Maven, Pizza Project

From last class: Pizza1 directories/packages

Here's the directory system for the pizza1 sources (using "tree" command in Windows), looking down from the top of the source directory tree, itself at src/main/java in the project's directory system.

```
C:\cs\cs636\pizza1\src\main\java>tree
Folder PATH listing
Volume serial number is 385D-9017
C:.
└───cs636
    └───pizza
        ├───config
        ├───dao
        ├───domain
        ├───presentation
        └───service
```

←for package cs636.pizza.config

←for package cs636.pizza.dao, etc.

How do we build this from scratch?

Using maven to build a prototype setup for pizza1

From tutorial from https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false

Edited to use pizza1 artifactId and cs636.pizza groupId--

mvn archetype:generate -DgroupId=cs636.pizza -DartifactId=pizza1 -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false

- We run this command to build the directory system that Maven wants us to have. Maven is "opinionated", i.e., has specific ideas on how to set up the project.
- This does make it easier to understand new project once you get used to it.

Resulting project, available at $cs636/maven-ex

```
F:\cs\cs636\testmaven\pizza1>tree
Folder PATH listing
Volume serial number is A081-715F
F:.
└───src
    ├───main
    │   └───java
    │       └───cs636
    │           └───pizza
    └───test
        └───java
            └───cs636
                └───pizza
```

←with App.java, a Hello World app

← with AppTest.java, a Junit4 unit test (we'll use Junit5 eventually)

- This sets up a standard Maven project, so the sources root is at src/main/java relative to the base directory of the project.
- Using Maven, we are expected to put the test sources in a separate tree from the regular sources--they are rooted at src/test/java, with the same tree structure for the packages.
- To finish this for the real project, we need to cd down to the pizza directory and create directories for presentation, service, dao, etc. But for now, just try out App.java…

Trying out this project: mvn = maven tool

`mvn package` compiles files, runs unit tests, builds jar file

Puts results in target directory: let's see what files are there…

```
F:\cs\cs636\testmaven1\pizza1>ls target
classes generated-test-sources maven-status surefire-reports
generated-sources maven-archiver pizza1-1.0-SNAPSHOT.jar test-classes
```

`classes` contains the main .class directory tree
`test-classes` contains the test .class dir tree
`pizza1-1.0-SNAPSHOT.jar` is the jar file we can use to execute the app

Run the app: put the jar on the classpath and execute cs636.pizza.App within it:

```
java -cp target/pizza1-1.0-SNAPSHOT.jar cs636.pizza.App
Hello World!
```

Look in the jar file

F:\cs\cs636\testmaven1\pizza1\target>jar -tf *.jar
META-INF/MANIFEST.MF
META-INF/
cs636/
cs636/pizza/
META-INF/maven/
META-INF/maven/cs636.pizza/
META-INF/maven/cs636.pizza/pizza1/
cs636/pizza/App.class ← the only .class file in this jar
META-INF/maven/cs636.pizza/pizza1/pom.properties
META-INF/maven/cs636.pizza/pizza1/pom.xml

Look at the proto's pom.xml: first part

```
<?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>                ←needed start to pom.xml up to here

  <groupId>cs636.pizza</groupId>          ←our app identified
  <artifactId>pizza1</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>pizza1</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <!-- Changed from 1.7 by eoneil -->
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
```

Properties: just name-value associations, i.e., in effect string variables. Evaluation of property foo: ${foo} Can build up strings from pieces

Rest of pom: just junit dependency, version specs

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>   ←we'll use version 5 of JUnit because that's what Spring Boot wants now
    <version>4.11</version>
    <scope>test</scope>    ← This says we only need this dependency for tests, not production executions
  </dependency>
 </dependencies>

 <build>
  <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be moved to parent pom) -->
    <plugins>
      <!-- clean lifecycle, see https://maven.apache.org/ref/current/maven-core/lifecycles.html#clean_Lifecycle -->
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <version>3.1.0</version>
      </plugin>
      <!-- default lifecycle, jar packaging: see https://maven.apache.org/ref/current/maven-core/default-
bindings.html#Plugin_bindings_for_jar_packaging -->
      <plugin>
        <artifactId>maven-resources-plugin</artifactId>
        <version>3.0.2</version>
      </plugin>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
      </plugin>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.22.1</version>
      </plugin>
…
```

We'll have these entries to ensure current-enough compile, unit-test execution support

Maven and running the program

We see that out of the box, maven doesn't support running the main class with a mvn command. It can be made to do so. See https://stackoverflow.com/questions/1089285/maven-run-project.

There is a mvn command for running unit tests: **mvn test.** To do this, mvn must use junit.jar, but it's nowhere to be found in our project directories. It's in maven's local repository.

When we get to Spring Boot, there will be a **mvn spring-boot:run** command to use. We are starting off simply, so we'll just build a big jar file (a "fat jar") with dependencies to execute from

Maven is an expert on gathering dependencies, managing them, and building programs. The bigger and more complex the project is, the more Maven is a help.

mvn clean

Another important mvn command: **mvn clean**
It deletes the whole target directory, where the classes and jars were built.

Use **mvn clean package** to do a clean build.

Pizza1 with the actual sources

The first step is studying the supplied Pizza1 project, the example to guide our own development project

Also the pizza project “Project Description”, linked to the class web page

We use the layered architecture discussed in the first class. We see packages for presentation, service, and dao, corresponding to the layers.

The package “domain” has classes PizzaOrder, PizzaTopping, and PizzaSize, MenuSize and MenuTopping. These represent the needed domain objects, the subjects of the application. so the domain package does not have its own layer, though is associated closely with the service lay

The domain objects are acted on by the code in the service layer, the heart of the application. They also are used in the DAO layer, in a more technical way, to allow the data to be persisted in the database

Now we see we have Java packages for each layer: presentation, service, dao
domain package: objects for PizzaOrder, etc.
config package: setting up the system

Real pizza1 pom.xml

We see, compared to the prototype pom.xml:

- Similar elements at the start of the file, identifying the project
- Many more dependencies:
  - One for each of our databases, for the needed drivers
  - Oracle needs an entry in <repositories> because the driver is not in the standard Maven repository
  - JUnit5 instead of JUnit4: this is to get ready for Spring Boot 2, which assumes v5 unless you override it
- Different plugins, that is, tools to build or execute things
  - Two plugin version specs from the proto, for maven-compile and maven-surefire (for Junit tests)
  - There's an additional plugin spec for building a "fat jar", that is, a jar with all the dependencies in it.

Dependencies: see structure of multiple dependencies inside one collection element:

```
<dependencies>
   <dependency> …
   </dependency>
   <dependency> …
   </dependency>
   ...
<dependencies>
```

- From https://maven.apache.org/pom.html#Quick_Overview: The cornerstone of the POM is its dependency list. Most projects depend on others to build and run correctly. If all Maven does for you is manage this list, you have gained a lot
- This project needs to use the JUnit library version 4.12 for unit testing, a "dependency" JUnit in turn needs other libraries, but maven takes care of that for us.

Pizza1: The layers and their APIs

- **Recall that the layers are “call-down” layers**: the presentation layer code calls the service API and thus into the service layer. The service layer code calls the DAO API, and the DAO code calls JDBC for pizza1.
- The presentation layer code never calls the DAO directly.
- And nothing ever calls methods of a higher layer. So this is strict layering. That helps with understanding the system and debugging it.
- The service layer API is the most important one, defining the actions of the application.
- We can extract the API by using the fact that all these methods are public, and very little else in the Java source is public. We just cd to the directory of the service package and search the files using "find" or "grep".

The layers and their APIs

- We can extract the service API by using the fact that all these methods are public, and very little else in the Java source is public. We just cd to the directory of the service package and search the files using "find" or "grep".
- The service package sources are in directory pizza1/src/main/java/cs636/pizza/service

F:\cs\cs636> cd pizza1/src/main/java/cs636/pizza/service
See the Service API plus a few other lines—

C:\cs\cs636\pizza1\src\cs636\pizza\service>find "public" *Service.java

- Or on pe07, where pizza1 is in the class website directory /data/htdocs/cs636:

pe07$ cd /data/htdocs/cs636/pizza1/src/main/java/cs636/pizza/service

pe07$ grep public *.java

Service API display (partial)

```
F:\cs\cs636\pizza1\src\main\java\cs636\pizza\service>find "public" *Service.java
---------- ADMINSERVICE.JAVA
public class AdminService {
	public AdminService(DbDAO db, AdminDAO admDAO,PizzaOrderDAO poDAO,  MenuDAO mDAO)
{
	public void initializeDb() throws ServiceException {
	public void addTopping(String name) throws ServiceException {
	public void removeTopping(String topping) throws ServiceException {
	public void addSize(String name) throws ServiceException {
	public void removeSize(String size) throws ServiceException {
	public void markNextOrderReady() throws ServiceException {
```

Continued: StudentService (full list)

```
---------- STUDENTSERVICE.JAVA
public class StudentService {
        public StudentService(PizzaOrderDAO pizzaDAO, MenuDAO mDAO, AdminDAO admDAO) {
        public Set<String> getSizeNames()throws ServiceException
        public Set<String> getToppingNames()throws ServiceException
        public void makeOrder(int roomNum, String sizeName, Set<String> toppingNames)
        public List<PizzaOrderData> getOrderStatus(int roomNumber) throws ServiceException {
        public void receiveOrders(int roomNumber)throws ServiceException {
```

The method names (in bold) give names for the actions here, the student-related actions of the app

StudentService actions in use

Student orders a pizza:

- First we need to show the possible toppings and sizes: call getSizeNames and getToppingNames
- Then the student selects what they want and makes to order: call makeOrder

Student checks the status of an order

- Call getOrderStatus

Student acknowledges receipt of pizza(s)

- Call receiveOrders

- Where is this code that does these calls?

Understanding the layers in use

Answer: in the presentation layer, in the presentation package.

How does makeOrder do its work, that is, get an order into the database?

The layers in use

Answer: by calling methods in the DAO layer, i.e., calling the DAO API

How can we find the DAO API?

Answer: by cd ../dao, search again for “public”

Does presentation have an API?

Answer: not in the same sense—it’s on top, taking orders from the user, i.e., the UI.

The types seen in the service API

```
public Set<String> getSizeNames()throws ServiceException
        public Set<String> getToppingNames()throws ServiceException
        public void makeOrder(int roomNum, String sizeName, Set<String> toppingNames)
        public List<PizzaOrderData> getOrderStatus(int roomNumber) throws ServiceException {
        public void receiveOrders(int roomNumber)throws ServiceException {
```

- The topping names and size names are provided to the presentation layer as simple Strings. That way, no domain objects are “leaked” into the UI code in presentation.
- The PizzaOrder information is provided using PizzaOrderData objects created from PizzaOrder objects just for the purpose of carrying the data to the presentation code.
- Each method throws ServiceException, an application-defined subclass of Exception—we should explore this further soon

More on call-down layers

Why would you want to call upwards?

- For notifications. Consider when the pizza is ready: the admin action “this pizza is ready” calls down and changes the database. If this change triggered a *notification* to the student user, that would need an **upcall** to the presentation layer.
- Such notifications can be done in say a Swing application (Java GUI program running on the user’s machine), because *the Swing app owns the screen*. It can pop up a little note on the screen.
- But we are designing for an (eventual) web app, where the service layer runs in the server and *doesn’t own the clients’ screens*. The clients use HTTP to ask questions and make commands.
- So the client has to explicitly ask “is the pizza done?” and this request is handled in a call-down manner.

Web apps using HTTP are request-driven

- That's the simple web app way. We could explore ways to notify (WebSockets for example), but it is quite difficult and requires app code on the client side, usually Javascript. We need to keep things simple to get where we want to go in one term.
- Consider that from the server's standpoint, the clients are often hidden behind firewalls—how can you send them information at some arbitrary moment?
- They don't have well-known network addresses! The chance to send them information is when they send a request in to you (the server) and you get to reply to it.
- Therefore, we will stick to the simple call-down layers.
- Next time--look at database for pizza1.

Eclipse and Maven

- When you create a new eclipse project, say with Open Project from File System, it recognizes the Maven directory structure and pom.xml
- You see a little M on the project, as well as a little J for Java
- Note that every time we edit pom.xml, we need to tell eclipse by right-clicking the project's icon in Project Explorer, selecting Maven and then Update Project.
- Eclipse has a special UI for the pom, showing the dependency hierarchy and the effective pom, as well as the plain source file
- Eclipse uses the local repository of jar files for populating the classpath for the project