Consistency in OSS Libraries:
Google’s Approach

Open Source Summit Japan 2019
Garrett Jones and Tomohiro Suzuki
About us

Garrett Jones
Staff Software Engineer
Google Seattle

Tomohiro Suzuki
Software Engineer
Google New York
Agenda

- Intro to diamond dependency conflicts
- Google's Java Library Best Practices
- Linkage Checker
- Q&A
Story

- Google monorepo vs OSS independent libraries
- Megathread prompted by user complaints
- No consensus
- Proposals, summits, proposals, hackathons
- Wrote best practices, created tools
Diamond Dependency Conflicts
Diamond dependency conflicts: A visual representation

Linear dependency graph: everything is happy

Legend

Library A, version 1
Diamond dependency conflicts: A visual representation

D introduces a new major version (D:2)
Diamond dependency conflicts: A visual representation

Another package, C, declares a dependency on D:2
A new version of A attempts to add C as a dependency. Diamond dependency conflict! Only one of D:1 or D:2 can be chosen.
Diamond dependency conflicts: A visual representation

When D:2 is selected, this breaks B
Diamond dependency conflicts: A visual representation

When D:1 is selected, this breaks C
Diamond dependency conflicts: A visual representation

The only solution: A has to force B to make a new version that depends on D:2
Google's approach

1) Fix burning conflicts
2) Establish best practices for library developers*
3) Create tools*
4) Create BOMs (bill of materials) for library users
Java Library Best Practices
(for library developers)
Google's Java library best practices (JLBP)

- I will cover only 5 here
JLBP-1: Minimize dependencies
JLBP-2: Minimize API surface
JLBP-3: Use Semantic versioning
JLBP-4: Avoid dependencies on unstable libraries and features

**JLBP-5: Avoid dependencies that overlap classes with other dependencies**

JLBP-6: Rename artifacts and packages together
JLBP-7: Make breaking transitions easy

**JLBP-8: Advance widely used functionality to a stable version**

JLBP-9: Support the minimum Java version of your consumers
JLBP-10: Maintain API stability as long as needed for consumers
JLBP-11: Stay up to date with compatible dependencies
JLBP-12: Make level of support and API stability clear

**JLBP-13: Quickly remove references to deprecated features in dependencies**

JLBP-14: Do not use version ranges
JLBP-15: Produce a BOM for multi-module projects
JLBP-16: Ensure upper version alignment of dependencies for consumers
JLBP-17: Coordinate Rollout of Breaking Changes

**JLBP-18: Only shade dependencies as a last resort**
Problem 1: "Overlapping" classes
Problem 1: "Overlapping" classes
Problem 1: "Overlapping" classes

A

class X

B

C
Problem 1: "Overlapping" classes

- Class X
- Class Y
- Class C
- Class A
Problem 1: "Overlapping" classes

Diagram:

- A
  - class X
  - B
    - class Y
  - C
    - class Z
Problem 1: "Overlapping" classes

Diagram:

- A
  - class X
  - B
    - class Y
      - class O
  - C
    - class Z
      - class O
Problem 1: "Overlapping" classes

Classes loaded:
- class X [from library A]
- class Y [from library B]
- class O [from library B]
- class Z [from library C]
Problem 1: "Overlapping" classes

Classes loaded:
- class X [from library A]
- class Y [from library B]
- class O [from library B]
- class Z [from library C]

Classes not loaded:
- class O [from library C]
Problem 1: "Overlapping" classes

Classes loaded:
- class X [from library A]
- class Y [from library B]
- class O [from library B]
- class Z [from library C]

Classes not loaded:
- class O [from library C]

What can break?
- calls from Z to O (for things not in the version of O included in B)
What needs to be true?

Every fully-qualified class name must be present in only one library in the dependency tree.
JLBP-5: Avoid dependencies that overlap classes with other dependencies
JLBP-6: Rename artifacts and packages together
Example violation of JLBP-5 & JLBP-6

java.x.servlet:javax.servlet-api:3.1.0
& javax.servlet:servlet-api:2.5

both contain classes with the same names under javax.servlet.
JLBP-6 corollaries

- Don't combine artifacts together while keeping the same fully-qualified names
- Don't split artifacts apart while keeping the same fully-qualified names
Problem 2: Wide scale breakage from changes
Problem 2: Wide scale breakage from changes
Problem 2: Wide scale breakage from changes
Problem 2: Wide scale breakage from changes

Diagram:

- A
- B
- C
- D
- E
- F
- G
- guava
Problem 2: Wide scale breakage from changes
Problem 2: Wide scale breakage from changes
JLBP-8: Advance widely used functionality to a stable version

(also follow JLBP-3: Use Semantic Versioning - "stable version" = 1.0.0+)
Example violation of JLBP-8

cm.google.auth:google-auth-library-java: still uses a 0.x version
Problem 3: finding compatible dependencies

1.0

A

X

70.0

B

Y
Problem 3: finding compatible dependencies

1.0

A

X

B

Y

70.0

2.0

A

X

B

¥ Z

71.0
Problem 3: finding compatible dependencies

1.0

A

X

B

Y

70.0

2.0

A

X

B

￥ Z

71.0

3.0

A

X

B

￥ Z

72.0
Problem 3: finding compatible dependencies

1.0
A
X
B
Y
70.0

2.0
A
X
B
Y
Z
71.0

3.0
A
X
B
Y
Z
72.0

4.0
A
X
B
Y
Z
73.0
Problem 3: finding compatible dependencies

1.0
A
X

2.0
A
X

3.0
A
X

4.0
A
X

5.0
A
X

70.0
B
Y

71.0
B
¥Z

72.0
B
¥Z

73.0
B
¥Z

74.0
B
Z
Problem 3: finding compatible dependencies

For the latest A (5.0)

A:1.0
A:2.0
A:3.0
A:4.0
A:5.0

B:70.0
B:71.0
B:72.0
B:73.0
B:74.0

The latest A (5.0) can use the last 4 versions of B.

This is nice and flexible.
Problem 3: finding compatible dependencies

For the latest B (74.0)

- A:1.0
- A:2.0
- A:3.0
- A:4.0
- A:5.0

- B:70.0
- B:71.0
- B:72.0
- B:73.0
- B:74.0

The latest B (74.0) can only use the last version of A (5.0).

This is somewhat restrictive!
Problem 3: finding compatible dependencies
Problem 3: finding compatible dependencies

1.0

A

X

B

Y

70.0

2.0

A

X

B

¥ Z

71.0
Problem 3: finding compatible dependencies

1.0

A

X

B

Y

2.0

A

X

B

¥ Z

3.0

A

X

B

¥ Z
Problem 3: finding compatible dependencies

1.0

A

X

B

Y

70.0

2.0

A

X

B

¥ Z

71.0

3.0

A

X

B

¥ Z

72.0

4.0

A

X

B

¥ Z

73.0
Problem 3: finding compatible dependencies
Problem 3: finding compatible dependencies

For the latest A (5.0)

A:1.0  B:70.0
A:2.0  B:71.0
A:3.0  B:72.0
A:4.0  B:73.0
A:5.0  B:74.0

The latest A (5.0) can still use the last 4 versions of B.
No regression, things are going good.
Problem 3: finding compatible dependencies

For the latest B (74.0)

The latest B (74.0) can be used with the last 4 versions of A.

Great improvement! This is a lot more flexible.
JLBP-13: Quickly remove references to deprecated features in dependencies
Example violation of JLBP-13

com.google.api:api-common-java: usage of deprecated methods in Guava removed 1 year + 3 months after deprecation (1.7.0), instead of earlier (e.g. 1.2.0)
What about shading?
What about shading?
What about shading?

A B

A C
What about shading?
What about shading?

```
A B
A C
A D
```

```
A B
A C
A E
```
What about shading?
What about shading?
What about shading?
What about shading?
What about shading?

No shading: 8 units

With shading: 54 units (27 of which are copies of A)

$\frac{54}{8} = 6.75 \times \text{the size!}$
What about shading?

Other problems:

- Bad shading config can create overlapping classes or missing classes
- Shaded dependencies can't be overridden to roll out security fixes
- Shading doesn't work well with JNI or reflection
JLBP-18: Only shade dependencies as a last resort
Consistency in OSS Libraries: Google’s Approach

Linkage Checker

github.com/GoogleCloudPlatform/cloud-opensource-java

Tomohiro Suzuki (suztomo@google.com)
Agenda: Linkage Checker

• Demo
• Implementation
• Case Study
Linkage Checker: Demo
Demo: Linkage Checker Maven Enforcer Rule

Problem: a simple project having dependency conflicts

Linkage Checker Enforcer Rule detects the conflict

Project

- google-api-client
- grpc-core

- guava version: 20.0
- guava version: 26.0
Demo: Linkage Checker Maven Enforcer Rule

A simple project with two dependencies

```xml
<dependencies>
  <dependency>
    <groupId>com.google.api-client</groupId>
    <artifactId>google-api-client</artifactId>
    <version>1.27.0</version>
  </dependency>
  <dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-core</artifactId>
    <version>1.17.1</version>
  </dependency>
</dependencies>
```
No issue in compile
Demo: Linkage Checker Maven Enforcer Rule

Runtime Error!

```
suoztom@suxto24:/<cloud-opensource-java/example-problems/no-such-method-error-signature-mismatch>$ mvn exec:java
[INFO] Scanning for projects...
[INFO]
[INFO] ---< com.google.cloud.tools.opensource:no-such-method-error-example >---
[INFO] Building no-such-method-error-example 1.0-SNAPSHOT
[INFO] ------------------------------------------[ jar ]------------------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ no-such-method-error-example ---
  at io.grpc.internal.DnsNameResolver.maybeChooseServiceConfig (DnsNameResolver.java:514)
  at io.grpc.internal.App.main (App.java:31)
  at sun.reflect.NativeMethodAccessorImpl.invoke0 (Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke (NativeMethodAccessorImpl.java:62)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke (DelegatingMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke (Method.java:498)
  at org.codehaus.mojo.exec.ExecJavaMojo$1.run (ExecJavaMojo.java:282)
  at java.lang.Thread.run (Thread.java:748)
```

Google
Q: Why java compiler does not detect the conflict?

A: the compiler only checks the references from the project

```
App.java
DnsNameResolver.class
google-api-client
grpc-core
```

```
guava version: 20.0
Verify.class
```

```
guava version: 26.0
```
Demo: Linkage Checker Maven Enforcer Rule

Add the enforcer rule

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-enforcer-plugin</artifactId>
      <version>3.0.0-M2</version>
      <dependencies>
        <dependency>
          <groupId>com.google.cloud.tools</groupId>
          <artifactId>linkage-checker-enforcer-rules</artifactId>
          <version>0.2.1</version>
        </dependency>
      </dependencies>
    </plugin>
  </plugins>
  <executions>
    <execution>
      <id>enforce</id>
      <phase>verify</phase>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration>
        <rules>
          <LinkageCheckerRule
            implementation="com.google.cloud.tools.dependencies.enforcer.LinkageCheckerRule"/>
        </rules>
      </configuration>
    </execution>
  </executions>
</build>
```
Demo: Linkage Checker Maven Enforcer Rule

$ mvn install

[INFO] --- maven-enforcer-plugin:3.0.0-M2:enforce (enforce) @ no-such-method-error-example ---
[ERROR] Linkage Checker rule found 8 errors. Linkage error report:  
Class org.apache.log4j.Priority is not found;  
  referenced by 1 class file  
Class org.apache.log4j.Logger is not found;  
  referenced by 1 class file  
Class org.apache.log4j.Level is not found;  
  referenced by 1 class file  
Class javax.servlet.ServletContextListener is not found;  
  referenced by 1 class file  
Class org.apache.log.Hierarchy is not found;  
  referenced by 1 class file  
Class org.apache.log.Logger is not found;  
  referenced by 1 class file  
(gruava-20.0.jar) com.google.common.base.Verify's method verify(boolean arg1, String arg2, Object arg3) is not found;  
  referenced by 3 class files
Demo: Google Libraries BOM Dashboard

Checks compatibility of artifacts in BOM (a set of libraries)

com.google.cloud:libraries-bom:2.1.0-SNAPSHOT Dependency Status

- **209** TOTAL ARTIFACTS CHECKED
- **10** HAVE LINKAGE ERRORS
- **5** HAVE UPPER BOUNDS ERRORS
- **205** HAVE GLOBAL UPPER BOUNDS ERRORS
- **202** FAIL TO CONVERGE

https://github.com/GoogleCloudPlatform/cloud-opensource-java#google-cloud-platform-java-dependency-dashboard
Linkage Checker: Implementation
Implementation Step 1: Dependency Tree

Creates dependency tree of Maven artifacts (Maven’s dependency mediation)
Implementation Step 2: JAR, Class files, and Constant Pool

Extracts references from constant pool

Example:

io.grpc.internal.DnsNameResolver constant pool:
- class io.grpc.NameResolver
  ...
- method: “com.google.base.Verify.verify(bool, String, Object)”
Implementation Step 3: Verification of Referents

Verifies the referents of the references

Does the referent (class B) exist?
Does the referent have the method with the expected signature?
Does the method accessible from class A?
etc...

Class A
Constant Pool:
...  
  - ClassB.methodX(String, Object)

Class B
public methodX(String, Object)
private methodY(String)
(default) methodZ()
Case Study
Case 1: Missing Class

JLB-18: Only shade dependencies as a last resort
Case 2: Missing Method

Runtime Error from the demo

NoSuchMethodError Verify.verify(ZLjava/lang/String;Ljava/lang/Object;)V
=> void verify(boolean, String, Object)
Case 2: Missing Method

Project

- google-api-client
  - guava version: 20.0
    - Verify.class
      - public verify(boolean, String, Object...)

- grpc-core
  - DnsNameResolver.class
    - guava version: 26.0
      - Verify.class
        - public verify(boolean, String, Object)

Java Language Specification Chapter 13: Binary Compatibility
Case 3: Missing Constructor

spring-cloud-gcp-starter-pubsub doesn't work with spring-cloud-gcp-starter-trace version higher than 1.0.0.RC

#1373

Closed shiraji opened this issue on Jan 11 · 16 comments

shiraji commented on Jan 11

Hi, there

We have been struggling with spring-cloud-gcp-starter-trace.

If we include the dependency of spring-cloud-gcp-starter-trace version higher than 1.0.0.RC, the app does not receive any Cloud Pub/Sub messages.

Here is the minimum sample project that reproduce this problem. https://github.com/shiraji/demo-pubsub

This project does not work with Cloud Pub/Sub in version 1.1.0.RC2. However, once we remove the dependency of spring-cloud-gcp-starter-trace, this project works perfectly.

Also, if we use 1.0.0.RC for all spring-cloud-gcp-starter's, the project works nicely.
Case 3: Missing Constructor

class AbstractClientStream{
    super(headers);
    ...
}

class NettyClientStream extends AbstractClientStream{
    super(headers);
    ...
}

class AbstractClientStream{
    AbstractClientStream(Headers Option) {
        ...
    }
}
Conclusion

Diamond dependency issues:
Dependency tree generated by different libraries may have conflicts.

- Java Library Best Practices
- Linkage Checker

https://github.com/GoogleCloudPlatform/cloud-opensource-java/