00:54+01:00
[INFO]
-----
```

Pośród standardowych informacji wypisywanych przez Maven, widzimy na ekranie wynik wykonania klasy `com.kursjava.maven.HelloMaven`: Hello Maven!

Plugin `exec:java` nie rekompiluje klas naszego projektu – jeżeli zmieniłeś coś w swoim projekcie od ostatniego użycia `exec:java`, to zrekompiluj swój projekt za pomocą `mvn compile`.

4.1.8 Dodawanie zależności do projektu i pierwszy test jednostkowy

Na koniec pracy z naszym pierwszym, najprostszym projektem, dodamy do niego zależność do JUnit i umieścimy w nim jedną klasę z przykładowym testem.

Zależności zawieramy w elementach `<dependency>`, których elementem nadrzędnym jest element `<dependencies>` w `pom.xml`. W elemencie `<dependency>` podajemy odpowiednie `groupId`, `artifactId`, oraz `version`, których znaczenie już znamy. Opcjonalnie możemy także ustawić wartość dla parametru `scope`, który dokładniej mówię w rozdziale o zależnościach.

Gdy Maven będzie budował nasz projekt i zauważy w pliku `pom.xml` element `<dependency>`, to pobierze dla nas automatycznie wymaganą zależność i umieści ją w lokalnym repozytorium `.m2`.

Elementem `scope` możemy poinformować Maven, że dana zależność jest np. tylko wymagana podczas wykonywania testów, a w produkcyjnej wersji naszej aplikacji nie jest w ogóle używana. Tak też jest w tym przypadku – nasz projekt jest zależny od JUnit tylko podczas wykonywania testów – dlatego dodając zależność do JUnit ustawiamy element `scope` na `test`.

Aby nasz projekt mógł korzystać z JUnit, dodajemy do pliku `pom.xml` następujący element:

najprostszy-projekt/pom.xml

```
<!-- poczatek pliku pom.xml zostal pominiety -->

<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
```



```

    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

W przypadku JUnit, zarówno groupId, jak i artifactId, mają taką samą wartość – junit.

Dodamy teraz przykładowy test do projektu. Testy w projektach Mavenowych umieszczamy w katalogu `${basedir}/src/test/java`, więc musimy utworzyć katalogi `test/java`. Klasa z testem będzie w tym samym pakiecie, w którym znajduje się klasa główna projektu `HelloMaven`, więc utworzymy kolejne katalogi: `com/kursjava/maven`. Na koniec dodajemy plik z klasą `HelloMavenTest.java` o treści:

```
najprostszy-projekt/src/test/java/com/kursjava/maven/HelloMavenTest.java
```

```

package com.kursjava.maven;

import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class HelloMavenTest {
    @Test
    public void shouldAnswerWithTrue() {
        assertTrue(true);
    }
}

```

Struktura naszego projektu powinna teraz wyglądać następująco:

```

najprostszy-projekt
|
|-- pom.xml
|
|-- src
|   |-- main
|   |   |-- java
|   |   |   |-- com
|   |   |   |   |-- kursjava
|   |   |   |   |   |-- maven
|   |   |   |   |   |   |-- HelloMaven.java
|   |-- test
|   |   |-- java
|   |   |   |-- com
|   |   |   |   |-- kursjava
|   |   |   |   |   |-- maven
|   |   |   |   |   |   |-- HelloMavenTest.java

```

Testy w Maven uruchamiamy za pomocą komend `mvn test`:

```
> mvn test

... poczatek logów pominiety ...

[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @
hello-maven ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to D:\kurs_maven\przyklady\najprostszy-
projekt\target\test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello-maven
---
[INFO] Surefire report directory: D:\kurs_maven\przyklady\najprostszy-
projekt\target\surefire-reports

-----
T E S T S
-----
Running com.kursjava.maven.HelloMavenTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.039 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 1.893 s
[INFO] Finished at: 2020-03-06T18:47:54+01:00
[INFO]
-----
```

Maven wykrył w naszym projekcie nową klasę – najpierw ją skompilował, a następnie wykonał test korzystając w tym celu z pluginu `maven-surefire-plugin`. Na końcu Maven przedstawił podsumowanie wykonanych testów – jak widać powyżej, wykonany został jeden test, który zakończył się sukcesem.

Standardowo Maven pobiera zależności z oficjalnego repozytorium artefaktów. Wiele firm ma jednak własne repozytoria. Znajdują się w nich zbudowane pliki JAR z projektów tworzonych przez daną firmę i tylko dla niej dostępne. Aby Maven mógł odpytywać takie prywatne repozytorium, należy dodać do katalogu `.m2` plik o nazwie `settings.xml`. Możemy w nim m. in. podać adres prywatnego repozytorium. Więcej informacji znajdziesz w [oficjalnej dokumentacji Maven](#).

4.1.9 Pomijanie testów

Wykonywanie testów jednostkowych jest częścią procesu budowania projektów w Maven. Są one zawsze wykonywane przed wygenerowaniem pliku JAR (bądź WAR).

Czasem może się jednak zdarzyć, że mamy jakieś chwilowo niedziałające testy, a chcemy pomimo tego zbudować projekt – w takim przypadku możemy ustawić parametr `maven.test.skip`, aby nakazać Mavenowi jednorazowe pominięcie wykonania testów:

```

> mvn install -Dmaven.test.skip=true

... poczatek logów pominiety ...

[INFO] Not copying test resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @
hello-maven ---
[INFO] Not compiling test sources
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ hello-maven
---
[INFO] Tests are skipped.
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ hello-maven ---
[INFO] Building jar: D:\kurs_maven\przyklady\najprostszy-
projekt\target\hello-maven-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ hello-maven
[INFO] Installing D:\kurs_maven\przyklady\najprostszy-projekt\target\hello-
maven-1.0-SNAPSHOT.jar to
C:\Users\Przemek\.m2\repository\com\kursjava\maven\hello-maven\1.0-
SNAPSHOT\hello-maven-1.0-SNAPSHOT.jar
[INFO] Installing D:\kurs_maven\przyklady\najprostszy-projekt\pom.xml to
C:\Users\Przemek\.m2\repository\com\kursjava\maven\hello-maven\1.0-
SNAPSHOT\hello-maven-1.0-SNAPSHOT.pom
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 1.699 s
[INFO] Finished at: 2020-03-06T18:59:58+01:00
[INFO]
-----

```

Maven pominął kompilację (*Not compiling test sources*) oraz wykonanie testów (*Tests are skipped*) i przeszedł do wygenerowania pliku JAR i zainstalowania (przekopiowania) go w lokalnym repozytorium .m2.

4.1.10 Gdzie szukać zależności?

Możemy teraz zadać pytanie: skąd mamy wiedzieć, co wpisać w elemencie `<dependency>`, aby Maven pobrał odpowiednią zależność?

Aby znaleźć informację, jakich wartości powinniśmy użyć dla `groupId`, `artifactId`, oraz `version`, możemy zajrzeć na stronę:

<https://repository.sonatype.org>

Znajdziemy tam wyszukiwarkę, w której możemy wpisać szukaną przez nas zależność. Pasujące wyniki będą miały gotowy do skopiowania element `<dependency>`.

Innym sposobem jest po prostu wpisanie w Google nazwy biblioteki, z której chcemy skorzystać, z dodatkiem "maven", np. *log4j maven*.

Dodatkowo, dzięki popularności Mavena, często na oficjalnych stronach różnych projektów możemy znaleźć gotowy do przekopiowania element `<dependency>`. Dla przykładu, na stronie Lomboka (<https://projectlombok.org/setup/maven>) znajdziemy następującą informację:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.12</version>
  <scope>provided</scope>
</dependency>
```

Czasami informacja o projekcie zapisywana jest także jako: `groupId:artifactId:version`.

Zależność opisaną w takim formacie wystarczy otagować w elementy `<dependency>`, `<groupId>`, `<artifactId>`, oraz `<version>`, i dodać do pliku `pom.xml`.

4.1.11 Czyszczenie projektu – mvn clean

Pracując nad projektem dodajemy, usuwamy, oraz przemieszczamy w nim pliki. Zbudowanie projektu powoduje, że skompilowane i wygenerowane pliki umieszczane są w katalogu `target`, co widzieliśmy w jednym z poprzednich rozdziałów.

Jeżeli usuniemy bądź przemieścimy pliki, powinniśmy skorzystać z komendy `mvn clean`, która usuwa katalog `target` z całą jego zawartością. Dzięki temu możemy wykonać "czysty" build projektu – już bez usuniętych plików oraz z uwzględnieniem nowej lokalizacji tych przemieszczonych. Ma to znaczenie, ponieważ pliki te, pozostając w katalogu `target` po poprzednim buildzie, mogłyby w niechciany sposób wpłynąć na nowozbudowany projekt.

Przykład użycia `mvn clean` na naszym projekcie:

```
> mvn clean

[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ hello-maven ---
[INFO] Deleting D:\kurs_maven\przyklady\najprostszy-projekt\target
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 0.294 s
[INFO] Finished at: 2020-03-08T17:57:03+01:00
[INFO]
-----
```

Wykonanie komend Mavena możemy łączyć, pisząc je jedna po drugiej – często stosowana jest np. kombinacja `mvn clean install` – czyszczenie projektu i ponowne jego zbudowanie.

4.2 Generator archetypów

Aby stworzyć nowy projekt Maven nie musimy tworzyć struktury katalogów ręcznie. Zrobiliśmy to w poprzednim rozdziale, aby zaznajomić się z Mavenem.

Maven udostępnia plugin nazywany *generatorem archetypów*, który tworzy strukturę i szkielet projektu wybranego przez nas typu. Na moment pisania tego dokumentu dostępnych jest 2590 archetypów. My skorzystamy z *maven-archetype-quickstart*, który tworzy najprostszy projekt wykorzystujący Maven.

Gdy uruchamiamy komendę `mvn archetype:generate`, możemy podać wszystkie wymagane parametry, by projekt został wygenerowany od razu, lub nie podać ich, przez co Maven przejdzie w tryb interaktywnego tworzenia projektu.

W interaktywnym trybie Maven zapyta nas o kilka informacji, takich jak: `groupId`, `artifactId`, `version`, oraz numer archetypu, który chcemy wygenerować. Możemy zawęzić listę wyświetlanych archetypów wpisując część nazwy poszukiwanego rodzaju projektu (np. `spring`, aby wyświetlić archetypy związane ze Springiem).

Poniżej znajduje się przykład użycia generatora archetypów w trybie interaktywnym:

```
> mvn archetype:generate

[INFO] >>> maven-archetype-plugin:3.1.2:generate (default-cli) > generate-
sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:3.1.2:generate (default-cli) < generate-
sources @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:3.1.2:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart
(org.apache.maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:
1: remote -> am.ik.archetype:elm-spring-boot-blank-archetype (Blank multi
project for Spring Boot + Elm)
2: remote -> am.ik.archetype:graalvm-blank-archetype (Blank project for
GraalVM)

(... lista archetypów pominięta ...)

2589: remote -> xyz.lua.generator:xyz-generator (-)
2590: remote -> za.co.absa.hyperdrive:component-archetype (-)
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 1497: 1497
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:
1: 1.0-alpha-1
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
7: 1.3
8: 1.4
Choose a number: 8: 8
Define value for property 'groupId': com.kursjava.maven
Define value for property 'artifactId': wygenerowany-projekt
```

```

Define value for property 'version' 1.0-SNAPSHOT: :
Define value for property 'package' com.kursjava.maven: :
Confirm properties configuration:
groupId: com.kursjava.maven
artifactId: wygenerowany-projekt
version: 1.0-SNAPSHOT
package: com.kursjava.maven
Y: :
[INFO]
-----
[INFO] Using following parameters for creating project from Archetype:
maven-arc
hetype-quickstart:1.4
[INFO]
-----
[INFO] Parameter: groupId, Value: com.kursjava.maven
[INFO] Parameter: artifactId, Value: wygenerowany-projekt
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.kursjava.maven
[INFO] Parameter: packageInPathFormat, Value: com/kursjava/maven
[INFO] Parameter: package, Value: com.kursjava.maven
[INFO] Parameter: groupId, Value: com.kursjava.maven
[INFO] Parameter: artifactId, Value: wygenerowany-projekt
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Project created from Archetype in dir:
D:\kurs_maven\przyklady\wygenerowany-projekt
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 02:28 min
[INFO] Finished at: 2020-03-09T19:14:00+01:00
[INFO]
-----

```

Ponieważ nie podałem parametru z numerem wskazującym na konkretny archetyp, Maven domyślnie proponował użycie archetypu `maven-archetype-quickstart` o numerze 1497. Zaznaczone powyżej na białym tle informacje to wartości podane przeze mnie – Maven potrzebował ich, aby wygenerować projekt. W niektórych miejscach zamiast wpisać wymaganą wartość nacisnąłem **Enter**, co spowodowało, że Maven skorzystał z proponowanej przez siebie domyślnej wartości.

W wyniku tej komendy wygenerowany został projekt o następującej strukturze:

```

wygenerowany-projekt
|
|-- pom.xml
|
|-- src
|   |-- main
|   |   |-- java
|   |   |   |-- com
|   |   |   |   |-- kursjava
|   |   |   |   |   |-- maven
|   |   |   |   |   |   |-- App.java
|   |-- test
|   |   |-- java
|   |   |   |-- com
|   |   |   |   |-- kursjava
|   |   |   |   |   |-- maven
|   |   |   |   |   |   |-- AppTest.java

```

Projekt ma identyczną strukturę jak projekt utworzony przez nas w poprzednim rozdziale. Maven utworzył w nim klasę `App`, która wypisuje na ekran tekst `"Hello World!"` oraz klasę `AppTest` z jednym testem. W pliku `pom.xml` zawarte są informacje o naszym projekcie w polach `groupId`, `artifactId`, oraz `version`, a także zależność do JUnit. Projekt może od razu zostać zbudowany za pomocą `mvn install`.

Korzystanie z generatora archetypów to szybki sposób na stworzenie szkieletu projektu Mavenowego.

Użyta wcześniej komendę `mvn archetype:generate` moglibyśmy także użyć w trybie nieinteraktywnym gdybyśmy podali wymagane wartości jako parametry:

```

mvn archetype:generate -B -DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-quickstart
-DgroupId=com.kursjava.maven -DartifactId=wygenerowany-projekt
-Dversion=1.0-SNAPSHOT -Dpackage=com.kursjava.maven

```

- `-B` (*batch mode*) – to parametr, dzięki któremu komenda ma wykonać się w trybie nieinteraktywnym,
- `archetypeGroupId` i `archetypeArtifactId` – opisują, który archetyp chcemy użyć – moglibyśmy jeszcze podać wersję za pomocą `archetypeVersion`, ale pomijając ten parametr użyta zostanie najnowsza wersja archetypu,
- `groupId`, `artifactId`, `version` – to znane nam już atrybuty opisujące nasz projekt,
- `package` – pakiet, w którym mają znaleźć się nasze klasy.

4.3 Podsumowanie

W tym rozdziale poznaliśmy podstawy pracy z Mavenem:

- Maven stosuje zasadę *konwencja przed konfiguracją* (*convention over configuration*) – aby Maven działał, wymagana jest minimalna konfiguracja – większość parametrów projektu ma domyślne wartości.
- Projekty korzystające z Maven powinny mieć odpowiednią strukturę:
 - `${basedir}/src/main/java` – tutaj umieszczamy kod Java,
 - `${basedir}/src/test/java` – tutaj umieszczamy testy jednostkowe.
- Jeżeli klasa ma być w pakiecie `com.kursjava.maven`, to powinniśmy umieścić ją w katalogu `${basedir}/src/main/java/com/kursjava/maven`
- `${basedir}` to zmienna w Mavenie która oznacza główny katalog projektu, w którym zawarte są wszystkie pozostałe pliki i katalogi projektu.
- Konfigurację Maven umieszczamy w pliku `pom.xml`, który powinien znajdować się w katalogu głównym projektu. Przykładowy, prosty plik `pom.xml`:

```
<?xml version="1.0" encoding="UTF-8" ?>

<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/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.kursjava.maven</groupId>
  <artifactId>hello-maven</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
</project>
```

- Powyższy plik `pom.xml` składa się z następujących elementów:
 - `modelVersion` – wersja pliku POM – powinna być ustawiana na wartość `4.0.0`,
 - `groupId`, `artifactId` oraz `version` to elementy identyfikujące nasz projekt:
 - `groupId` – unikalny identyfikator grupy, do której należy ten projekt – zazwyczaj jest to odwrócona domena autora bądź firmy odpowiedzialnej za projekt, z ewentualnym dodatkiem identyfikującym podgrupę projektów,
 - `artifactId` – nazwa tego konkretnego projektu,
 - `version` – aktualna wersja projektu – wykonując zmiany w naszym projekcie i dokonując release'ów powinniśmy tą wersję aktualizować.
 - `properties` – parametry wpływające na kompilację kodu:

- `maven.compiler.source` – źródła mają być traktowane jako kod Java w wersji 1.8,
 - `maven.compiler.target` – nasze klasy mają być kompilowane do wersji 1.8,
 - `project.build.sourceEncoding` – kodowanie czytanych i zapisywanych plików to UTF-8.
- Aby zbudować projekt w Maven, korzystamy z komendy `mvn install`.
 - Proces budowy projektu składa się z faz (*build lifecycle phases*), które są wykonywane jedna po drugiej w celu zbudowanie projektu.
 - Fazy Mavena mają przypisane pluginy, który wykonują prace związane z daną fazą. Dla przykładu, kompilacją zajmuje się plugin `maven-compiler-plugin`, uruchamianiem testów `maven-surefire-plugin`, a generacją pliku JAR `maven-jar-plugin`.
 - Uruchomienie komendy `mvn install` powoduje wykonanie wielu operacji, ponieważ fazy budowania projektu są od siebie zależne – faza `install` wymaga pliku JAR generowanego w fazie `package`. Faza `package` wymaga przetestowania klas w fazie `test`, a faza `test` – skompilowanego w fazie `compile` kodu.
 - Wszystkie pliki wygenerowane podczas budowania projektu, w tym plik JAR, umieszczane są w katalogu o nazwie `target` w katalogu głównym projektu.
 - Maven generuje w katalogu `target` plik JAR o nazwie złożonej z połączonych wartości `artifactId` oraz `version`, które konfigurujemy w pliku `pom.xml`. Dla następujących wartości:

```
<groupId>com.kursjava.maven</groupId>
<artifactId>hello-maven</artifactId>
<version>1.0-SNAPSHOT</version>
```

Maven wygeneruje plik `target/hello-maven-1.0-SNAPSHOT.jar`

- Maven przechowuje na dysku lokalne repozytorium artefaktów w katalogu `.m2/repository` użytkownika (np. `C:\Users\Przemek\.m2`). Znajdują się w nim:
 - pluginy, z których korzysta Maven,
 - zależności, z których korzystają nasze projekty,
 - plik JAR z naszymi projektami zbudowanymi za pomocą Mavena – dla przykładu, JAR z opisanego powyżej projektu `hello-maven` znajdzie się w lokalizacji opisanej na poniższym rysunku:

groupId artifactId version
`.m2\repository\com\kursjava\maven\hello-maven\1.0-SNAPSHOT`

- Maven automatycznie pobiera i umieszcza w lokalnym repozytorium `.m2` pliki JAR bibliotek, których wymagają nasze projekty (np. JUnit, Spring, Hibernate itp.).
- Jeżeli na projekcie używane jest prywatne repozytorium artefaktów, z którego Maven powinien pobierać zależności, to jego adres możemy ustawić w pliku `settings.xml` w katalogu `.m2`.
- Aby sprawdzić lokalizację lokalnego repozytorium `.m2` możesz użyć komendy:

```
mvn help:evaluate -Dexpression=settings.localRepository
```

- Aby uruchomić aplikację, możemy skorzystać z pluginu Exec Maven, który ustawi odpowiednio `classpath` (pamiętaj o rekompilacji za pomocą `mvn compile`, jeżeli wykonałeś zmiany od ostatniego uruchomienia projektu):

```
mvn exec:java -Dexec.mainClass=com.kursjava.maven.HelloMaven
```

- Zależności w projektach Mavenowych dodajemy do elementu `<dependencies>` w pliku `pom.xml` podając odpowiednią kombinacją wartości `groupId`, `artifactId`, oraz `version`:

```
<!-- poczatek pliku pom.xml zostal pominiety -->

<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

- Aby dowiedzieć się, co wpisać w elemencie `<dependency>`, możemy:
 - poszukać odpowiedzi na Google,
 - użyć wyszukiwarki na stronie <https://repository.sonatype.org>,
 - znaleźć na oficjalnej stronie danej biblioteki zależność zapisaną w formacie `groupId:artifactId:version` lub przekopiować element `<dependency>`, jeżeli jest dostępny.
- Wykonanie testów przez Maven osiągamy za pomocą komendy `mvn test`
- Aby pominąć wykonanie testów należy ustawić parametr `maven.test.skip`:

```
mvn install -Dmaven.test.skip=true
```

- Aby wyczyścić pliki wygenerowane przez Maven w ramach budowania projektu korzystamy z komendy `mvn clean`
- Aby wywołać więcej niż jedną komendę w Maven możemy zapisać je jedna po drugiej: `mvn clean install`
- Generator archetypów to plugin w Maven pozwalający na wygenerowanie szkieletu projekt. Do wyboru jest wiele różnych archetypów. Archetypy można generować w trybie interaktywnym lub podać wszystkie wartości od razu.
- Przykład wygenerowanie prostego archetypu w trybie nieinteraktywnym:

```
mvn archetype:generate -B -DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-quickstart
-DgroupId=com.kursjava.maven -DartifactId=wygenerowany-projekt
-Dversion=1.0-SNAPSHOT -Dpackage=com.kursjava.maven
```

4.4 Zadania

Wygeneruj za pomocą generatora archetypów prosty projekt Mavenowy, a następnie:

- w głównej klasie dodaj metodę `kwadrat`, która będzie zwracała kwadrat liczby podanej jako argument; nie usuwaj metody `main`,
- w klasie z testami jednostkowymi dodaj kilka testów metody `kwadrat`,
- zbuduj projekt za pomocą `mvn install` i sprawdź, czy wszystkie testy wykonały się bez błędów,
- sprawdź, czy plik JAR wygenerowany przez Maven został przeniesiony do Twojego lokalnego repozytorium `.m2`,
- usuń pliki wygenerowane podczas budowania projektu za pomocą komendy `mvn clean`,
- ponownie zbuduj projekt, tym razem pomijając fazę testów,
- uruchom swój program:
 - przy pomocy pluginu Exec Maven,
 - w klasyczny sposób korzystając z `java` w linii komend.

5 Fazy, pluginy, zadania, testy

W tym rozdziale:

- dowiemy się więcej o fazach budowy projektów Mavenowych,
- zobaczymy jak konfigurować pluginy i korzystać z ich zadań,
- użyjemy pluginu Assembly do wygenerowania pliku JAR zawierającego wszystkie zależności,
- nauczymy się jak konfigurować plugin do wykonywania testów,
- dodamy do projektu możliwość uruchamiania testów integracyjnych.

5.1 Fazy budowy projektu i pluginy

Maven buduje nasze projekty poprzez wykonanie szeregu zależnych od siebie faz (*build lifecycle phases*). Każda faza odpowiedzialna jest za wykonanie określonego zadania.

Kilka z tych fazy widzieliśmy w poprzednim rozdziale – były to m. in. fazy:

- `compile`, podczas której źródła projektu są kompilowane,
- `test`, w której wykonywane są testy,
- `package`, której wynikiem jest np. plik JAR lub WAR,
- `install`, dzięki której plik wygenerowany w fazie `package` jest przenoszony do lokalnego repozytorium `.m2`, i staje się dostępny z poziomu innych naszych projektów.

Ponieważ fazy są od siebie zależne, uruchomienie np. fazy `test` w Maven za pomocą komendy:

```
mvn test
```

powoduje wykonanie przez Maven wszystkich poprzednich w kolejności faz. Jeżeli zakończą się one sukcesem, na końcu zostanie wykonana zlecona przez nas faza `test`.

Listę wszystkich faz możemy otrzymać korzystając z pluginu Help Mavena:

```
> mvn help:describe -Dcmd=install

[INFO] 'install' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-install-plugin:2.4:install

It is a part of the lifecycle for the POM packaging 'jar'. This lifecycle
includes the following phases:
* validate: Not defined
* initialize: Not defined
* generate-sources: Not defined
* process-sources: Not defined
* generate-resources: Not defined
* process-resources: org.apache.maven.plugins:maven-resources-
plugin:2.6:resources
* compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
* process-classes: Not defined
* generate-test-sources: Not defined
* process-test-sources: Not defined
* generate-test-resources: Not defined
* process-test-resources: org.apache.maven.plugins:maven-resources-
plugin:2.6:testResources
* test-compile: org.apache.maven.plugins:maven-compiler-
plugin:3.1:testCompile
* process-test-classes: Not defined
* test: org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
* prepare-package: Not defined
* package: org.apache.maven.plugins:maven-jar-plugin:2.4:jar
* pre-integration-test: Not defined
* integration-test: Not defined
* post-integration-test: Not defined
* verify: Not defined
* install: org.apache.maven.plugins:maven-install-plugin:2.4:install
* deploy: org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy
```

Każda faza zależna jest od faz poprzednich, więc aby wykonać fazę `deploy` Maven musi najpierw

przejsć przez ponad 20 wcześniejszych faz.

Jak widać na powyższym listingu, niektóre z faz mają przyporządkowane informacje o pluginach. Dla przykładu, do fazy `test` przypisany jest plugin Surefire:

```
org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
```

Pierwsze trzy człony informacji o tym pluginie to poznane już `groupId`, `artifactId`, oraz `version` – pluginy to także projekty Mavenowe. Na samym końcu opisu tego pluginu znajduje się *zadanie* tego pluginu (*plugin goal*), które ma zostać wykonane przez Maven w trakcie fazy `test`. W tym przypadku zadanie pluginu, jak i faza budowania projektu, z którą to zadanie jest skojarzone, mają taką samą nazwę – `test`.

Pluginy to po prostu projekty Maven mające do wykonania konkretną pracę jak generowanie pliku JAR czy kompilacja klas Java. Pobierane są z oficjalnego repozytorium Maven w postaci plików JAR i umieszczane w lokalnym repozytorium `.m2`. Niektóre pluginy są od razu dostępne do wykorzystania w Mavenie, np. plugin `Compile` lub `Surefire`. Inne, jak plugin `Assembly`, z którego będziemy korzystać w następnym rozdziale, trzeba najpierw skonfigurować w `pom.xml`. Możemy także pisać własne pluginy do Mavena wedle naszych potrzeb. Więcej informacji o własnych pluginach znajdziesz w [oficjalnej dokumentacji](#).

To, jakie zadania udostępniają pluginy, zależy od twórców tych pluginów. Dla przykładu, do tej pory korzystaliśmy z zadań `describe` (w tym rozdziale) oraz `evaluate` (w poprzednim rozdziale) pluginu `Help Mavena`:

```
mvn help:describe -Dcmd=install
mvn help:evaluate -Dexpression=settings.localRepository
```

Listę zadań danego pluginu możemy sprawdzić zaglądając do oficjalnej dokumentacji. Innym sposobem, aby zapoznać się z możliwościami pewnego pluginu, jest użycie pluginu `Help Mavena`.

Plugin `Help` ma zadanie `describe`, któremu możemy przekazywać różne parametry. Jednym z nich jest `plugin`. Jeżeli z niego skorzystamy, `Help` zwróci informację o podanym przez nas pluginie:

```
> mvn help:describe -Dplugin=org.apache.maven.plugins:maven-surefire-plugin

[INFO] org.apache.maven.plugins:maven-surefire-plugin:2.12.4

Name: Maven Surefire Plugin
Description: Surefire is a test framework project.
Group Id: org.apache.maven.plugins
Artifact Id: maven-surefire-plugin
Version: 2.12.4
Goal Prefix: surefire

This plugin has 2 goals:

surefire:help
  Description: Display help information on maven-surefire-plugin.
  Call mvn surefire:help -Ddetail=true -Dgoal=<goal-name> to display
  parameter details.

surefire:test
  Description: Run tests using Surefire.

For more information, run 'mvn help:describe [...] -Ddetail'
```

Powyżej kazaliśmy pluginowi `help` wykonać zadanie `describe` (`help:describe`), przekazując parametr `plugin` o wartości `org.apache.maven.plugins:maven-surefire-plugin`

W zwróconych informacjach widzimy szczegółowe dane pluginu, o który zapytaliśmy, czyli pluginu `Surefire`. Widzimy, że udostępnia on ma dwa zadania: `test` oraz `help`.

W poprzednim rozdziale korzystaliśmy kilka razy z pluginów uruchamiając je wraz z informacją, które z udostępnianych zadań miały wykonać. Przykładowo, pluginowi `Archetype` kazaliśmy wykonać zadanie `generate`, a pluginowi `Exec` – zadanie `java`.

5.2 Generowanie pliku JAR z zależnościami

Nasze projekty zazwyczaj będą miały wiele zależności do innych projektów. Aby uruchomić nasz program będziemy musieli ustawić w `classpath` ścieżki do wymaganych JARów. Możemy zamiast tego tak skonfigurować Maven, aby był w stanie wygenerować jeden duży JAR, który będzie zawierał klasy naszego projektu oraz wszystkie jego zależności. W tym celu skorzystamy z pluginu Maven o nazwie *Assembly*.

W tym rozdziale utworzymy nowy projekt za pomocą generatora archetypów Maven – ponownie będzie to archetyp `maven-archetype-quickstart`. Użyjemy trybu nieinteraktywnego, przekazując wszystkie potrzebne wartości jako argumenty:

```
mvn archetype:generate -B -DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-quickstart
-DgroupId=com.kursjava.maven -DartifactId=policz-silnie -Dversion=1.0-
SNAPSHOT -Dpackage=com.kursjava.maven
```

Po wygenerowaniu projektu dodamy do pliku `pom.xml` zależność do Log4j, aby później przetestować, czy plik JAR wygenerowany przez plugin Assembly będzie zawierał zależność do tej biblioteki. Poniższe wpisy dodajemy do elementu `<dependencies>`:

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.13.1</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.13.1</version>
</dependency>
```

Wygenerowaną w katalogu `src/main/java/com/kursjava/maven` klasę `App.java` zastąpimy plikiem `FactorialCounter.java` o następującej treści:

`policz-silnie/src/main/java/com/kursjava/maven/FactorialCounter.java`

```
package com.kursjava.maven;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class FactorialCounter {
    private static final Logger log =
        LogManager.getLogger(FactorialCounter.class);

    public static void main(String[] args) {
        System.out.println("Przykład liczenia silni.");
        System.out.println("Silnia 5 = " + factorial(5));
    }

    public static int factorial(int n) {
        if (n < 0) {
            log.error("Nieprawidlowa wartosc n: {}", n);
            throw new IllegalArgumentException(
                "Silnia moze byc liczona tylko dla n >= 0"
            );
        }
    }
}
```



```

    int result = 1;

    for (int i = 2; i <= n; i++) {
        result *= i;
    }

    return result;
}
}

```

Ta prosta klasa zawiera metodę liczącą silnię podanej liczby. Korzysta ona z Log4j do ewentualnego zalogowania błędnego argumentu.

Do projektu dodamy jeszcze jeden zasób – konfigurację Log4j. Nie musimy tego robić, ponieważ Log4j mógłby użyć swojej domyślnej konfiguracji, ale jest to dobry przykład, by zobaczyć gdzie w projektach Mavenowych umieszcza się tego rodzaju zasoby.

Poniższy plik `log4j2.xml` umieszczamy w katalogu `src/main/resources` projektu:

`policz-silnie/src/main/resources/log4j2.xml`

```

<?xml version="1.0" encoding="UTF-8"?>

<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss} %-5level %logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="error">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>

```

Jeżeli zbudujemy teraz nasz projekt i spróbujemy uruchomić go korzystając ze standardowo wygenerowanego pliku JAR w katalogu `target`, to zobaczymy następujący komunikat błędu:

```

> java -cp target/policz-silnie-1.0-SNAPSHOT.jar com.kursjava.maven.FactorialCounter

Exception in thread "main" java.lang.NoClassDefFoundError:
org/apache/logging/log4j/LogManager
    at com.kursjava.maven.FactorialCounter.<clinit>(FactorialCounter.java:8)
Caused by: java.lang.ClassNotFoundException: org.apache.logging.log4j.LogManager

```

Powodem komunikatu jest brak biblioteki Log4j w `classpath`, od której zależna jest nasza klasa z pliku `FactorialCounter.java`.

Moglibyśmy JAR z Log4j umieścić w `classpath`, ale zamiast tego skorzystamy z pluginu Assembly do wygenerowania JARa zawierającego naszą klasę wraz ze wszystkimi zależnościami.

Zanim będziemy mogli skorzystać z pluginu Assembly, musimy skonfigurować go w pliku `pom.xml` projektu. Konfigurację pluginów umieszcza się w elementach `<plugin>`, dla których elementami nadrzędnymi są elementy `<plugins>` i `<build>`.

Aby móc korzystać z pluginu Assembly, wystarczy dodać poniższy element do pliku `pom.xml`:

```

<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>3.2.0</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </plugin>
  </plugins>
</build>

```

Plik `pom.xml` projektu wygenerowanego za pomocą archetypu `maven-archetype-quickstart` zawiera już kilka konfiguracji pluginów, na przykład `maven-compiler-plugin`. W tej konfiguracji ustawiane są konkretne wersje tych pluginów, by nie były używane ich domyślne wersje zdefiniowane w Mavenie, ponieważ te domyślne wersje mogłyby się zmienić z czasem. Dodatkowo, konfiguracja tych pluginów zawarta jest w jeszcze jednym elemencie – `<pluginManagement>`, o którym opowiemy sobie w rozdziale o projektach wielomodułowych.

Możemy teraz skorzystać z zadania `single` pluginu Assembly, którego celem jest wygenerowanie jednego JARa zawierającego klasy naszego projektu i ich zależności. W pierwszej kolejności musimy skompilować nasz projekt. Możemy połączyć obie komendy rozdzielając je spacją:

```

> mvn compile assembly:single

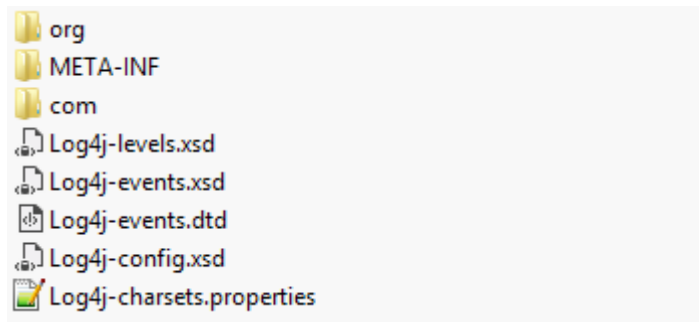
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.kursjava.maven:policz-silnie >-----
[INFO] Building policz-silnie 1.0-SNAPSHOT
[INFO] (...)
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ policz-silnie
[INFO] (...)
[INFO] --- maven-assembly-plugin:3.2.0:single (default-cli) @ policz-silnie
[INFO] ---
[INFO] Building jar: D:\kurs_maven\przyklady\policz-silnie\target\policz-silnie-1.0-SNAPSHOT-jar-with-dependencies.jar
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 2.149 s
[INFO] Finished at: 2020-03-12T19:53:18+01:00
[INFO]

```

W wyniku działania pluginu Assembly wygenerowany został następujący plik w katalogu `target`: `policz-silnie-1.0-SNAPSHOT-jar-with-dependencies.jar`

Ten plik zawiera zarówno skompilowaną klasę naszego projektu, jak i wszystkie zależności wymagane w trakcie jego działania – w naszym przykładzie jest to biblioteka Log4j.

Jeżeli otworzymy ten plik JAR, to zobaczymy następujące pliki i katalogi:



W katalogu `com` znajduje się skompilowana klasa naszego projektu, a wszystkie pozostałe katalogi i pliki należą do Log4j.

Możemy ponownie spróbować uruchomić główną klasę naszego projektu, tym razem korzystając jednak z nowego pliku JAR:

```
> java -cp target/policz-silnie-1.0-SNAPSHOT-jar-with-dependencies.jar  
com.kursjava.maven.FactorialCounter
```

```
Przykład liczenia silni.  
Silnia 5 = 120
```

Tym razem udało nam się uruchomić naszą klasę. Wszystkie zależności naszego projektu (aż jedna – Log4j) zawarte są w pliku JAR `policz-silnie-1.0-SNAPSHOT-jar-with-dependencies.jar` wygenerowanym przez plugin Assembly.

5.3 Korzystanie ze zmiennych w pliku pom.xml

W plikach pom.xml możemy definiować nazwane wartości, z których możemy potem korzystać w pliku pom.xml. Dla przykładu, w poprzednim rozdziale dodaliśmy zależność do biblioteki Log4j – wymagało to użycia dwóch następujących wpisów:

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.13.1</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.13.1</version>
</dependency>
```

Wersja obu zależności jest taka sama – zamiast wpisywać ją na sztywno w obu elementach `<version>`, możemy utworzyć *property* z tą wartością i użyć jej w elementach `<dependency>`.

W plik pom.xml wygenerowanego projektu jest już kilka nazwanych wartości:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
</properties>
```

Dodamy do nich nową wartość o nazwie log4j.version:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
  <log4j.version>2.13.1</log4j.version>
</properties>
```

Aby wskazać Mavenowi, że ma skorzystać z pewnej nazwanej wartości, stosujemy składnię `${nazwa}` (znak dolara, klamra, nazwa, klamra), co widać na poniższym listingu:

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>${log4j.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>${log4j.version}</version>
</dependency>
```

Dzięki zastąpieniu wpisanych na sztywno wersji tych zależności możemy w łatwy sposób zmienić wymaganą wersję zmieniając ją tylko w jednym miejscu naszego pliku pom.xml – w elemencie `<log4j.version>` zdefiniowanym w `<properties>`.

5.4 Podłączanie zadania pluginu do fazy budowy projektu

Na początku rozdziału o pluginach zobaczyliśmy, że budowa projektu składa się z wielu faz, ale tylko niektóre z nich mają domyślnie przypisane zadanie pewnego pluginu, które ma zostać przez ten plugin wykonane.

Dla przypomnienia, spójrzmy na fragment listy faz budowy projektu i przypisanych do nich zadań pluginów:

```
(...)  
* package: org.apache.maven.plugins:maven-jar-plugin:2.4:jar  
* pre-integration-test: Not defined  
* integration-test: Not defined  
* post-integration-test: Not defined  
* verify: Not defined  
* install: org.apache.maven.plugins:maven-install-plugin:2.4:install  
* deploy: org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy
```

Faza `verify` nie ma powiązanego zadania żadnego pluginu, a podczas fazy `package` wykonywane jest zadanie `jar` pluginu `maven-jar-plugin`.

W pliku `pom.xml` możemy skonfigurować pluginy w taki sposób, by jedno z ich dostępnych zadań zostało wykonane automatycznie, gdy Maven będzie wykonywał pewną fazę budowy projektu.

Wcześniej w tym rozdziale skorzystaliśmy z pluginu `Assembly` do zbudowania jednego, dużego pliku JAR naszego projektu, zawierającego wszystkie zależności. Możemy dodać do naszego pliku `pom.xml` konfigurację, dzięki której plugin `Assembly` będzie wykonywał swoje zadanie zawsze podczas np. fazy `package`. Dzięki temu zawsze budując nasz projekt będziemy dodatkowo otrzymywali w katalogu `target` plik JAR z zależnościami, bez potrzeby ręcznego wywoływania pluginu `Assembly` z linii poleceń.

Aby dodać użycie `Assembly` do fazy `package`, uzupełniamy konfigurację tego pluginu o element `<executions>`, w którym definiujemy fazę, podczas której plugin ma zostać użyty, oraz które z jego zadań ma wtedy zostać wykonane. Konfiguracja pluginu `Assembly` będzie więc w naszym projekcie wyglądać następująco:

```
<build>  
  <plugins>  
    <plugin>  
      <artifactId>maven-assembly-plugin</artifactId>  
      <version>3.2.0</version>  
      <configuration>  
        <descriptorRefs>  
          <descriptorRef>jar-with-dependencies</descriptorRef>  
        </descriptorRefs>  
      </configuration>  
      <executions>  
        <execution>  
          <phase>package</phase>  
          <goals>  
            <goal>single</goal>  
          </goals>  
        </execution>  
      </executions>  
    </plugin>  
  </plugins>  
</build>
```

W elemencie `<phase>` podajemy nazwę fazy, podczas której plugin ma zostać użyty, a w elemencie `<goal>` – zadanie, które plugin ma wtedy wykonać.

Jeżeli zbudujemy teraz od nowa projekt, to zobaczymy, że w katalogu `target`, poza plikiem JAR standardowo generowanym w fazie `package`, znajduje się także drugi plik JAR – ten wygenerowany przez podłączony przez nas do fazy `package` plugin Assembly:

```
> mvn clean install

[INFO] Building policz-silnie 1.0-SNAPSHOT
(...)
[INFO] --- maven-compiler-plugin:3.8.0:compile (default-compile) @ policz-silnie ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to D:\kurs_maven\przyklady\policz-silnie\target\classes
[INFO]
(...)
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ policz-silnie ---
[INFO] Building jar: D:\kurs_maven\przyklady\policz-silnie\target\policz-silnie-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-assembly-plugin:3.2.0:single (assemble-jar-with-dependencies) @ policz-silnie ---
[INFO] Building jar: D:\kurs_maven\przyklady\policz-silnie\target\policz-silnie-1.0-SNAPSHOT-jar-with-dependencies.jar
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ policz-silnie ---
[INFO]
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO]
[INFO] Total time: 5.086 s
[INFO] Finished at: 2020-03-14T13:46:49+01:00
[INFO]
```

Maven w fazie `package` poza standardowym wykonaniem zadania `jar` pluginu `maven-jar-plugin`, wykonał także zadanie `single` pluginu `maven-assembly-plugin`, co zostało zaznaczone na powyższym listingu.

Zauważ, że powyżej skorzystaliśmy z `mvn install`, a plugin Assembly podpięliśmy do fazy `package`. Plugin i tak został użyty, ponieważ wykonanie fazy `package` jest jedną z czynności poprzedzających wykonanie fazy `install`.

5.5 Konfiguracja i uruchamianie testów jednostkowych

W fazie `test`, Maven domyślnie wykonuje testy zawarte w plikach projektu, których nazwa pasuje do któregoś z poniższych wzorców (`**` oznacza dowolny katalog projektu):

- `**/Test*.java`
- `**/*Test.java`
- `**/*Tests.java`
- `**/*TestCase.java`

Testy powinniśmy umieszczać w katalogu `src/test/java`. Jeżeli mamy potrzebę korzystać z pliku z testami, który nie pasuje po powyższych wzorców, to możemy skonfigurować plugin `Surfire`, którego Maven używa do uruchamiania testów, aby brał pod uwagę pliki z testami o innych nazwach.

Dla przykładu, dodajmy do katalogu `src/test/java` plik o nazwie `CheckFactorial.java`, w którym dodamy dwa testy jednostkowe klasy `FactorialCounter`, którą dodaliśmy do projektu w jednym z poprzednich rozdziałów:

policz-silnie/src/test/java/com/kursjava/maven/CheckFactorial.java

```
package com.kursjava.maven;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CheckFactorial {

    @Test
    public void shouldReturnFactorial() {
        assertEquals(120, FactorialCounter.factorial(5));
    }

    @Test(expected = IllegalArgumentException.class)
    public void shouldThrowExceptionForInvalidArgument() {
        FactorialCounter.factorial(-1);
    }
}
```

Jeżeli teraz zlecimy Mavenowi wykonanie testów, to testy z naszego pliku `CheckFactorial` nie zostaną wykonane. Jedyne testy, jakie się wykona, to ten zawarty w pliku `AppTest.java`, który został utworzony podczas generowania projektu za pomocą generatora archetypów:

```
> mvn test

(...)
[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ policz-silnie ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.kursjava.maven.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.033 s - in com.kursjava.maven.AppTest
[INFO]
[INFO] Results:
```

```

[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 2.725 s
[INFO] Finished at: 2020-03-15T11:47:31+01:00
[INFO]
-----

```

Aby plugin Surefire brał pod uwagę testy zawarte w naszym pliku `CheckFactorial.java`, musimy skonfigurować go w pliku `pom.xml`:

```

<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.1</version>
  <configuration>
    <includes>
      <include>**/*Test.java</include>
      <include>CheckFactorial.java</include>
    </includes>
  </configuration>
</plugin>

```

Pliki z testami, które ma brać pod uwagę plugin Surefire, umieszczamy w elementach `<include>`. Zauważ, że dodaliśmy także `<include>` z `**/*Test.java`, ponieważ ręczna konfiguracja nazw plików testowych powoduje, że domyślne wzorce, które przedstawiłem na początku rozdziału, przestają być używane. Gdybym pominął ten `<include>`, to jedynie testy z pliku `CheckFactorial.java` byłyby wykonywane.

Jeżeli teraz wykonamy testy, to zobaczymy, że wykonane zostały testy z obu plików:

```

> mvn test

(...)
[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ policz-silnie ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.kursjava.maven.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.046 s - in com.kursjava.maven.AppTest
[INFO] Running com.kursjava.maven.CheckFactorial
12:09:59 ERROR com.kursjava.maven.FactorialCounter - Nieprawidlowa wartosc
n: -1
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.384 s - in com.kursjava.maven.CheckFactorial
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
-----

```



```
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 3.161 s
[INFO] Finished at: 2020-03-15T12:10:00+01:00
[INFO]
-----
```

Pliki z testami możemy także wykluczać korzystając z elementu `<exclude>`, którego elementem nadrzędnym powinien być element `<excludes>` zawarty z kolei w elemencie `<configuration>`, widocznym powyżej.

Czasami możemy mieć potrzebę wykonać testy jednostkowe z jednego, konkretnego pliku. Aby wskazać ten plik, korzystamy z parametru `test`, którego wartością powinna być nazwa klasy (bez rozszerzenia):

```
> mvn -Dtest=CheckFactorial test

(...)
[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ policz-silnie ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.kursjava.maven.CheckFactorial
12:13:41 ERROR com.kursjava.maven.FactorialCounter - Nieprawidlowa wartosc
n: -1
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.415 s - in com.kursjava.maven.CheckFactorial
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 2.453 s
[INFO] Finished at: 2020-03-15T12:13:42+01:00
[INFO]
-----
```

Za pomocą `mvn -Dtest=CheckFactorial test` zleciliśmy Mavenowi wykonanie testów w klasie `CheckFactorial` – nazwę tej klasy ustawiliśmy jako parametr o nazwie `test` (parametr określający plik z testami do wykonania i faza testów nazywają się w tym przypadku tak samo – `test`).

Więcej informacji o uruchamianiu i konfiguracji testów znajdziesz w oficjalnej dokumentacji Maven:

<https://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html>
<https://maven.apache.org/surefire/maven-surefire-plugin/examples/single-test.html>

5.6 Dodawanie testów integracyjnych do fazy verify

Poza testami jednostkowymi, Maven może także uruchamiać dla nas testy integracyjne w fazie `verify`.

Domyślnie jednak funkcjonalność ta nie jest włączona. Aby Maven wykonał testy integracyjne, musimy do pliku `pom.xml` dodać konfigurację pluginu o nazwie *Failsafe*, którego zadaniem jest właśnie uruchamianie testów integracyjnych.

Plugin Failsafe domyślnie wykonuje testy w plikach, które pasują do któregoś z poniższych wzorców (** oznacza dowolny katalog w projekcie):

- `**/IT*.java`
- `**/*IT.java`
- `**/*ITCase.java`

Poniżej znajduje się przykładowa konfiguracja tego pluginu dodana do pliku `pom.xml` naszego projektu:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.0.0-M4</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Do katalogu `src/test/java/com/kursjava/maven` dodamy plik `ITFactorialCounter.java` z przykładowym testem:

`src/test/java/com/kursjava/maven/ITFactorialCounter.java`

```
package com.kursjava.maven;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class ITCheckFactorial {
    @Test
    public void shouldReturnFactorial() {
        assertEquals(24, FactorialCounter.factorial(4));
    }
}
```

Możemy teraz zlecić Mavenowi zbudowanie projektu wraz z wykonaniem testów integracyjnych przy użyciu `mvn verify`:

```

> mvn verify

(...)
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.kursjava.maven.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.046 s - in com.kursjava.maven.AppTest
[INFO] Running com.kursjava.maven.CheckFactorial
18:18:49 ERROR com.kursjava.maven.FactorialCounter - Nieprawidlowa wartosc
n: -1
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.381 s - in com.kursjava.maven.CheckFactorial
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO]
(...)
[INFO] --- maven-failsafe-plugin:3.0.0-M4:integration-test (default) @
policz-silnie ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.kursjava.maven.ITCheckFactorial
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.412 s - in com.kursjava.maven.ITCheckFactorial
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-failsafe-plugin:3.0.0-M4:verify (default) @ policz-silnie
---
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 6.994 s
[INFO] Finished at: 2020-03-15T18:18:53+01:00
[INFO]
-----

```

Jak widzimy powyżej, najpierw wykonane zostały testy jednostkowe przez plugin Surefire, a dopiero potem plugin Failsafe wykonał test integracyjny.

Plugin ten możemy konfigurować podobnie jak plugin Surefire. Możemy zmienić konfigurację pluginu, by szukał testów integracyjnych w innym podkatalogu czy też zmienić wzorce dopasowania plików z testami itp.

Więcej informacji o pluginie Failsafe znajdziesz w oficjalnej dokumentacji Maven:

<https://maven.apache.org/surefire/maven-failsafe-plugin/plugin-info.html>

<https://maven.apache.org/surefire/maven-failsafe-plugin/examples/inclusion-exclusion.html>

5.7 Podsumowanie

- Aby zbudować projekt, Maven wykonuje ponad 20 zależnych od siebie faz (*build lifecycle phases*), w skład których wchodzi m. in. fazy: `compile`, `test`, `package`, oraz `install`.
- Pluginy Mavena to po prostu projekty Mavenowe, które udostępniają pewną funkcjonalność, z której możemy korzystać używając Mavena.
- Część z pluginów jest od razu używana przez Maven, jak na przykład plugin Maven Compiler, a inne musimy wpierw skonfigurować w pliku `pom.xml`.
- Z pluginów korzysta się podając nazwę pluginu oraz jedno z zadań (*plugin goal*), które może on wykonać, np. komenda `mvn exec:java` zleca wykonanie zadania `java` pluginu `Exec`.
- Fazy budowy projektu mogą mieć przypisane zadania pluginów. Gdy Maven wykonuje daną fazę, to uruchomi wszystkie zadania pluginów skojarzone z tą fazą. Dla przykładu, faza `package` uruchamia zadanie `jar` pluginu `Maven Jar`.
- Aby sprawdzić, jakie fazy budowy projektu są dostępne i zobaczyć ich domyślnie przypisane zadania pluginów, skorzystaj z komendy:

```
mvn help:describe -Dcmd=install
```

- Aby zobaczyć informacje o pluginie i jego zadania, skorzystaj z poniższej komendy (wartość dla parametru `plugin` to połączenie `groupId` i `artifactId` pluginu):

```
mvn help:describe -Dplugin=org.apache.maven.plugins:maven-surefire-plugin
```

- Domyślnie generowany przez Maven plik JAR w katalogu `target` zawiera jedynie klasy naszego projektu. Możemy skorzystać z pluginu *Assembly*, aby wygenerować jeden duży JAR zawierający wszystkie zależności.
- Plugin *Assembly* przed użyciem należy skonfigurować w pliku `pom.xml`:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>3.2.0</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- Aby skorzystać z tego pluginu, korzystamy z komendy:

```
mvn compile assembly:single
```

- Wynikiem działania pluginu *Assembly* jest plik JAR w katalogu `target` o przykładowej nazwie `policz-silnie-1.0-SNAPSHOT-jar-with-dependencies.jar`.
- W pliku `pom.xml` możemy podłączyć zadania pluginów pod fazy budowy projektu. Dla

przykładu, plugin Assembly mógłby zostać przypisany do fazy package:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>3.2.0</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

- W fazie `test`, Maven domyślnie wykonuje testy zawarte w plikach projektu, których nazwa pasuje do któregoś z poniższych wzorców (`**` oznacza dowolny katalog projektu):
 - `**/Test*.java`
 - `**/*Test.java`
 - `**/*Tests.java`
 - `**/*TestCase.java`
- Testy powinniśmy umieszczać w katalogu `src/test/java`.
- Jeżeli mamy potrzebę korzystać z pliku z testami, który nie pasuje po powyższych wzorców, to możemy skonfigurować plugin Surefire, którego Maven używa do uruchamiania testów, aby brał pod uwagę pliki z testami o innych nazwach:

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.1</version>
  <configuration>
    <includes>
      <include>**/*Test.java</include>
      <include>CheckFactorial.java</include>
    </includes>
  </configuration>
</plugin>
```

- Zauważmy, że dodaliśmy także `<include>` z `**/*Test.java`, ponieważ ręczna konfiguracja nazw plików testowych powoduje, że domyślne wzorce, które są wylistowane powyżej, przestają być używane.
- Pliki z testami możemy także wykluczać korzystając z elementu `<exclude>`.
- Możemy zlecić wykonanie testów z konkretnej klasy, przekazując jej nazwę jako wartość parametru `test` w trakcie wykonywania fazy o tej samej nazwie:

```
mvn -Dtest=CheckFactorial test
```

- Aby pominąć wykonanie testów należy ustawić parametr `maven.test.skip`:

```
mvn install -Dmaven.test.skip=true
```

- Poza testami jednostkowymi, Maven może także uruchamiać dla nas testy integracyjne w fazie `verify`. Domyślnie ta funkcjonalność nie jest włączona.
- Aby Maven wykonał testy integracyjne, musimy do pliku `pom.xml` dodać konfigurację pluginu o nazwie *Failsafe*. Plugin *Failsafe* domyślnie wykonuje testy w plikach, które pasują do któregoś z poniższych wzorców (** oznacza dowolny katalog w projekcie):
 - `**/IT*.java`
 - `**/*IT.java`
 - `**/*ITCase.java`
- Poniżej znajduje się przykładowa konfiguracja tego pluginu dodana do pliku `pom.xml` naszego projektu:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>3.0.0-M4</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

- W plikach `pom.xml` możemy używać nazwanych wartości, umieszczając je w elemencie `<properties>`:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.7</maven.compiler.source>
  <maven.compiler.target>1.7</maven.compiler.target>
  <log4j.version>2.13.1</log4j.version>
</properties>
```

- Aby odnieść się do nazwanej wartości, korzystamy ze składni `${nazwa}`, na przykład:

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>${log4j.version}</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>${log4j.version}</version>
</dependency>
```

5.8 Zadania

Dodaj do swojego projektu klasę z testami, która będzie miała nazwę pasującą do jednego ze wzorców klas testowych używanych w Maven (np. niech klasa ta kończy się na słowo `Test`).

Skonfiguruj plugin Surefire tak, by klasa ta była wykluczana podczas uruchamiania testów w fazie `test`. Zajrzyj do oficjalnej dokumentacji po więcej informacji:

<https://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html>

6 Zależności i projekty wielomodułowe

W tym rozdziale:

- dowiemy się więcej o parametrze `scope` zależności,
- wykorzystamy lokalny projekt jako zależność,
- dowiemy się, co to jest efektywny POM i zależności przechodnie,
- zobaczymy, jak skonfigurować prosty projekt wielomodułowy.

6.1 Parametr scope zależności

Gdy konfigurujemy zależność w pliku `pom.xml` w elemencie `<dependency>`, możemy ustawić wartość opcjonalnego parametru o nazwie `scope`.

Domyślnie `scope` ma wartość `compile`, co powoduje, że zależność jest wymagana zarówno podczas kompilacji, testów, a także wykonywania naszego programu. Gdy w jednym z poprzednich rozdziałów dodawaliśmy zależność do Log4j, nie ustawiliśmy `scope`, więc parametrowi temu nadana została wartość domyślna `compile`:

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-api</artifactId>
  <version>2.13.1</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.13.1</version>
</dependency>
```

Scope może przyjmować kilka wartości – spójrzmy na trzy z nich:

- `compile` – wartość domyślna, zależność wymagana podczas kompilacji, testów, oraz wykonywania programu,
- `test` – zależność potrzebna tylko w fazie testów – taką wartość ustawiamy dla np. JUnit, ponieważ JUnit potrzebujemy w naszych aplikacjach tylko, gdy je testujemy – w wersji produkcyjnej naszego kodu JUnit nie jest potrzebne:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

- `runtime` – zależność potrzebna dopiero na etapie działania programu – nie jest wymagana ani podczas kompilacji, ani testów – przykładem może być np. sterownik do obsługi bazy danych takiej jak Oracle.

Więcej informacji o `scope` i opis pozostałych wartości, jakie może przyjmować, znajdziesz w [oficjalnej dokumentacji](#).

6.2 Zależności przechodnie (transitive dependencies)

Gdy w pliku `pom.xml` dodajemy pewną zależność, Maven pobiera ją z centralnego repozytorium. Poza plikiem JAR, pobiera także plik `pom.xml`, dzięki czemu może sprawdzić, *jakie zależności mają nasze zależności*.

Zależności takie nazywamy *przechodnimi* (*transitive dependencies*) i w większych projektach może ich być bardzo wiele – dodanie do projektu jednej biblioteki może pociągnąć za sobą kilkanaście zależności przechodnich.

Spójrzmy na poniższy przykład – wygenerowałem prosty szkielet aplikacji Spring Boot za pomocą *Spring Initializr*. Aby otrzymać listę wszystkich zależności tego projektu (bezpośrednich oraz przechodnich), korzystamy z komendy `mvn dependency:tree`, która zwraca informację o zależnościach w formie drzewa:

```
> mvn dependency:tree

--- maven-dependency-plugin:3.1.1:tree (default-cli) @ spring-przyklad ---
com.kursjava.maven:spring-przyklad:jar:0.0.1-SNAPSHOT
+- org.springframework.boot:spring-boot-starter:jar:2.2.5.RELEASE:compile
| +- org.springframework.boot:spring-boot:jar:2.2.5.RELEASE:compile
| | \- org.springframework:spring-context:jar:5.2.4.RELEASE:compile
| |   +- org.springframework:spring-aop:jar:5.2.4.RELEASE:compile
| |   +- org.springframework:spring-beans:jar:5.2.4.RELEASE:compile
| |   \- org.springframework:spring-expression:jar:5.2.4.RELEASE:compile
| +- org.springframework.boot:spring-boot-autoconfigure:jar:2.2.5.RELEASE:compile
| +- org.springframework.boot:spring-boot-starter-logging:jar:2.2.5.RELEASE:compile
| | +- ch.qos.logback:logback-classic:jar:1.2.3:compile
| | | \- ch.qos.logback:logback-core:jar:1.2.3:compile
| | +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.12.1:compile
| | | \- org.apache.logging.log4j:log4j-api:jar:2.12.1:compile
| | \- org.slf4j:jul-to-slf4j:jar:1.7.30:compile
| +- jakarta.annotation:jakarta.annotation-api:jar:1.3.5:compile
| +- org.springframework:spring-core:jar:5.2.4.RELEASE:compile
| | \- org.springframework:spring-jcl:jar:5.2.4.RELEASE:compile
| \- org.yaml:snakeyaml:jar:1.25:runtime
\-- org.springframework.boot:spring-boot-starter-test:jar:2.2.5.RELEASE:test
    +- org.springframework.boot:spring-boot-test:jar:2.2.5.RELEASE:test
    +- org.springframework.boot:spring-boot-test-autoconfigure:jar:2.2.5.RELEASE:test
    +- com.jayway.jsonpath:json-path:jar:2.4.0:test
    | +- net.minidev:json-smart:jar:2.3:test
    | | \- net.minidev:accessors-smart:jar:1.2:test
    | |   \- org.ow2.asm:asm:jar:5.0.4:test
    | \- org.slf4j:slf4j-api:jar:1.7.30:compile
    +- jakarta.xml.bind:jakarta.xml.bind-api:jar:2.3.2:test
    | \- jakarta.activation:jakarta.activation-api:jar:1.2.2:test
    +- org.junit.jupiter:junit-jupiter:jar:5.5.2:test
    | +- org.junit.jupiter:junit-jupiter-api:jar:5.5.2:test
    | | +- org.apiguardian:apiguardian-api:jar:1.1.0:test
    | | +- org.opentest4j:opentest4j:jar:1.2.0:test
    | | \- org.junit.platform:junit-platform-commons:jar:1.5.2:test
    | +- org.junit.jupiter:junit-jupiter-params:jar:5.5.2:test
    | \- org.junit.jupiter:junit-jupiter-engine:jar:5.5.2:test
    |   \- org.junit.platform:junit-platform-engine:jar:1.5.2:test
    +- org.mockito:mockito-junit-jupiter:jar:3.1.0:test
    +- org.assertj:assertj-core:jar:3.13.2:test
    +- org.hamcrest:hamcrest:jar:2.1:test
    +- org.mockito:mockito-core:jar:3.1.0:test
    | +- net.bytebuddy:byte-buddy:jar:1.10.8:test
    | +- net.bytebuddy:byte-buddy-agent:jar:1.10.8:test
    | \- org.objenesis:objenesis:jar:2.6:test
    +- org.skyscreamer:jsonassert:jar:1.5.0:test
```

```
| \- com.vaadin.external.google:android-json:jar:0.0.20131108.vaadin1:test  
+- org.springframework:spring-test:jar:5.2.4.RELEASE:test  
\- org.xmlunit:xmlunit-core:jar:2.6.3:test
```

Spring Initializr to generator aplikacji Springowych, dostępny przez przeglądarkę na stronie <https://start.spring.io>. Pozwala on na łatwe "wyklikanie" szkieletu aplikacji korzystającej ze Springa z możliwością ustawienia wymaganych modułów Springa, ich wersji itp.

Powyższa aplikacja to prosty szkielet aplikacji korzystającej ze Spring Boota – pomimo tego, liczba zależności przechodnich jest bardzo duża, co widać na powyższym listingu.

6.3 Zależność do lokalnego projektu

Zaletą Mavena jest nie tylko to, że możemy wskazać zależności do pewnych bibliotek, które Maven pobierze z centralnego repozytorium, ale także możliwość korzystania z naszych własnych projektów jako zależności.

Na początku tego kursu widzieliśmy, że Maven w fazie `install` kopiuje wygenerowany plik JAR z naszym projektem do lokalnego repozytorium artefaktów `.m2/repository` – od tej pory możemy w innych naszych projektach dodawać elementy `<dependency>` wskazujące na nasz projekt.

W projekcie z poprzedniego rozdziału zawarliśmy klasę, która liczyła silnię. W innym projekcie moglibyśmy dodać następujący wpis do pliku `pom.xml`, aby móc wykorzystać funkcjonalności z tamtego projektu:

wykorzystanie-innego-projektu/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.kursjava.maven</groupId>
  <artifactId>wykorzystanie-innego-projektu</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.kursjava.maven</groupId>
      <artifactId>policz-silnie</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```

W klasach Java tego projektu możemy teraz korzystać z klasy `FactorialCounter`, którą zaimportowaliśmy za pomocą elementu `<dependency>`:

wykorzystanie-innego-projektu/src/main/java/com/kursjava/maven/Main.java

```
package com.kursjava.maven;

import com.kursjava.maven.FactorialCounter;

public class Main {
  public static void main(String[] args) {
    System.out.println("Silnia 10 = " + FactorialCounter.factorial(10));
  }
}
```

6.4 Projekty wielomodułowe

Często projekty, nad którymi pracujemy, dzielimy na różne moduły – jeden może być odpowiedzialny za persystencję danych, inny będzie zawierał model danych, a kolejny – webowy interfejs. Różne elementy takich projektów są zależne od siebie, a dodatkowo wszystkie razem tworzą pewien system. Możemy takie moduły powiązać w Mavenie w jeden projekt wielomodułowy.

Projekty wielomodułowe zawierają w katalogu głównym plik `pom.xml`, który jest "rodzicem" dla wszystkich podmodułów, które są w nim zdefiniowane. Podmoduły to także projekty Mavenowe, które w swoich plikach `pom.xml` odnoszą się do pliku-rodzica `pom.xml`. W tym nadrzędnym pliku `pom.xml` możemy zdefiniować konfigurację pluginów oraz zależności, z których będą mogły korzystać podprojekty. Zaoszczędzi nam to czas i skróci konfigurację plików `pom.xml` w podmodułach.

Spójrzmy na przykładowy plik `pom.xml` będący "rodzicem" w projekcie składającym się z dwóch podmodułów:

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.kursjava.maven.multimod</groupId>
  <artifactId>wielomodulowy-projekt-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>

  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <modules>
    <module>podprojekt1</module>
    <module>podprojekt2</module>
  </modules>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>3.0.0-M4</version>
        <executions>
          <execution>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
```

```
<version>4.11</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

Element `<packaging>` określa, co jest wynikiem zbudowania projektu – do tej pory go nie ustawialiśmy, ponieważ jego domyślna wartość to `jar`. W przypadku pliku `pom.xml`, który jest nadrzędnym plikiem konfiguracyjnym w projekcie wielomodulowym, ten element należy ustawić na `pom`.

Ponadto, powyższy plik zawiera element `<modules>`, w którym wylistowane są wszystkie podmoduły tego projektu – w tym przypadku są to `podprojekt1` oraz `podprojekt2`.

Zauważmy, że w tym pliku skonfigurowany jest także jeden plugin oraz jedna zależność. Dzięki temu, wszystkie podmoduły w tym projekcie będą od razu mogły korzystać z uruchamiania testów integracyjnych (dzięki konfiguracji pluginu Failsafe) oraz będą mogły stosować JUnit w testach jednostkowych. Podprojekty odziedziczą także konfigurację z elementu `<properties>`.

Pierwszy z podmodułów, który powinien znaleźć się w podkatalogu o nazwie `podprojekt1`, skonfigurowany jest następująco:

```
<project>
  <parent>
    <groupId>com.kursjava.maven.multimod</groupId>
    <artifactId>wielomodulowy-projekt-parent</artifactId>
    <version>1.0</version>
  </parent>

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.kursjava.maven.multimod</groupId>
  <artifactId>podprojekt1</artifactId>
  <version>1.0-SNAPSHOT</version>
</project>
```

W pliku `pom.xml` tego podmodułu odnosimy się do projektu-rodzica za pomocą elementu `<parent>`, który zawiera `groupId`, `artifactId`, oraz `version`, nadrzędnego projektu. W ten sposób ten podprojekt dziedziczy konfigurację z nadrzędnego projektu. Pomimo, że ten moduł nie definiuje bezpośrednio zależności do JUnit, to możemy w nim od razu z tej biblioteki korzystać, ponieważ zależność do niej dziedziczymy po projekcie-rodzicu.

Konfiguracja drugiego podprojektu zostanie pominięta – także zawierałaby ona element `<parent>`.

Struktura powyższego projektu powinna wyglądać następująco:

```
wielomodulowy-projekt
|
|-- pom.xml
|
|-- podprojekt1
|   |
|   |-- pom.xml
|   |
|   |-- src
|   |
|   |-- (pozostałe katalogi podprojektu)
|
|-- podprojekt2
|   |
|   |-- pom.xml
|   |
|   |-- src
|   |
|   |-- (pozostałe katalogi podprojektu)
```

W tym wielomodulowym projekcie istnieją trzy pliki `pom.xml` – jeden "rodzic" w katalogu głównym projektu, oraz po jednym pliku `pom.xml` na każdy z podprojektów.

Podprojekty można normalnie budować i uruchamiać na ich rzecz różne fazy Mavena i pluginy, ale możemy także zbiorczo zbudować wszystkie z poziomu projektu nadrzędnego:

```
wielomodulowy-projekt> mvn install

[INFO] Scanning for projects...
[INFO] -----
[INFO] Reactor Build Order:
[INFO]
[INFO] wielomodulowy-projekt-parent [pom]
[INFO] podprojekt1 [jar]
[INFO] podprojekt2 [jar]
[INFO]
[INFO] ----< com.kursjava.maven.multimod:wielomodulowy-projekt-parent >----
[INFO] Building wielomodulowy-projekt-parent 1.0
[INFO] [1/3]
[INFO] -----[ pom ]-----
[INFO]
[INFO] --- maven-failsafe-plugin:3.0.0-M4:integration-test (default) @
wielomodulowy-projekt-parent ---
[INFO] No tests to run.
[INFO]
[INFO] --- maven-failsafe-plugin:3.0.0-M4:verify (default) @ wielomodulowy-
projekt-parent ---
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @
wielomodulowy-projekt-parent ---
[INFO] Installing D:\kurs_maven\przyklady\wielomodulowy-projekt\pom.xml to
C:\Users\Przemek\.m2\repository\com\kursjava\maven\multimod\wielomodulowy-
projekt-parent\1.0\wielomodulowy-projekt-parent-1.0.pom
[INFO]
[INFO] -----< com.kursjava.maven.multimod:podprojekt1 >-----
[INFO] Building podprojekt1 1.0-SNAPSHOT
[INFO] [2/3]
[INFO] -----[ jar ]-----
[INFO]
```

```
(...budowa pierwszego podprojektu...)

[INFO]
[INFO] -----< com.kursjava.maven.multimod:podprojekt2 >-----
[INFO] Building podprojekt2 1.0-SNAPSHOT
[3/3]
[INFO] -----[ jar ]-----

(...budowa drugiego podprojektu...)

[INFO]
-----
[INFO] Reactor Summary:
[INFO]
[INFO] wielomodulowy-projekt-parent 1.0 ..... SUCCESS [ 0.698 s]
[INFO] podprojekt1 1.0-SNAPSHOT ..... SUCCESS [ 1.938 s]
[INFO] podprojekt2 1.0-SNAPSHOT ..... SUCCESS [ 0.595 s]
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 3.364 s
[INFO] Finished at: 2020-03-19T19:52:20+01:00
[INFO]
-----
```

Logi z budowy podmodułów zostały pominięte – byłyby to znane nam już informacje o kolejnych fazach `compile`, `test` (wraz z wykonaniem i podsumowaniem testów wykonanych na rzecz każdego z podmodułów), `package` itd. Na końcu Maven wypisał podsumowanie o projektach zbudowanych w ramach budowy całego wielomodułowego projektu.

6.4.1 Plugin management i dependency management

Konfigurując pluginy i zależności w nadrzędnym pliku `pom.xml` mamy dwie możliwości:

- możemy tak skonfigurować pluginy i/lub zależności, aby zawsze były dziedziczone przez podmoduły,
- możemy skonfigurować pluginy i/lub zależności, ale nie będą one automatycznie dziedziczone przez podprojekty – każdy podprojekt, który będzie chciał tą konfigurację odziedziczyć z pliku `pom.xml`-rodzica, będzie musiał ten plugin/zależność umieścić w swoim pliku `pom.xml`. Taki rodzaj konfiguracji zawarty jest w dodatkowym elemencie w pliku `pom.xml`: dla pluginów jest to element `<pluginManagement>`, a dla zależności – `<dependencyManagement>`.

To rozróżnienie wynika z faktu, że czasem możemy chcieć skonfigurować pewne pluginy i zależności dla wszystkich podmodułów, a inne wstępnie skonfigurować tylko dla tych podmodułów, które faktycznie będą z nich korzystały. Dzięki temu nie wszystkie podmoduły muszą dziedziczyć od razu wszystkie zależności (od pluginów i bibliotek).

Spójrzmy najpierw na przykład użycia elementu `<pluginManagement>` w pliku `pom.xml`-rodzicu:


```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>3.0.0-M4</version>
      <executions>
        <execution>
          <goals>
            <goal>integration-test</goal>
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
        <executions>
          <execution>
            <id>assemble-jar-with-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

```

W powyższym pliku `pom.xml` znajdują się konfiguracje dwóch pluginów: Failsafe oraz Assembly. Ten drugi zawarty jest w elemencie `<pluginManagement>`, więc nie będzie on od razu wykorzystywany w podprojektach – jeżeli zbudowalibyśmy teraz jeden z podprojektów, to plugin Assembly *nie* zostałby automatycznie użyty w fazie `package` do wygenerowanie pliku JAR ze wszystkimi zależnościami.

Aby plugin skonfigurowany w pliku `pom.xml`-rodzicu był wykorzystywany w podprojekcie, musi on dodać do swojego pliku `pom.xml` informację, że taki plugin chce używać:

```

<project>
  <parent>
    <groupId>com.kursjava.maven.multimod</groupId>
    <artifactId>wielomodulowy-projekt-parent</artifactId>
    <version>1.0</version>
  </parent>

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.kursjava.maven.multimod</groupId>
  <artifactId>podprojekt1</artifactId>
  <version>1.0-SNAPSHOT</version>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>

```

Zauważ, że konfiguracja tego pluginu w podprojekcie jest minimalna – w zasadzie podajemy tylko jego nazwę. Reszta konfiguracji zostanie wzięta z pliku `pom.xml`-rodzica. Powyższa konfiguracja spowoduje, że teraz budując projekt, w fazie `package` będzie dodatkowo używany plugin Assembly (zgodnie z konfiguracją odziedziczoną z nadrzędnego pliku `pom.xml`).

Element `<dependencyManagement>` działa podobnie. Spójrzmy na przykład:

```

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
      <version>${log4j.version}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
      <version>${log4j.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>

```

Jeżeli skonfigurujemy zależność do Log4j w elemencie `<dependencyManagement>`, to podprojekty nie będą od razu zależne od Log4j w przeciwieństwie do JUnit, do którego zależność została

skonfigurowana poza tym elementem.

Jeżeli któryś z podprojektów chciałby korzystać z Log4j, to musiałby do swojego pliku `pom.xml` dodać następujący wpis:

wielomodulowy-projekt/podprojekt2/pom.xml

```
<project>
  <parent>
    <groupId>com.kursjava.maven.multimod</groupId>
    <artifactId>wielomodulowy-projekt-parent</artifactId>
    <version>1.0</version>
  </parent>

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.kursjava.maven.multimod</groupId>
  <artifactId>podprojekt2</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
    </dependency>
  </dependencies>
</project>
```

Tak jak w przypadku konfiguracji pluginów dziedziczonej z nadrzędnego pliku `pom.xml`, tak i w przypadku zależności wystarczy podać jej nazwę. Wersja (i ewentualnie inne elementy konfiguracji) zostaną odziedziczone z pliku `pom.xml`-rodzica. Teraz podprojekt2 może korzystać z Log4j.

Podprojekt ten może także stosować JUnit, ale tej zależności w ogóle nie musi konfigurować, ponieważ została ona skonfigurowana w nadrzędnym pliku `pom.xml` poza elementem `<dependencyManagement>`, więc jest od razu automatycznie dziedziczona przez podprojekty.

W podprojektach możesz zmieniać konfigurację pluginów i zależności odziedziczoną z nadrzędnego pliku `pom.xml`, jeżeli masz taką potrzebę.

W przykładach w tym rozdziale skróciłem początki plików `pom.xml` dla zwięzłości – powinny one wyglądać następująco:

```
<?xml version="1.0" encoding="UTF-8"?>

<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/xsd/maven-4.0.0.xsd">
```

6.5 Efektywny POM

Czasem możemy mieć potrzebę sprawdzić, jak wygląda *finalny* plik `pom.xml` np. w jednym z podprojektów w projekcie wielomodulowym. Możemy wtedy skorzystać z komendy `mvn help:effective-pom`, która zwróci zawartość pliku `pom.xml` projektu uwzględniając:

- domyślne ustawienia Mavena dla projektów,
- ustawienia odziedziczone z nadrzędnego pliku `pom.xml`.

Efektywny `pom.xml` jest bardzo długim plikiem – poniżej znajduje się fragment wyniku komendy `mvn help:effective-pom` wywołanej na rzecz modułu `podprojekt1`. Zauważ, że w tym efektywnym pliku `pom.xml` są m.in. elementy `<properties>` i `<dependency>` z JUnit, odziedziczone z pliku `pom.xml`-rodzica:

```
podprojekt1> mvn help:effective-pom
```

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.kursjava.maven.multimod</groupId>
    <artifactId>wielomodulowy-projekt-parent</artifactId>
    <version>1.0</version>
  </parent>
  <groupId>com.kursjava.maven.multimod</groupId>
  <artifactId>podprojekt1</artifactId>
  <version>1.0-SNAPSHOT</version>
  <properties>
    <log4j.version>2.13.1</log4j.version>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>2.13.1</version>
      </dependency>
      <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>2.13.1</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <repositories>
```

```
<repository>
  <snapshots>
    <enabled>>false</enabled>
  </snapshots>
  <id>central</id>
  <name>Central Repository</name>
  <url>https://repo.maven.apache.org/maven2</url>
</repository>
</repositories>

(... pozostała część pliku została pominięta ... )

</project>
```

Komenda `mvn help:effective-pom` przydaje się nie tylko w projektach wielomodułowych. Dzięki niej możemy zbadać wszystkie wartości ustawiane domyślnie przez Maven w naszym projekcie.

6.6 Podsumowanie

- Parametr `scope`, który możemy ustawić w `<dependency>`, określa, kiedy zależność jest wymagana przez nasz projekt.
- Domyślnie `scope` ma wartość `compile`, co oznacza, że zależność jest wymagana zarówno podczas kompilacji, testów, a także wykonywania naszego programu.
- Inną możliwą wartością `scope` jest np. `test`, co powoduje, że zależność wymagana jest jedynie podczas fazy testów. Przykładem jest JUnit:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

- Gdy w pliku `pom.xml` dodajemy pewną zależność, Maven pobiera ją z centralnego repozytorium. Poza plikiem JAR, pobiera także plik `pom.xml`, dzięki czemu może sprawdzić *jake zależności mają nasze zależności*.
- Zależności przechodnie (*transitive dependencies*) to zależności naszych zależności.
- Dodanie do projektu jednej biblioteki może pociągnąć za sobą kilkanaście zależności przechodnich.
- Aby otrzymać listę wszystkich zależności projektu (bezpośrednich oraz przechodnich), korzystamy z komendy `mvn dependency:tree`
- Zaletą Mavena jest nie tylko to, że możemy wskazać zależności do pewnych bibliotek, które Maven pobierze z centralnego repozytorium, ale także możliwość korzystania z naszych własnych projektów jako zależności:

```
<dependency>
  <groupId>com.kursjava.maven</groupId>
  <artifactId>policz-silnie</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

- Maven pozwala na tworzenie projektów wielomodułowych.
- Projekty wielomodułowe zawierają w katalogu głównym plik `pom.xml`, który jest "rodzicem" dla wszystkich podmodułów, które są w nim zdefiniowane.
- W tym "nadrzędnym" pliku `pom.xml` możemy zdefiniować konfigurację pluginów oraz zależności, z których będą mogły korzystać podprojekty. Zaoszczędzi nam to czas i skróci konfigurację plików `pom.xml` w podmodułach.
- W nadrzędnym pliku `pom.xml` listę podmodułów umieszczamy w elemencie `<modules>`:

```
<modules>
  <module>podprojekt1</module>
  <module>podprojekt2</module>
</modules>
```

- Dla projektu-rodzica ustawiamy element `<packaging>` na `pom`. Element ten określa, co jest wynikiem zbudowania projektu – do tej pory nie ustawialiśmy tego elementu, ponieważ

jego domyślna wartość to jar:

```
<groupId>com.kursjava.maven.multimod</groupId>
<artifactId>wielomodulowy-projekt-parent</artifactId>
<packaging>pom</packaging>
<version>1.0</version>
```

- Podmoduły to także projekty Mavenowe, które w swoich plikach `pom.xml` odnoszą się do pliku-rodzica `pom.xml` w elemencie `<parent>`:

```
<project>
  <parent>
    <groupId>com.kursjava.maven.multimod</groupId>
    <artifactId>wielomodulowy-projekt-parent</artifactId>
    <version>1.0</version>
  </parent>

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.kursjava.maven.multimod</groupId>
  <artifactId>podprojekt1</artifactId>
  <version>1.0-SNAPSHOT</version>
</project>
```

- Podprojekty można normalnie budować i uruchamiać na ich rzecz różne fazy Mavena i pluginy, ale możemy także zbiorczo zbudować wszystkie z poziomu projektu nadrzędnego.
- Konfiguruując pluginy i zależności w nadrzędnym pliku `pom.xml` mamy dwie możliwości:
 - możemy tak skonfigurować pluginy i/lub zależności, aby zawsze były dziedziczone przez podmoduły,
 - możemy skonfigurować pluginy i/lub zależności, ale nie będą one automatycznie dziedziczone przez podprojekty – każdy podprojekt, który będzie chciał tą konfigurację odziedziczyć z pliku `pom.xml`-rodzica, będzie musiał ten plugin/zależność umieścić w swoim pliku `pom.xml`. Taki rodzaj konfiguracji zawarty jest w dodatkowym elemencie w pliku `pom.xml`: dla pluginów jest to element `<pluginManagement>`, a dla zależności – `<dependencyManagement>`.
- Czasem możemy mieć potrzebę sprawdzić, jak wygląda finalny plik `pom.xml` np. w jednym z podprojektów w projekcie wielomodulowym. Możemy wtedy skorzystać z komendy `mvn help:effective-pom`, która zwróci zawartość pliku `pom.xml` projektu uwzględniając:
 - domyślne ustawienia Mavena dla projektów,
 - ustawienia odziedziczone z nadrzędnego pliku `pom.xml`.

7 Podsumowanie Podstaw Maven

Po przeczytaniu tego dokumentu powinieneś mieć solidne podstawy do rozpoczęcia korzystania z Mavena. Zwróć uwagę, że Maven to potężne narzędzie i oferuje bardzo dużo – poruszone przeze mnie tematy to jedynie wierzchołek góry lodowej. Pracując na projektach komercyjnych lub open source będziesz musiał często korzystać z dokumentacji Maven, w której znajdziesz bardzo dużo informacji i przykładów:

<https://maven.apache.org/guides/index.html>

Jeżeli uważasz, że coś istotnego zostało pominięte, lub niewystarczająco wytłumaczone, daj znać – będę wdzięczny!

8 Dodatek – przydatne informacje i komendy

Poniżej znajdziesz zbiór przydatnych informacji i komend Maven.

8.1 Informacje

- Maven wymaga ustawienia `JAVA_HOME` na główny katalog JDK.
- Konfiguracja projektów Maven zawarta jest w pliku `pom.xml` w katalogu głównym projektu.
- `${basedir}` oznacza główny katalog projektu w Maven.
- `${basedir}/src/main/java` – tutaj umieszczamy kod Java.
- `${basedir}/src/main/resources` – miejsce na zasoby.
- `${basedir}/src/test/java` – testy jednostkowe.
- `${basedir}/src/test/resources` – zasoby testów.
- `target` to katalog, który zawiera skompilowane klasy projektu i plik JAR/WAR.
- Katalog `target` powinien być w pliku `.gitignore`, aby nie był wersjonowany.
- Lokalne repozytorium artefaktów znajduje się w katalogu użytkownika w podkatalogu `.m2`.
- Ustawienia dla Mavena można zawrzeć w pliku `settings.xml`, który powinien być umieszczony w katalogu `.m2`
- `groupId`, `artifactId` i `version` – określają projekty w Mavenie.
- Zależność lokalizacji w lokalnym repozytorium `.m2` od powyższych wartości:

`.m2\repository\com\kursjava\maven\hello-maven\1.0-SNAPSHOT`

```
graph TD
    groupId[groupId] --- path[com]
    artifactId[artifactId] --- path
    version[version] --- path
    path --- fullPath[.m2\repository\com\kursjava\maven\hello-maven\1.0-SNAPSHOT]
```

- Nazwy plików z testami jednostkowymi to pliki pasujące do jednego ze wzorców: `Test*.java`, `*Test.java`, `*Tests.java`, `*TestCase.java`
- Nazwy plików z testami integracyjnymi to pliki pasujące do jednego ze wzorców: `IT*.java`, `*IT.java`, `*ITCase.java`
- Aby testy integracyjne były uruchamiane w fazie `verify`, należy skonfigurować plugin `Failsafe`.

8.2 Przydatne komendy

- Pobranie lokalizacji lokalnego repozytorium .m2:

```
mvn help:evaluate -Dexpression=settings.localRepository
```

- Lista faz budowy projektu i domyślnie skonfigurowane pluginy / informacje o pluginie:

```
mvn help:describe -Dcmd=install
```

```
mvn help:describe -Dplugin=org.apache.maven.plugins:maven-surefire-plugin
```

- Efektywny POM:

```
mvn help:effective-pom
```

- Uruchom klasę bez ręcznego ustawiania classpath (przed użyciem należy `mvn compile`):

```
mvn exec:java -Dexec.mainClass=com.kursjava.maven.HelloMaven
```

- Skompiluj / uruchom testy / uruchom testy integracyjne / wygeneruj jar / zainstaluj projekt:

```
mvn compile
mvn test
mvn verify
mvn package
mvn install
```

- Czyszczenie plików wygenerowanych podczas budowania projektu / łączenie komend:

```
mvn clean
mvn clean compile
```

- Pomiń testy jednostkowe:

```
mvn install -Dmaven.test.skip=true
```

- Uruchamianie testów z konkretnej klasy:

```
mvn -Dtest=CheckFactorial test
```

- Wygeneruj projekt za pomocą generatora archetypów – tryb interaktywny / nieinteraktywny:

```
mvn archetype:generate
mvn archetype:generate -B -DarchetypeGroupId=org.apache.maven.archetypes
-DarchetypeArtifactId=maven-archetype-quickstart
-DgroupId=com.kursjava.maven -DartifactId=wygenerowany-projekt
-Dversion=1.0-SNAPSHOT -Dpackage=com.kursjava.maven
```

- Generowanie JARa z zależnościami przy użyciu pluginu Assembly:

```
mvn compile assembly:single
```

- Zależności projektu (bezpośrednie i przechodnie):

```
mvn dependency:tree
```