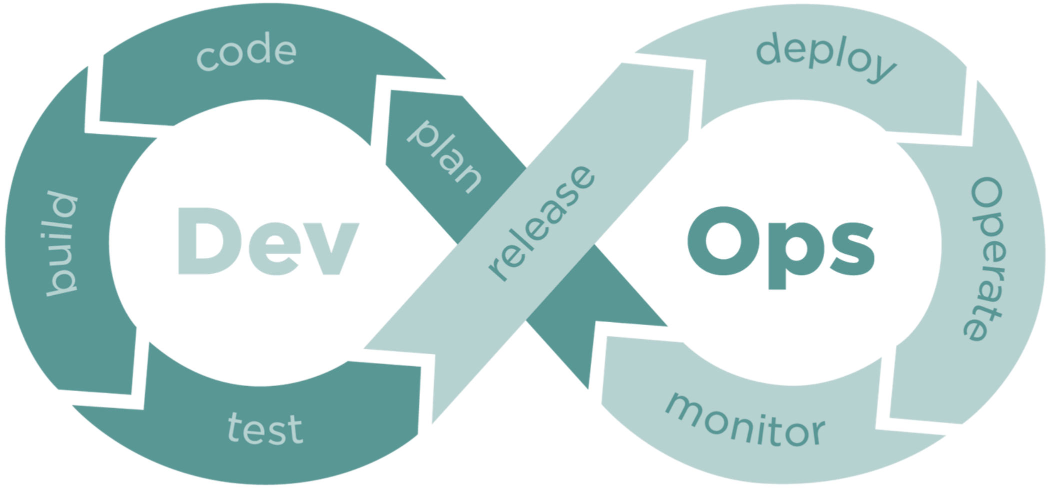
CS304 SOFTWARE ENGINEERING

Yida Tao

taoyd@sustech.edu.cn

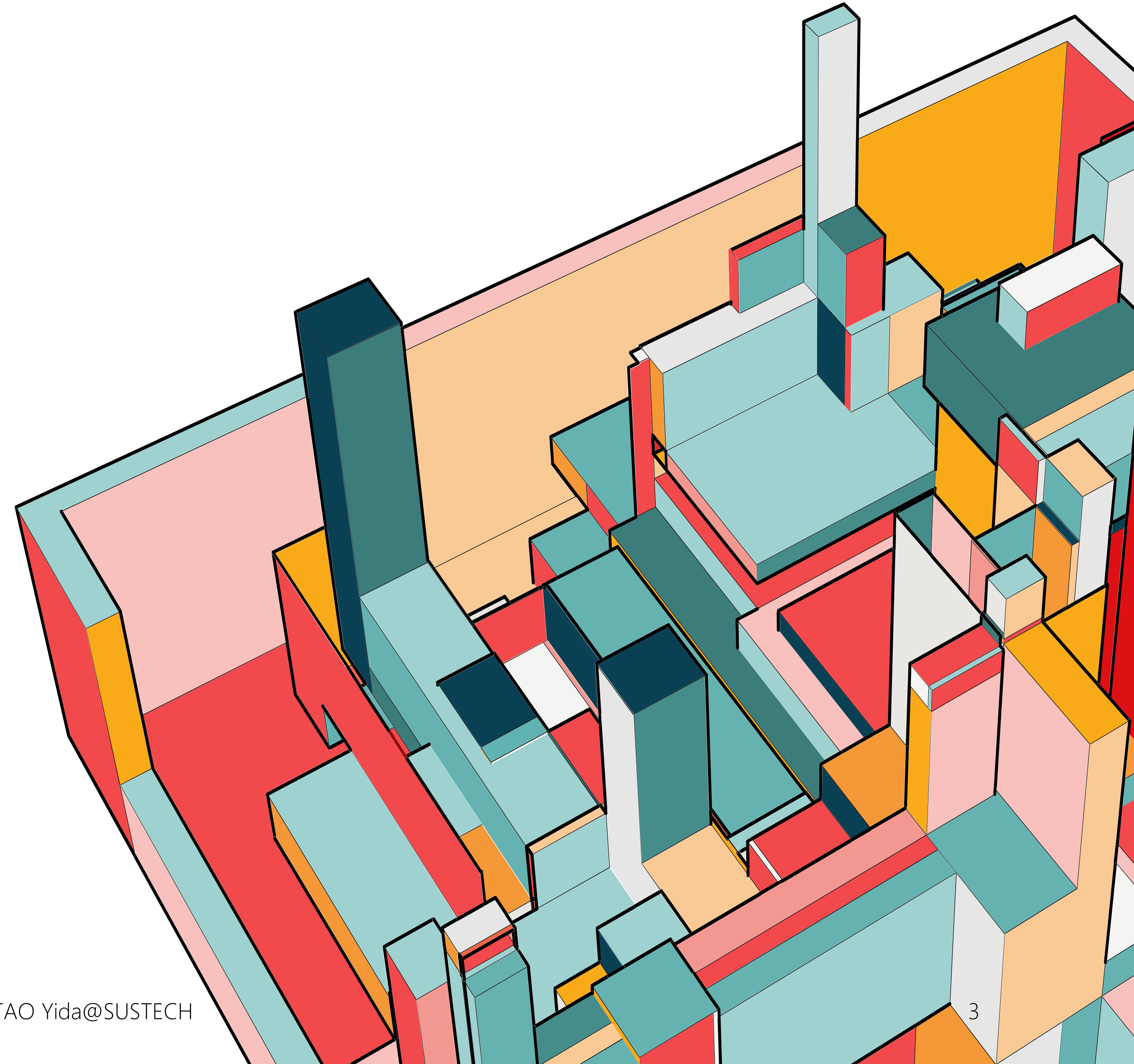
WHERE ARE WE NOW?

Build



LECTURE 6

- Build system overviews
- Types of build systems
- Build artifacts
- Managing dependencies



BUILD SYSTEMS

Fundamentally, all build systems have a straightforward purpose: transforming the source code into executable binaries

Compilers?

COMPILERS MAY NOT BE ENOUGH

```
>>>    javac  *.java
```

- What if code are stored in other parts of the filesystem?
- What if code depends on 3rd party JAR files?
- What if the dependent JARs become outdated?
- What if the dependency has orders?
- What if the system is written in various programming languages?
- What if the system needs proper configuration files to start?
- ……

HOW ABOUT SHELL SCRIPT?

Imagine the monster this script will become when software becomes larger and more complex……

```bash
#!/bin/bash

# 设置变量
SOURCE_DIR="src"
OUTPUT_DIR="build"
MAIN_CLASS="com.example.Main"
CLASSPATH="lib/*:$OUTPUT_DIR"

# 创建输出目录
mkdir -p $OUTPUT_DIR

# 编译Java源代码
echo "Compiling Java source code..."
javac -d $OUTPUT_DIR -cp $CLASSPATH $SOURCE_DIR/*.java

# 检查编译是否成功
if [ $? -eq 0 ]; then
    echo "Compilation successful."

    # 创建可执行JAR文件
    echo "Creating executable JAR file..."
    jar cfe $OUTPUT_DIR/my_program.jar $MAIN_CLASS -C $OUTPUT_DIR .

    # 检查JAR文件创建是否成功
    if [ $? -eq 0 ]; then
        echo "JAR creation successful. Executable: $OUTPUT_DIR/my_program.jar"
    else
        echo "JAR creation failed. Check for errors."
    fi
else
    echo "Compilation failed. Check for errors."
fi
```

BUILD SYSTEMS

- Building is the process of creating a complete, executable software by **compiling** and **linking** the software components, external libraries, configuration files, etc.

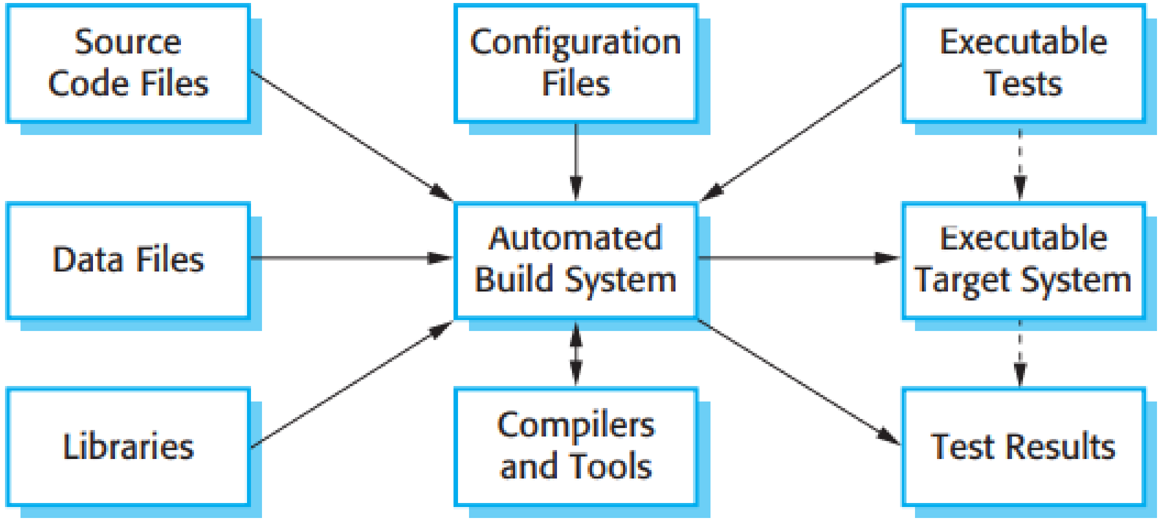
- Building involves assembling a large amount of info about the software and its operating environment.

BUILD SYSTEMS

For anything apart from very small systems, it always makes sense to use an automated build tool to create a system build

Source code might have different versions

Data files might be scattered in file systems



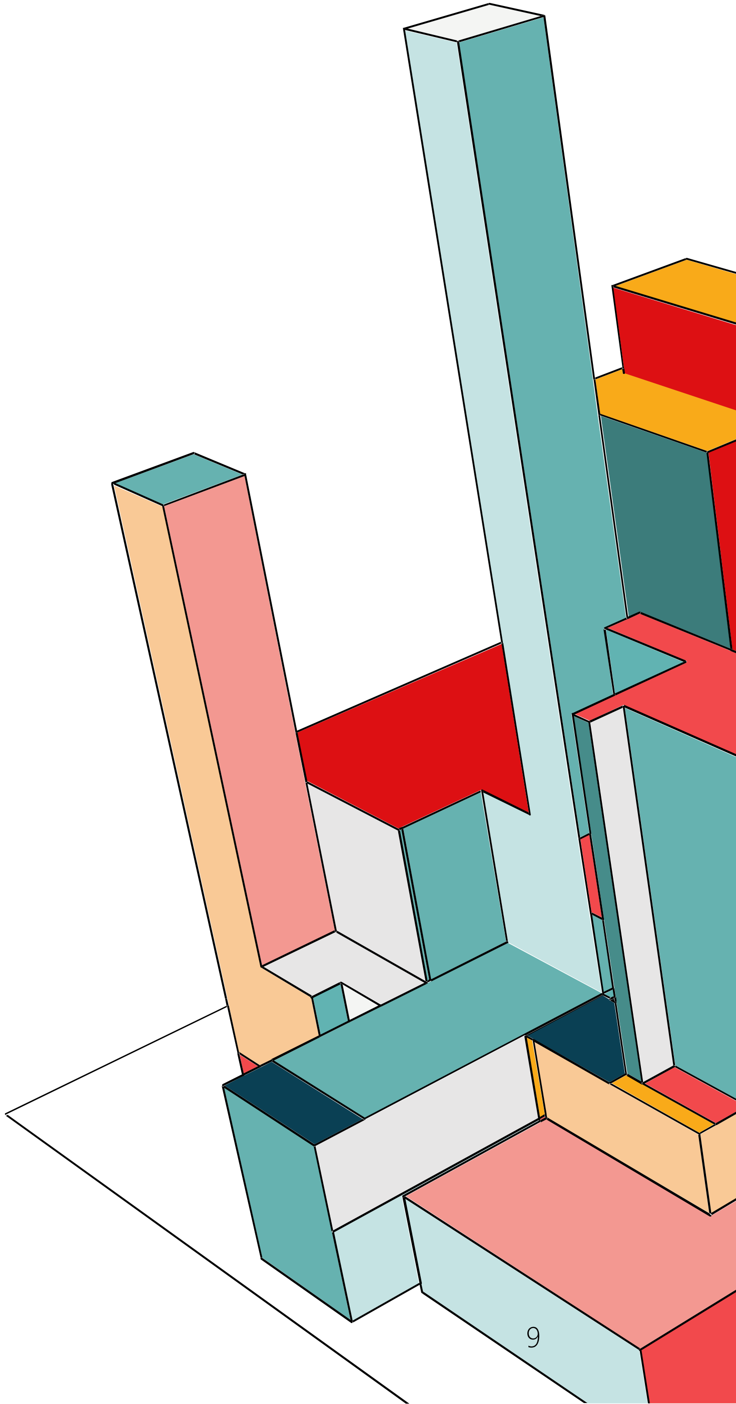
Tests may depend on certain frameworks

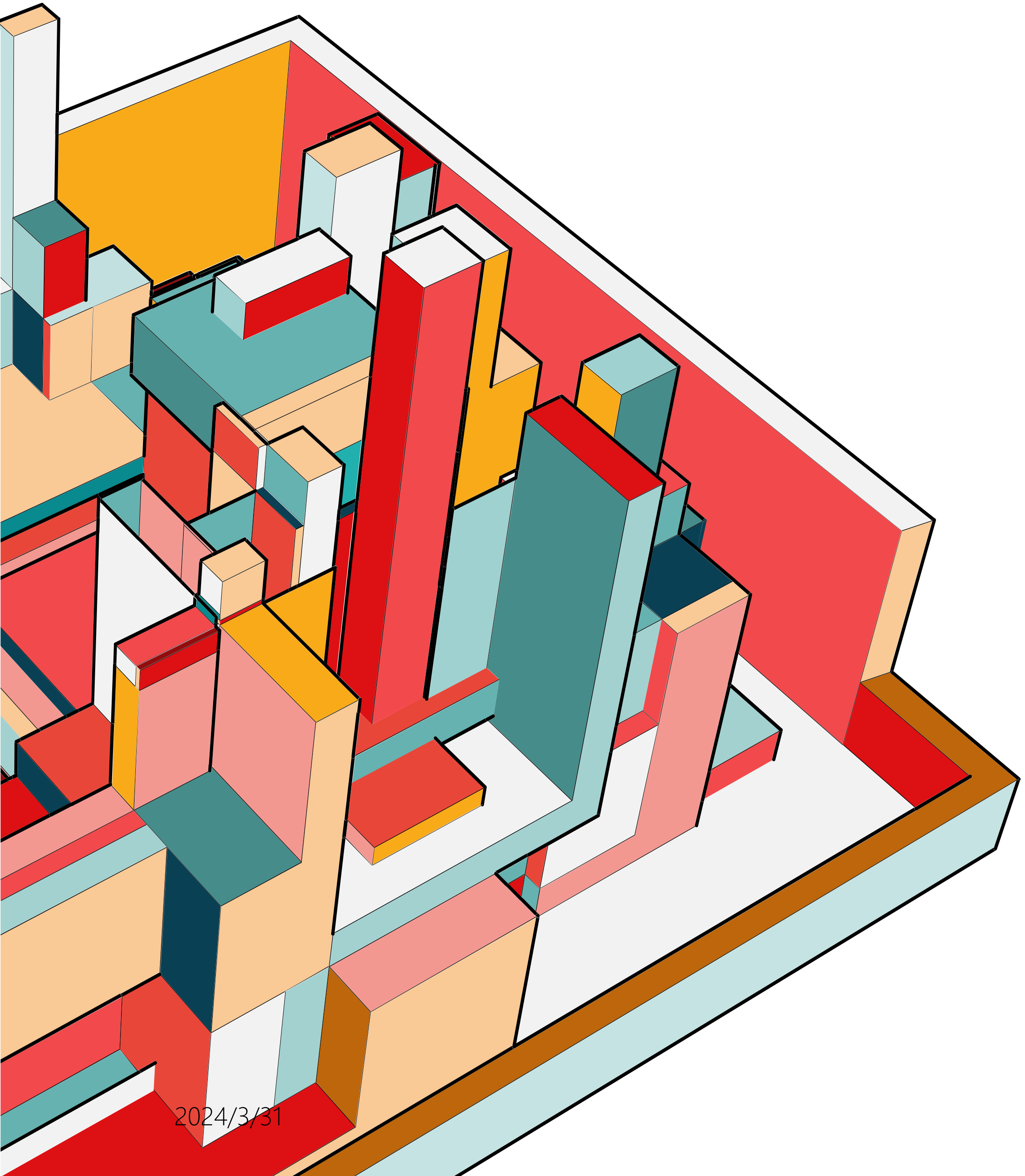
External libraries may have conflict dependencies and may be outdated

Code might be written in different languages

TYPES OF BUILD SYSTEMS

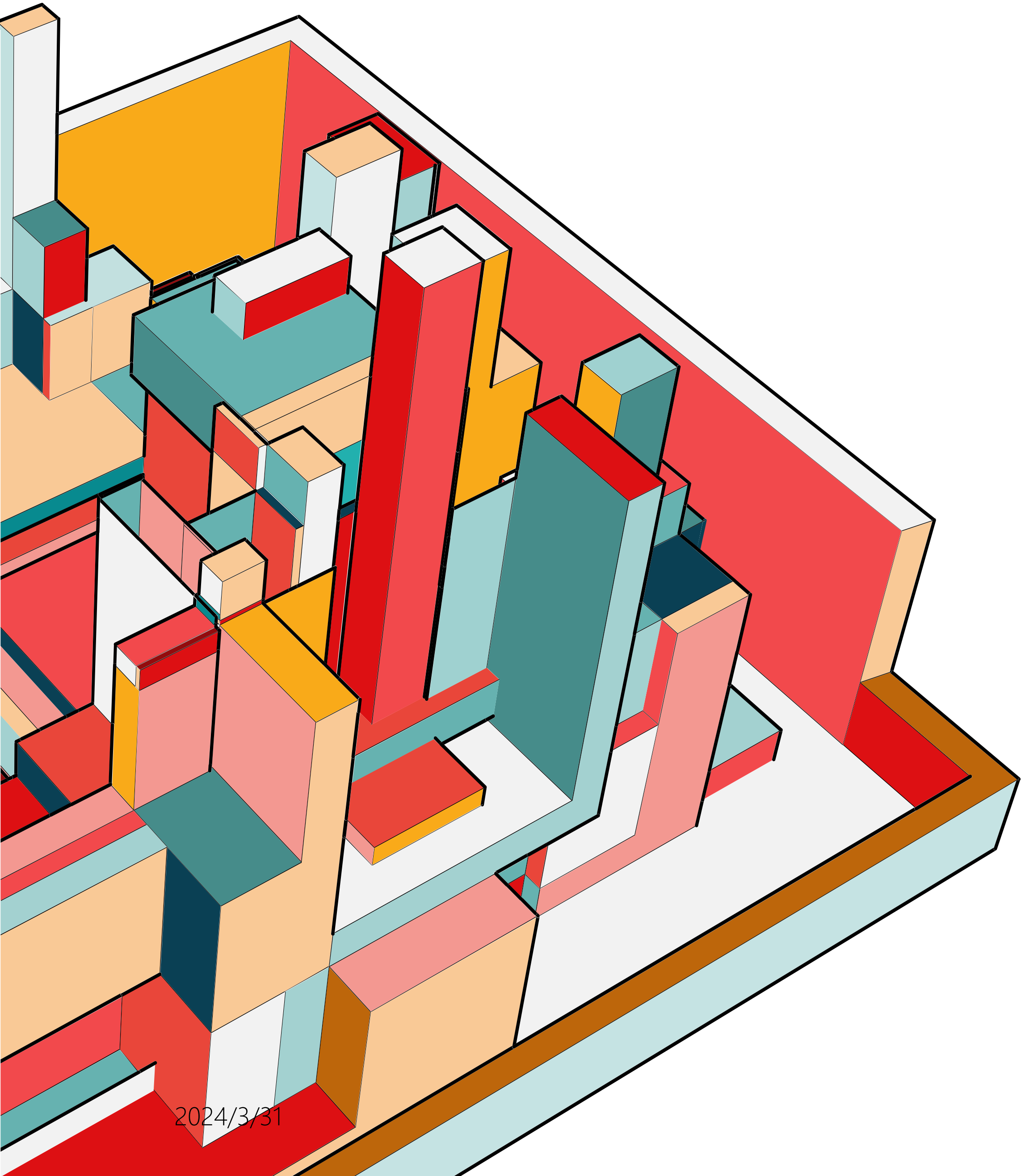
- Task-based Build Systems
- Artifact-based Build Systems





TASK-BASED BUILD SYSTEMS

- In a task-based build system, the fundamental unit of work is the task
- Each task is a script of some sort that can execute any sort of logics, and tasks can specify other tasks as dependencies that must run before them

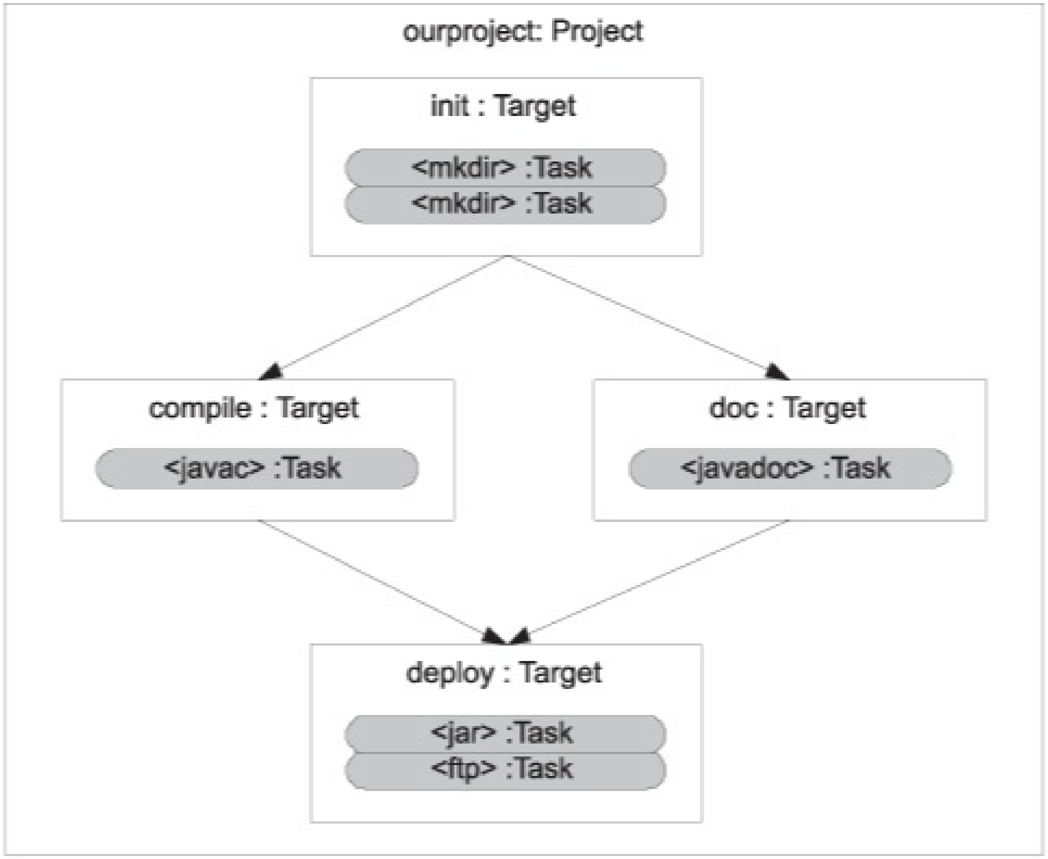


TASK-BASED BUILD SYSTEMS

- Most major build systems (e.g., Ant, Maven, Gradle, Grunt, and Rake), are task based

- Most modern build systems require engineers to create **buildfiles** that describe how to perform the build (e.g., `pom.xml` for Maven)

ANT: TASK DEPENDENCY & BUILDFILE



https://livebook.manning.com/book/ant-in-action/chapter-1/45

```
<?xml version="1.0" ?>
<project name="ourproject" default="deploy">

  <target name="init">
    <mkdir dir="build/classes" />
    <mkdir dir="dist" />
  </target>

  <target name="compile" depends="init">
    <javac srcdir="src"
      destdir="build/classes"/>
  </target>

  <target name="doc" depends="init" >
    <javadoc destdir="build/classes"
             sourcepath="src"

             packagenames="org.*" />
  </target>

  <target name="package" depends="compile,doc" >
    <jar destfile="dist/project.jar"
        basedir="build/classes"/>
  </target>

  <target name="deploy" depends="package" >
    <ftp server="${server.name}"
        userid="${ftp.username}"
        password="${ftp.password}">
      <fileset dir="dist"/>
    </ftp>
  </target>
</project>
```

Create two output directories for generated files

Compile the Java source

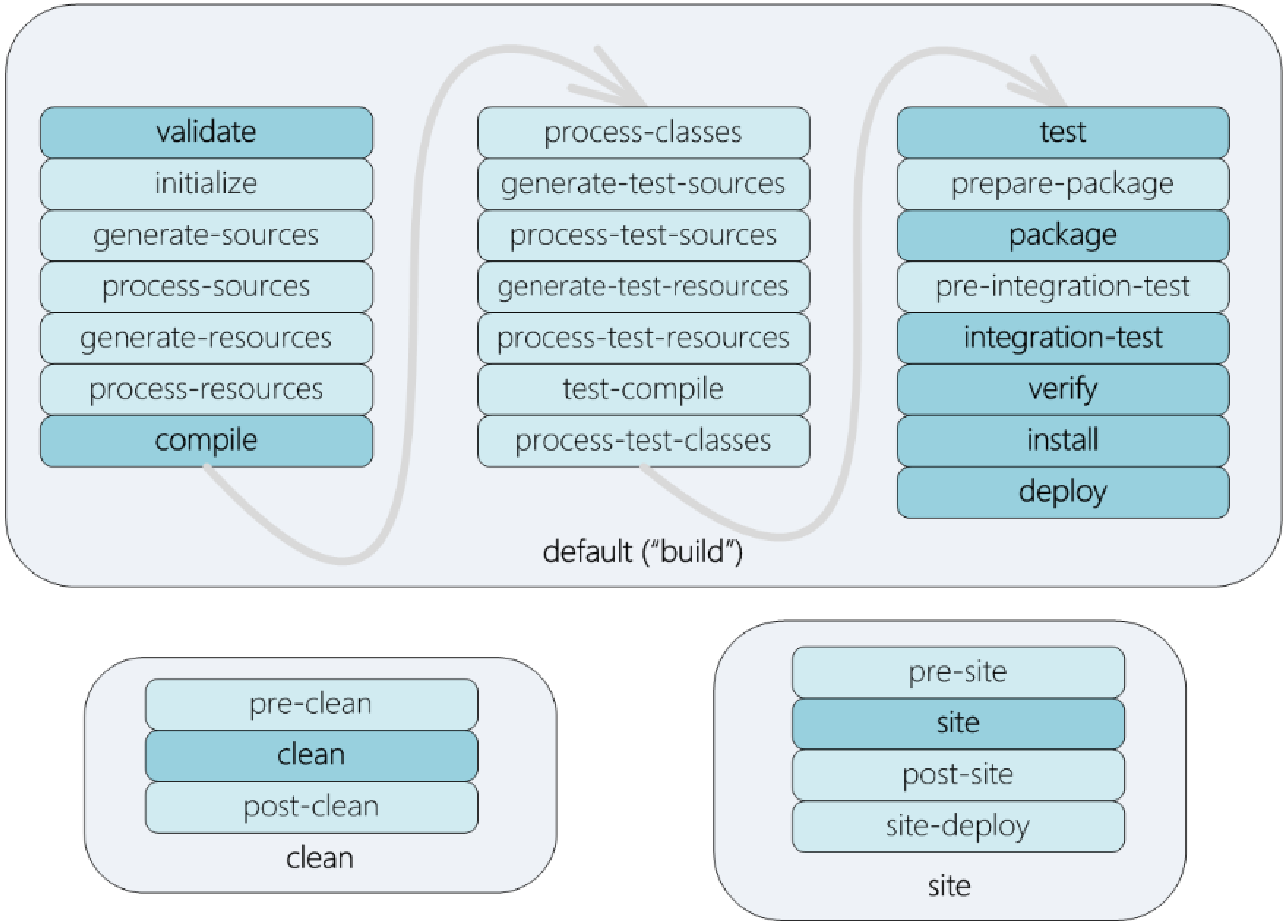
Create the javadocs of all org.* source files

Create a JAR file of everything in build/classes

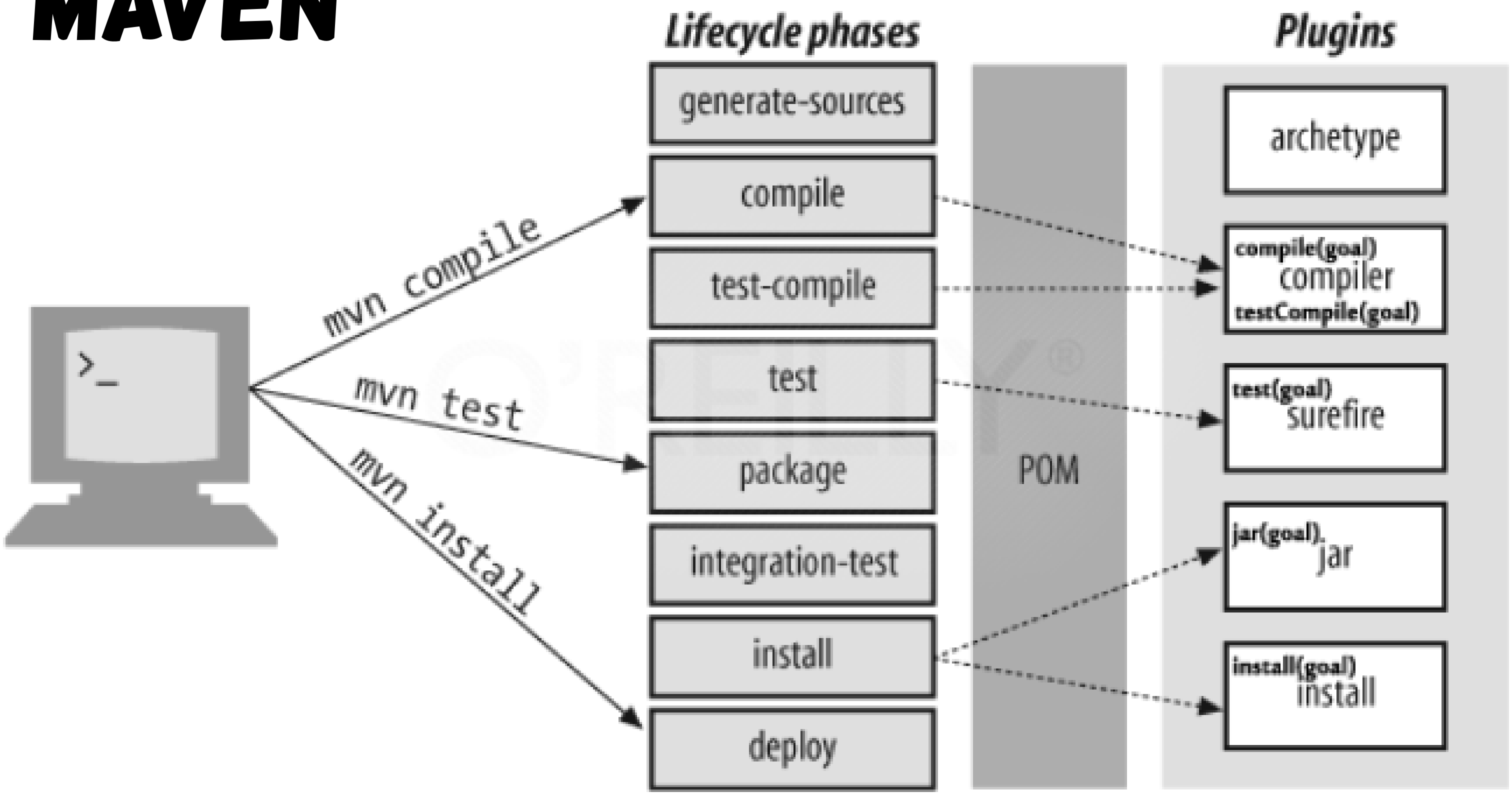
Upload all files in the dist directory to the ftp server

MAVEN

- **Build lifecycle**: an **ordered** list of build phases. Maven has 3 lifecycles
- **Build phase**: a set of build tasks
- We can execute a build phase, which runs all phases prior to it and the phase itself



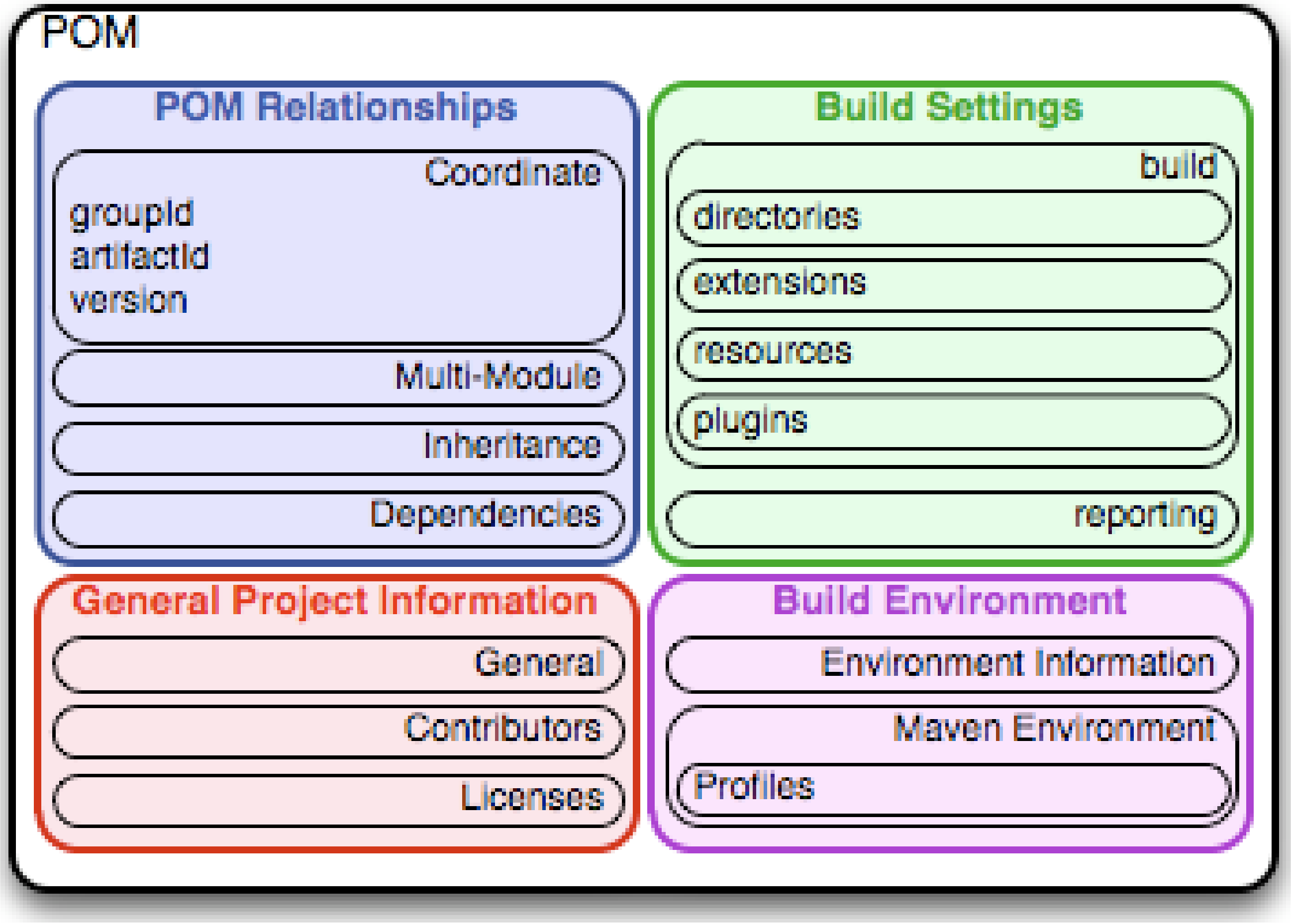
MAVEN



- **Plugin**: an artifact used for executing build tasks. A plugin provides one or more goals
- **Goals**: used to execute build tasks. Smallest step of maven builds
- Goals of plugins are attached to one or more phases
- We could also execute a plugin goal

MAVEN

The POM contains all necessary information about a project, as well as configurations of plugins to be used during the build process



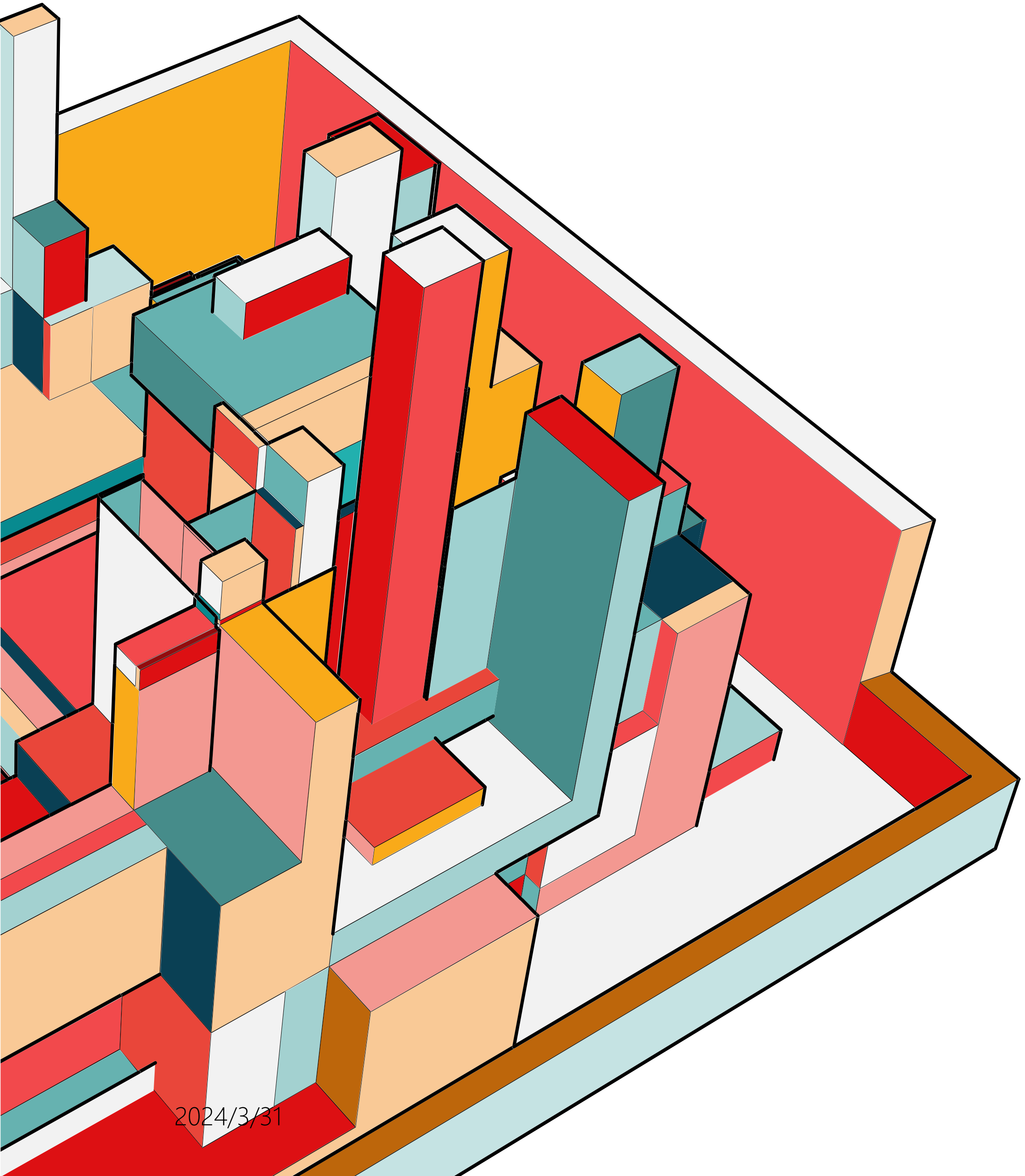
DRAWBACKS OF TASK-BASED BUILD SYSTEMS?

Difficulty maintaining & debugging build scripts

- Task-based build tools give too much power to engineers by allowing them to define any script as a task
- The scripts are easy places for bugs to hide and tend to grow in complexity
- The scripts end up being another thing that needs debugging & maintaining

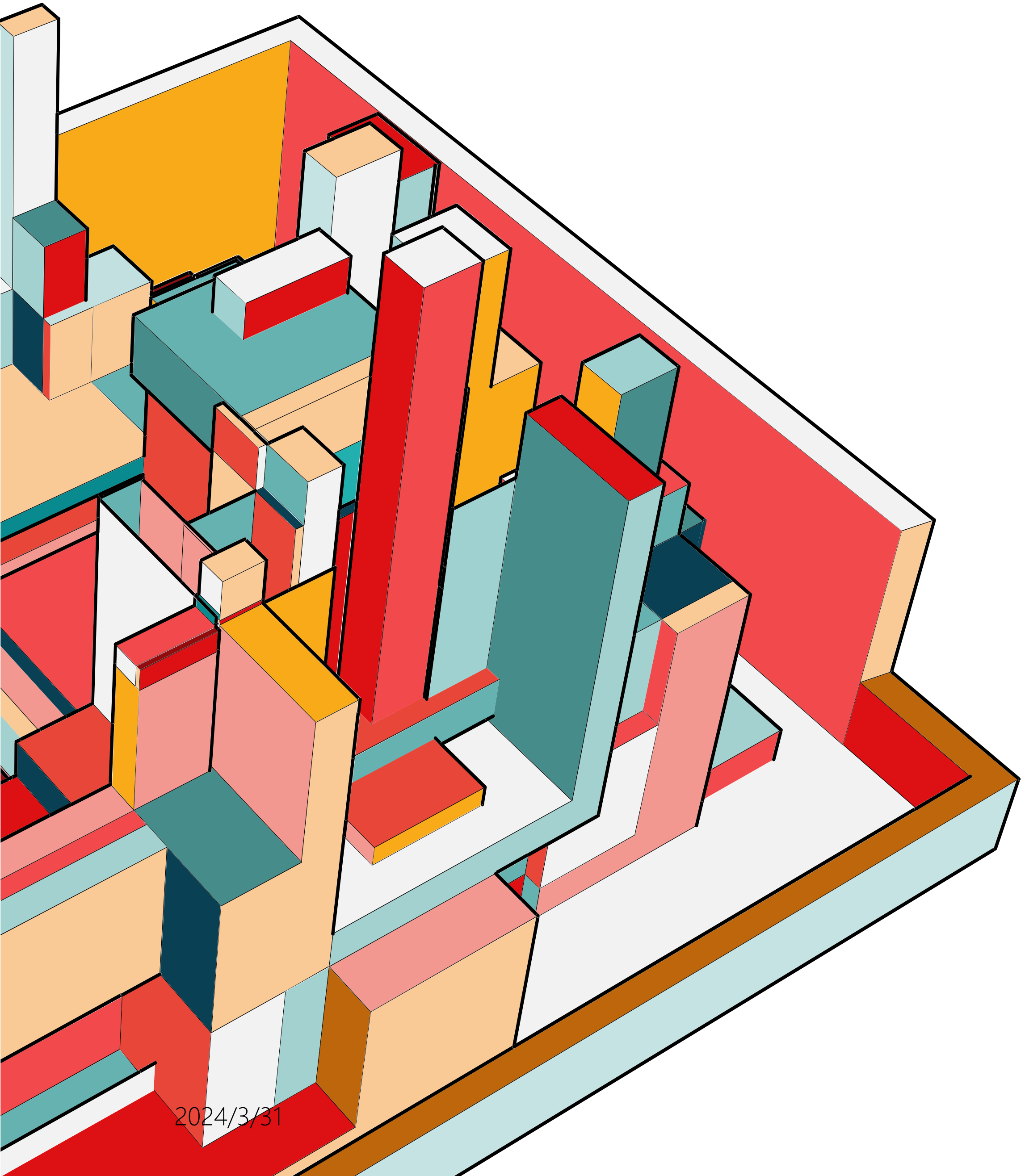
Difficulty performing incremental builds & parallelism

- Because of the flexibility of task-based build systems, it's hard to determine the change impact and side effects if anything is changed in the build process
- It's difficult to perform incremental builds and parallel builds



TASK-BASED BUILD SYSTEMS

- Engineers can write arbitrary code to execute any tasks during build
- Build systems can't have enough information/control to always be able to run builds quickly and correctly



ARTIFACT-BASED BUILD SYSTEMS

- In a build process, the role of the build system is **producing artifacts** (e.g., executable binary, documentation, etc.)
- Engineers still need to tell the system **what** to build, but **how** to do the build would be left to the build system
- The approach that Google takes with `Blaze` (internal version) and `Bazel` (open-source version)

BAZEL

- `Buildfiles` in artifact-based build systems like Bazel are a **declarative** manifest describing:
  - A set of artifacts to build
  - Their dependencies
  - Limited set of configurations

- Engineers run bazel by specifying a set of targets to build (the **“what”**)

- Bazel is responsible for configuring, running, and scheduling the compilation steps (the **“how”**)

```
java_binary(
  name = "MyBinary",
  srcs = ["MyBinary.java"],
  deps = [
    ":mylib",
  ],
)

java_library(
    name = "mylib",
    srcs = ["MyLibrary.java", "MyHelper.java"],
    visibility = ["//java/com/example/myproduct:__subpackages__"],
    deps = [
        "//java/com/example/common",
        "//java/com/example/myproduct/otherlib",
        "@com_google_common_guava_guava//jar",
    ],
)
```

TARGETS

- Buildfiles define **targets**; each target correspond to an artifact that can be created by the system
- java_binary: `binary` targets produce binaries that can be executed directly
- java_library: `library` targets produce libraries that can be used by binaries or other libraries

```
java_binary(
  name = "MyBinary",
  srcs = ["MyBinary.java"],
  deps = [
    ":mylib",
  ],
)

java_library(
    name = "mylib",
    srcs = ["MyLibrary.java", "MyHelper.java"],
    visibility = ["//java/com/example/myproduct:__subpackages__"],
    deps = [
        "//java/com/example/common",
        "//java/com/example/myproduct/otherlib",
        "@com_google_common_guava_guava//jar",
    ],
)
```

TARGETS

Every target has

- `name`, which could be used to reference this target
- `srcs`, which define the source files that must be compiled to create the artifact for the target
- `deps`, which define other targets that must be built before this target and linked into it

```
java_binary(
  name = "MyBinary",
  srcs = ["MyBinary.java"],
  deps = [
    ":mylib",
  ],
)

java_library(
    name = "mylib",
    srcs = ["MyLibrary.java", "MyHelper.java"],
    visibility = ["//java/com/example/myproduct:__subpackages__"],
    deps = [
        "//java/com/example/common",
        "//java/com/example/myproduct/otherlib",
        "@com_google_common_guava_guava//jar",
    ],
)
```

BUILD PROCESS

1. Parse every BUILD file in the workspace to create **a graph of dependencies** among artifacts.

2. Use the graph to determine the **transitive dependencies** of `MyBinary`

3. Build (or download for external dependencies) each of those dependencies, **in order**.

4. Build `MyBinary` to produce a final executable binary that links in all dependencies that were built in step 3.

Difference w.r.t task-based build?

```
java_binary(
  name = "MyBinary",
  srcs = ["MyBinary.java"],
  deps = [
    ":mylib",
  ],
)

java_library(
    name = "mylib",
    srcs = ["MyLibrary.java", "MyHelper.java"],
    visibility = ["//java/com/example/myproduct:__subpackages__"],
    deps = [
        "//java/com/example/common",
        "//java/com/example/myproduct/otherlib",
        "@com_google_common_guava_guava//jar",
    ],
)
```

BENEFITS 1: PARALLELISM

- As Bazel knows that each target will only produce a Java library, it knows that **all it has to do is run the Java compiler rather than an arbitrary user-defined script**, so it knows that it's safe to run **step 3** in parallel.
- **Order of magnitude performance improvement** over building targets one at a time on a multicore machine
- Since the artifact-based approach leaves the build system in charge of its own execution strategy so that **it can make stronger guarantees about parallelism**.

BENEFITS 2: INCREMENTAL BUILD

- Bazel knows that each target is the result only of running a Java compiler, and it knows that the output from the Java compiler depends only on its inputs, so **as long as the inputs haven't changed, the output can be reused**.
- Similar to functional programming: the same input always produce the same output, with **no side effects**
- It's able to **rebuild only the minimum set of artifacts** each time while guaranteeing that it won't produce stale builds.

BENEFITS 2: INCREMENTAL BUILD

If `MyBinary.java` changes, Bazel knows to rebuild `MyBinary` but reuse `mylib`

If a source file for `//java/com/example/common` changes, Bazel knows to rebuild that library, `mylib`, and `MyBinary`, but reuse `//java/com/example/myproduct/otherlib`.

```
java_binary(
  name = "MyBinary",
  srcs = ["MyBinary.java"],
  deps = [
    ":mylib",
  ],
)

java_library(
    name = "mylib",
    srcs = ["MyLibrary.java", "MyHelper.java"],
    visibility = ["//java/com/example/myproduct:__subpackages__"],
    deps = [
        "//java/com/example/common",
        "//java/com/example/myproduct/otherlib",
        "@com_google_common_guava_guava//jar",
    ],
)
```

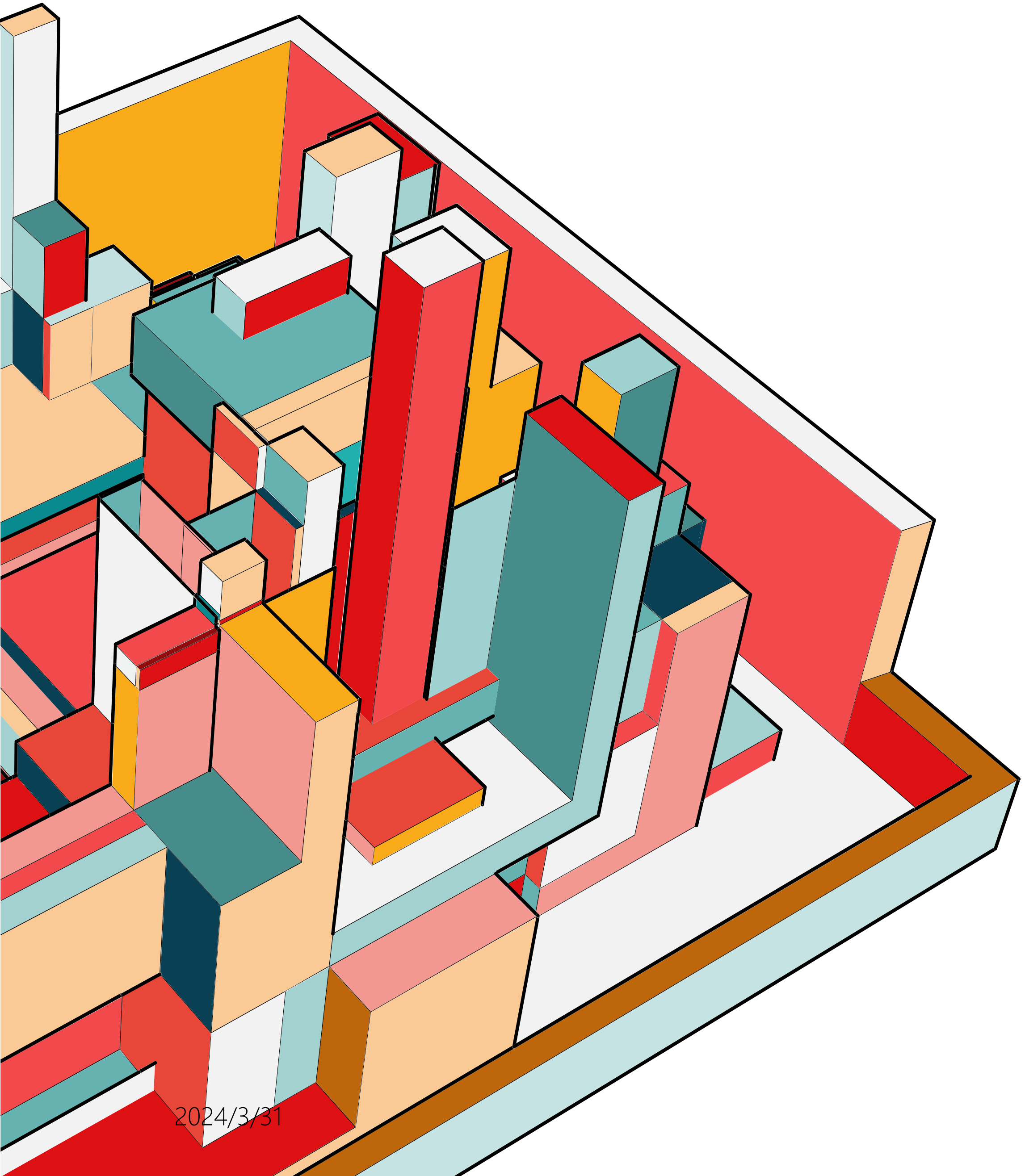
COMPARISON

Task-based Build Systems

- Engineers define a series of steps to execute
- **Imperative** approach: flexible and powerful, but hard to guarantee correctness and parallelize

Artifact-based Build Systems

- Engineers declare a manifest describing the input (e.g., source files and tools like compilers) and output (e.g., binaries)
- Let the system figure out how to execute the build
- **Declarative** approach: strong guarantees about the correctness and easy to parallelize



BUILD ARTIFACTS

The outcome of a build process is typically binary artifacts (e.g., .jar)

- Q1: How do we uniquely identify these artifacts?

SEMANTIC VERSIONING

SemVer is the widely used convention for versioning releases using a three decimal-separated integers

Format: {MAJOR}.{MINOR}.{PATCH} (e.g., 2.4.72 or 1.1.4.)

- MAJOR version change indicates a change to an existing API that can break existing usage (not backward compatible)
- MINOR version change indicates purely added functionality that should not break existing usage (backward compatible)
- PATCH version change indicates non-API-impacting implementation details (bug fixes) that are viewed as particularly low risk

Additional labels for pre-release and build metadata are available as extensions (e.g., 1.0.0-beta, 3.1.0-alpha.1+20220310)

SEMANTIC VERSIONING

Using SemVer, version requirement can be expressed as "anything newer than", excluding API-incompatible changes (i.e., major version changes)

Declare dependency on "Bar ≥ 2.1"

- √ Bar 2.1
- √ Bar 2.1.0, 2.1.1
- √ Bar 2.2 onwards

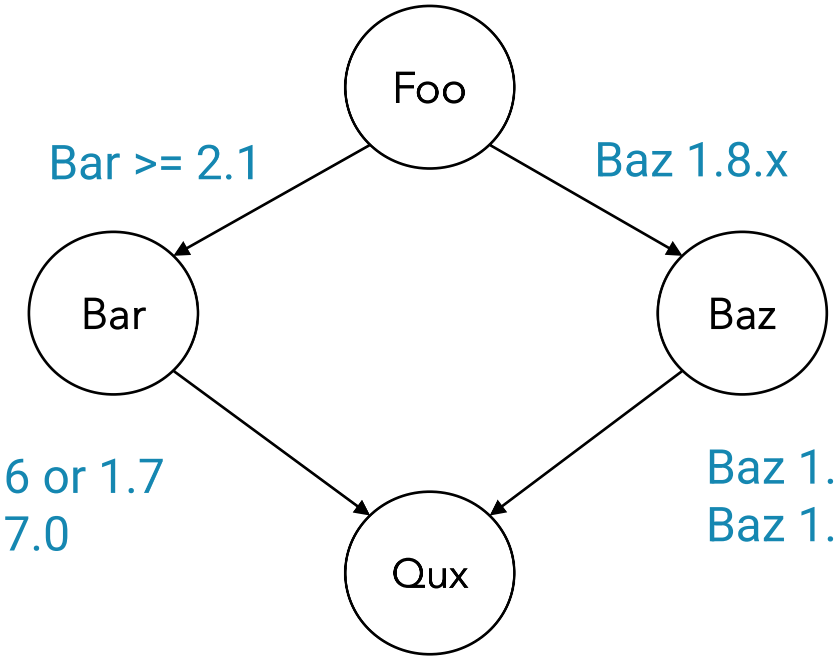
- × Bar 2.0.x
- × Bar 3.x (some APIs in Bar were change incompatibly)

SEMANTIC VERSIONING

Using SemVer, we can model the dependency problem as: given a set of constraints (version requirements on dependency edges), can we find a set of versions for the nodes that satisfies all constraints?

Bar >= 2.1

Baz 1.8.x



SAT Solver:
Use Bar 2.1.0, Baz 1.8.1 and Qux 1.6.{latest}

Bar 2.1.0 depends on Qux 1.6 or 1.7
Bar 2.1.1 depends on Qux 1.7.0

Baz 1.8.0 depends on Qux 1.5.x
Baz 1.8.1 depends on Qux 1.6.x

SEMANTIC VERSIONING

People rely on SemVer contract

rohanpadhye / JQF Public

Code | Issues 9 | Pull requests 3 | Actions | Wiki | Security | Insights

Clarify versioning schema #150

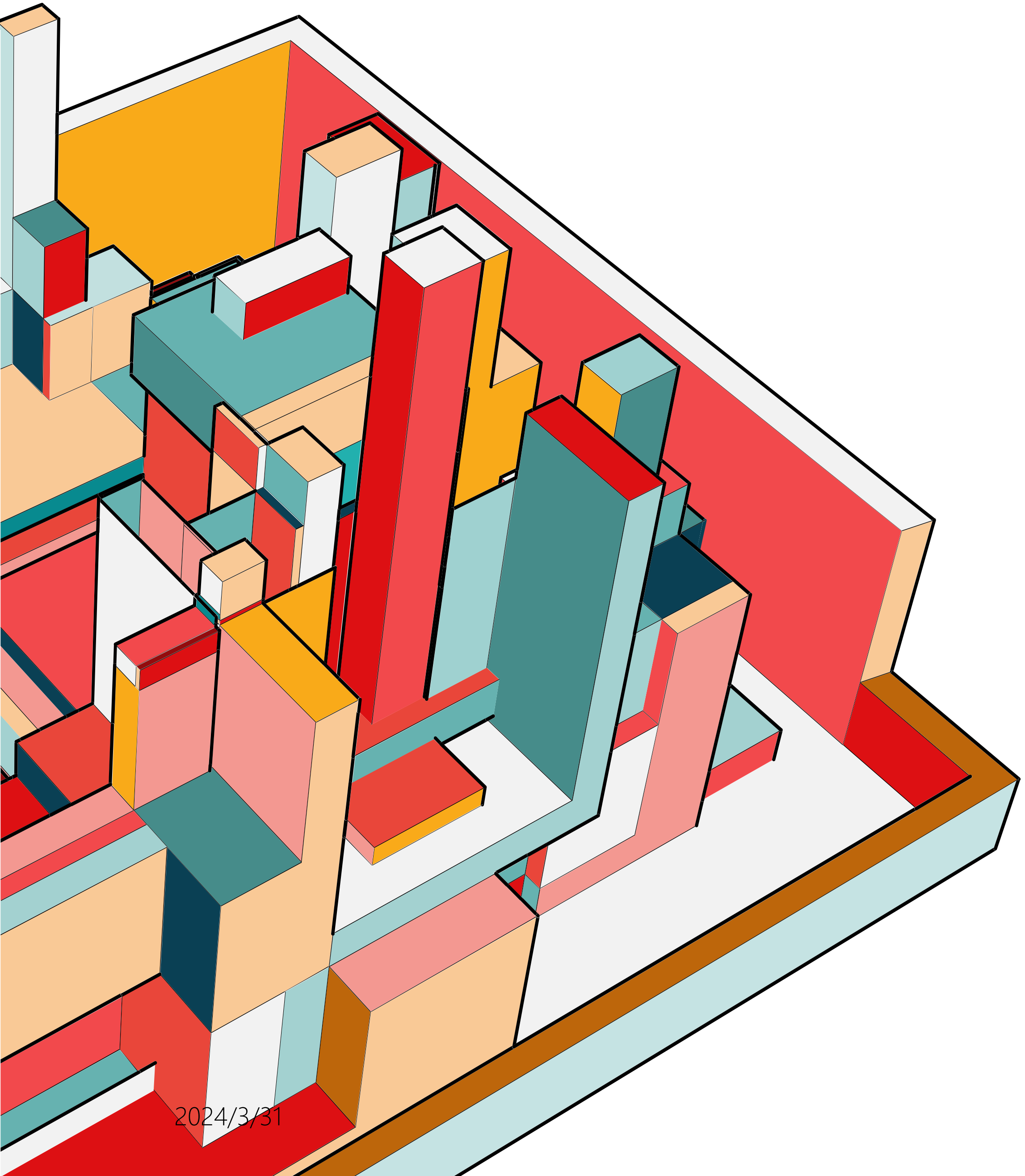
Closed sdruskat opened this issue on Aug 18, 2021 · 3 comments

sdruskat commented on Aug 18, 2021

Hi, and thanks for a great project.

I'm wondering what the versioning schema for this project is. Seeing the tags (containing `1.8`, etc.), I was assuming SemVer, but I see that the API has changed between minor increments (e.g., the newly added constructor arguments in `ZestGuidance`)? Or am I mixing up things?

FWIW, I think that following semantic versioning would be great, and make it easier for forks to contribute back to the upstream.



BUILD ARTIFACTS

The outcome of a build process is typically binary artifacts (e.g., .jar)

- Q1: How do we uniquely identify these artifacts?
- Q2: Should we package dependencies into the final artifacts?

OPTION 1: BUNDLE EVERYTHING TOGETHER

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

Linux Distribution Bundles

Table 1–3 Linux x86 Distribution Bundles

Distribution Bundle	Contents Included
Linux x86 platform	All product and shared components Installer Uninstaller

OPTION 1: BUNDLE EVERYTHING TOGETHER

Pros

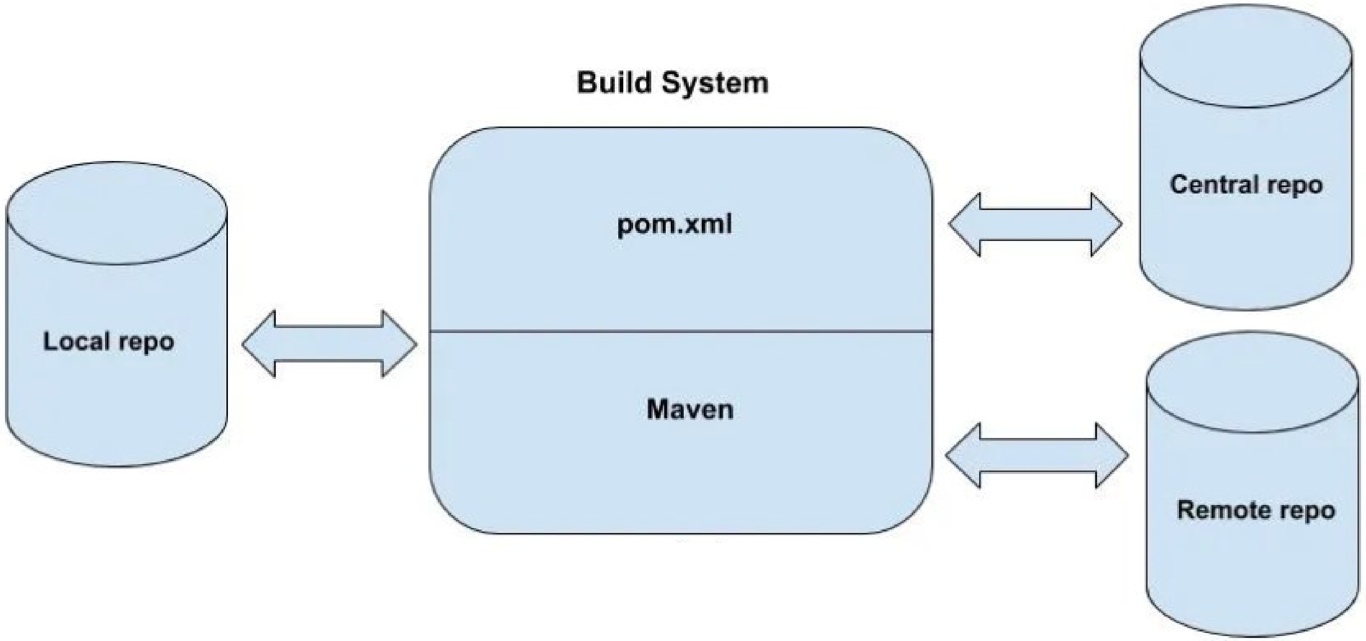
- Easier for end users to install

Cons

- Long time to build
- Dedicated Distributors
- Hard to manage dependency updates

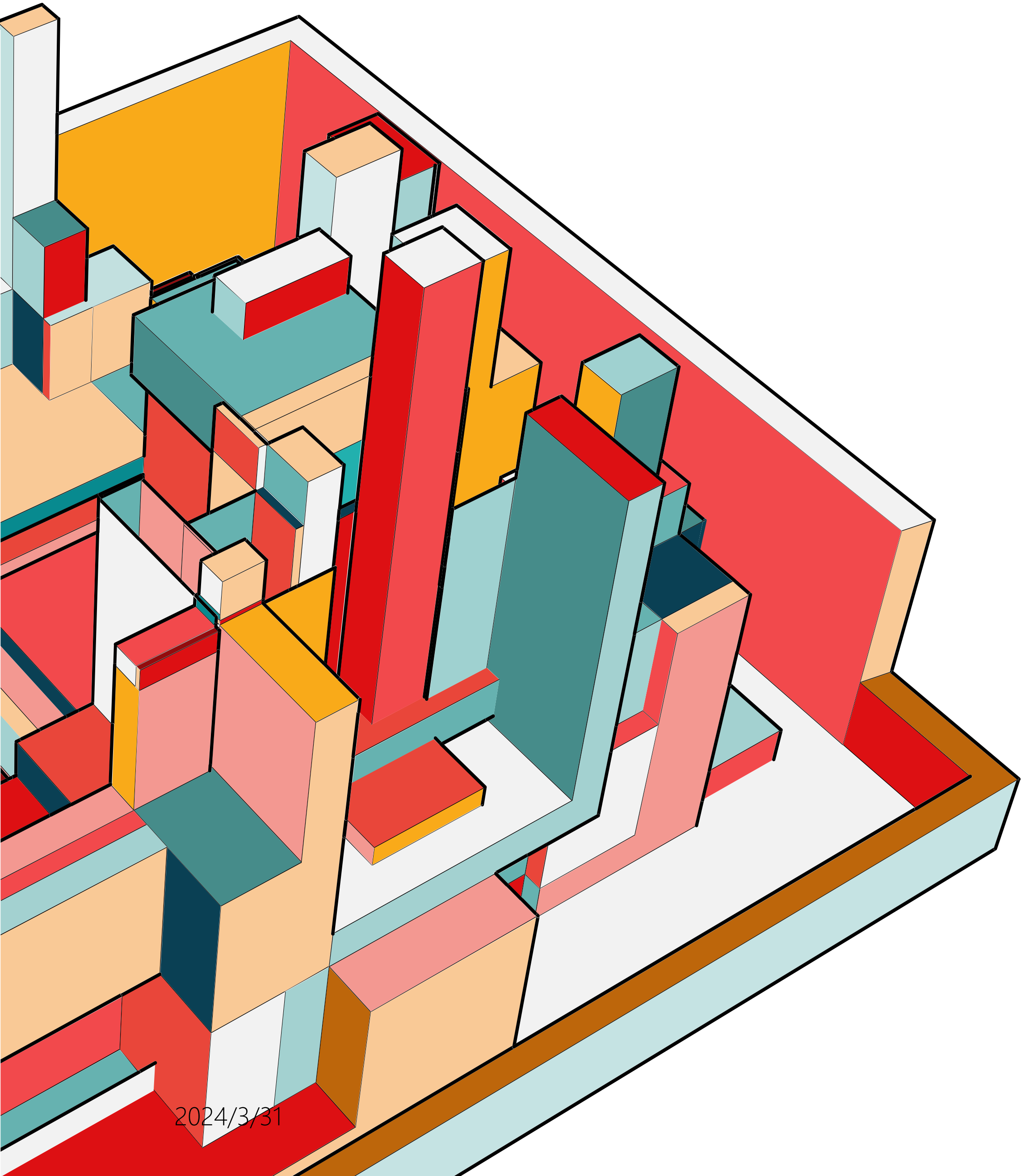
OPTION 2: LEAVE THE DEPENDENCIES OUT

The build depends on artifacts built and stored outside of the project and typically accessed via Internet



Maintained by the Maven community

Upload our artifacts to a team or organization **artifact repo** (not source repo like github) for reuse



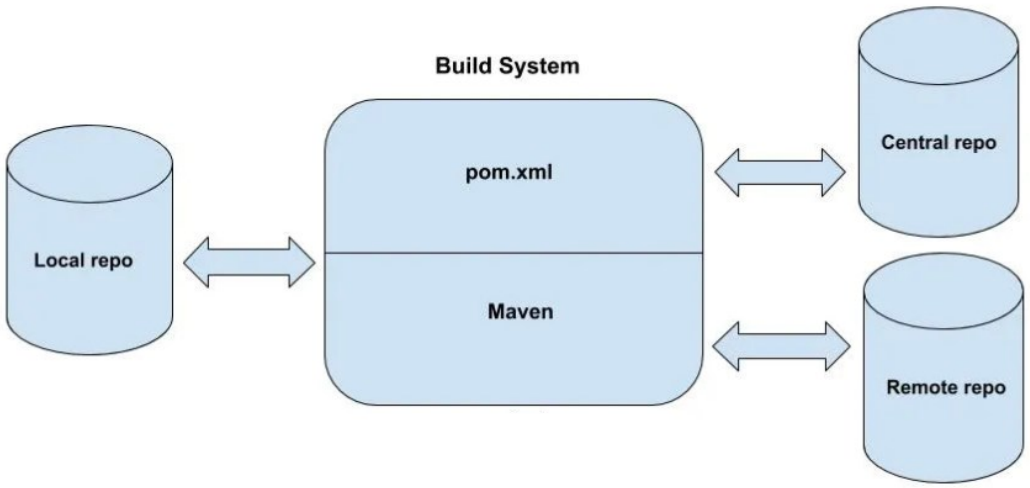
BUILD ARTIFACTS

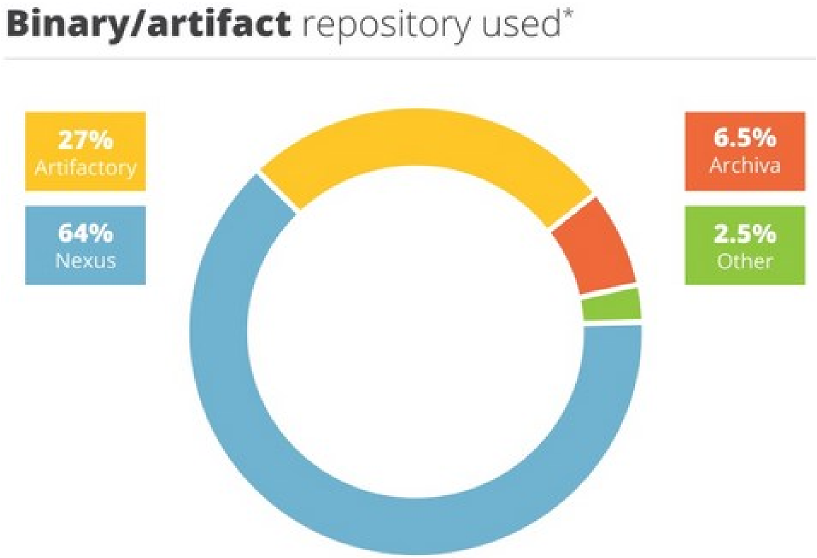
The outcome of a build process is typically binary artifacts (e.g., .jar)

- Q1: How do we uniquely identify these artifacts?
- Q2: Should we package dependencies into the final artifacts?
- Q3: How do we manage the versions of these artifacts?

ARTIFACT(BINARY) REPOSITORY

Artifact repository is a collection of binary software artifacts and metadata stored in a defined directory structure which is used by clients such as Maven to retrieve binaries during a build process.



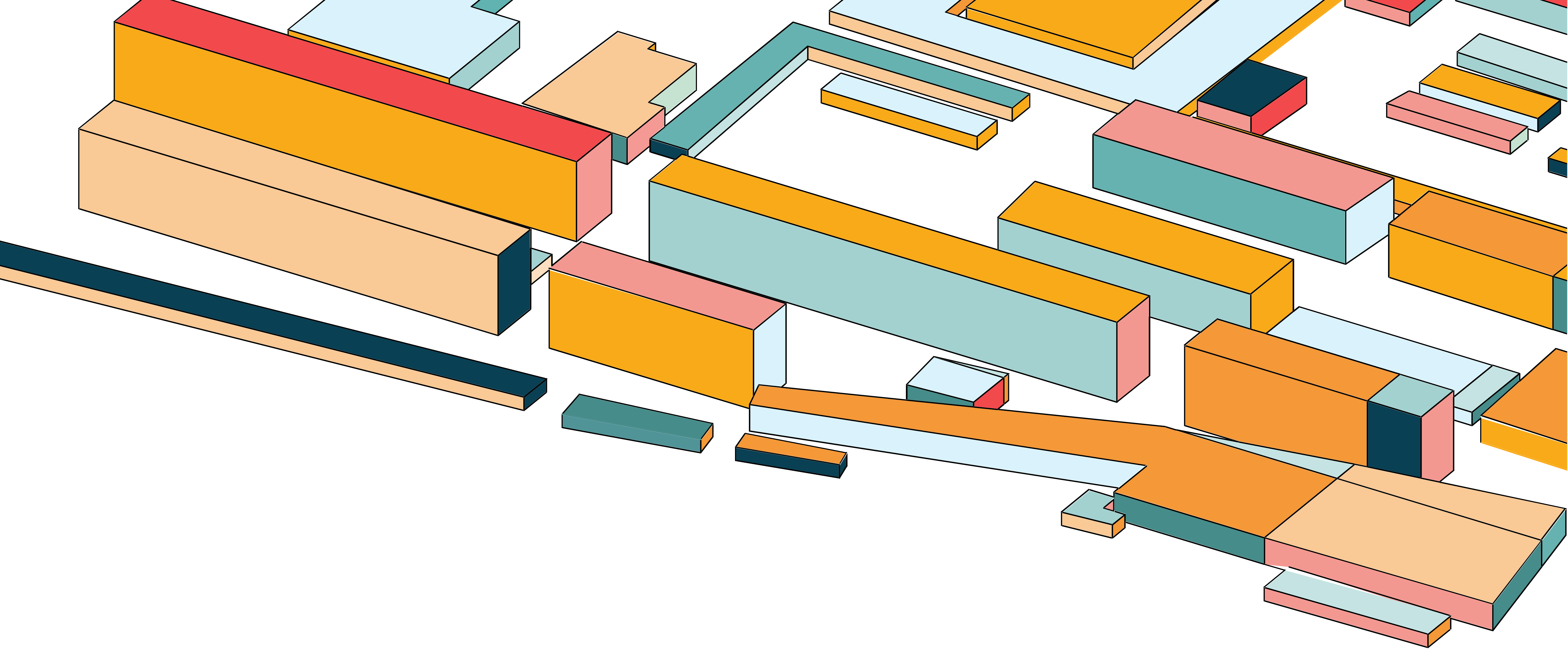


BUILDFILE AS CODE

- Buildfile (e.g., `pom.xml`) should be version controlled just like source code
- Binary dependencies and artifacts are stored in other places (artifacts repo)

As long as we have source code and the buildfile, we can always build the software

A crucial step for CI/CD



MANAGING DEPENDENCIES

DEPENDENCY

"I need that before I can have this"

Internal vs External Dependency

- **Internal dependency**: depend on another part of the codebase owned by your team or other teams in the same organization
- **External dependency**: depend on code or data owned by 3rd party individuals or providers

Task vs Artifact Dependency

- **Task dependency**: "I need to push the documentation before I mark a release as complete"
- **Artifact dependency**: "I need to have the latest version of the compute vision library before I could build my code"

DEPENDENCY

“I need that before I can have this”

Dependency Scopes

- **Compile-time**: `Foo` uses classes or functions defined by `Bar`
- **Runtime**: `Foo` needs `Bar` (e.g., a database or network server) to be ready in order to execute
- **Test**: `Foo` needs `Bar` only for tests (e.g., JUnit)

DEPENDENCY PROBLEM 1

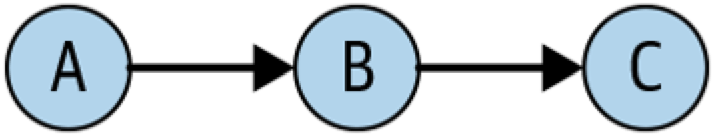
Task A depends on task B to produce a particular file as output.
The owner of task B doesn't realize that other tasks rely on it, so they update it to produce output in a different location.
This can't be detected until someone tries to run task A and finds that it fails.

Internal dependency: communicate with the owners of B
External dependency: A needs to be updated

DEPENDENCY PROBLEM 2

Task A depends on task B, which depends on task C, which is producing a particular file as output that's needed by task A.

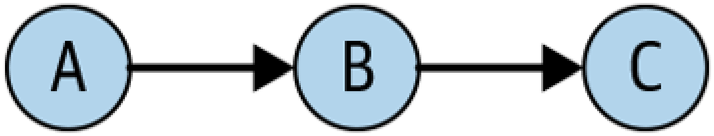
The owner of task B decides that it doesn't need to depend on task C any more, which causes task A to fail.



DEPENDENCY PROBLEM 2

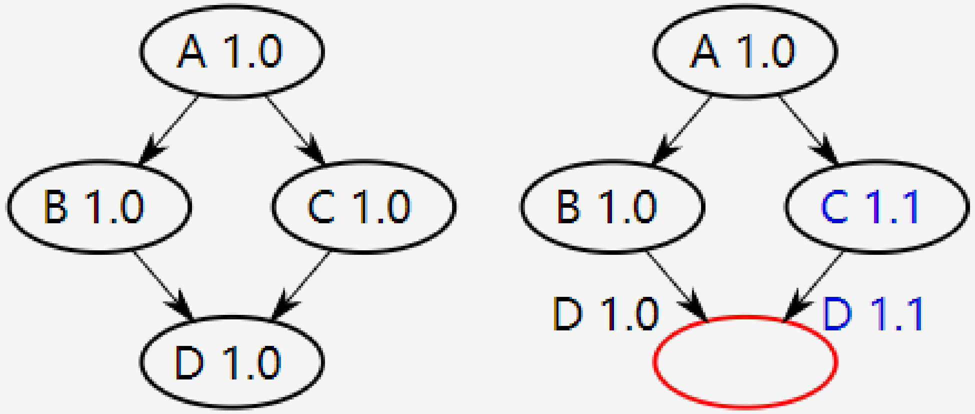
Solution: Google enforces strict transitive dependencies on Java code by default.

- Blaze detects whether a target tries to reference a symbol without depending on it directly
- If so, the build fails with an error and a shell command that can be used to automatically insert the dependency
- Google also developed tools that automatically detect many missing dependencies and add them to a BUILD files without any developer intervention.



DEPENDENCY PROBLEM 3

Diamond Dependency Issues lead to conflicts and unexpected results

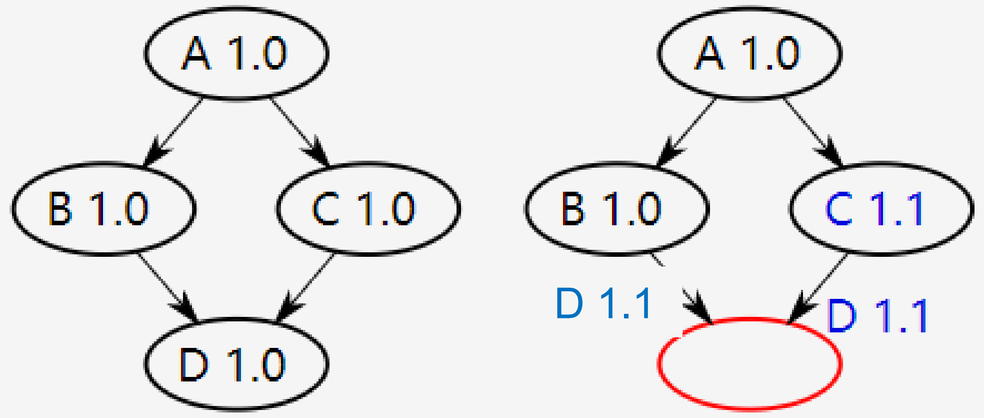


When two dependencies require conflicting versions of a common transitive dependency, you either have a conflict that needs solving, or you have duplication.

https://www.tedinski.com/2018/03/27/maven-design-case-study.html

DEPENDENCY PROBLEM 3

Solution 1: Update B1.0 so that it depends on D1.1

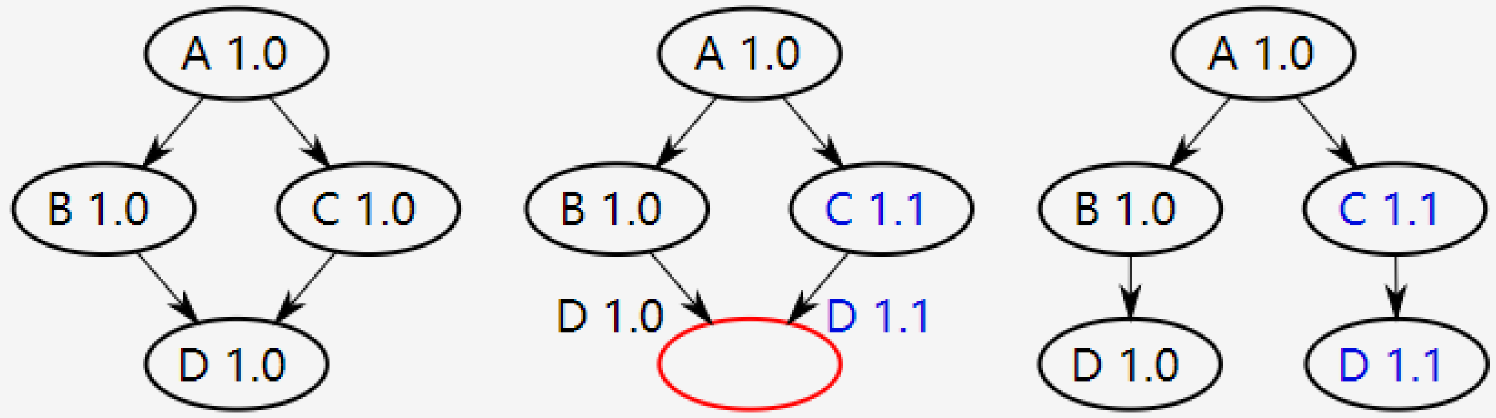


When two dependencies require conflicting versions of a common transitive dependency, you either have a conflict that needs solving, or you have duplication.

https://www.tedinski.com/2018/03/27/maven-design-case-study.html

DEPENDENCY PROBLEM 3

Solution 2 (Duplication):
figure out a way so that D1.0 and D1.1 can co-exist in the same environment



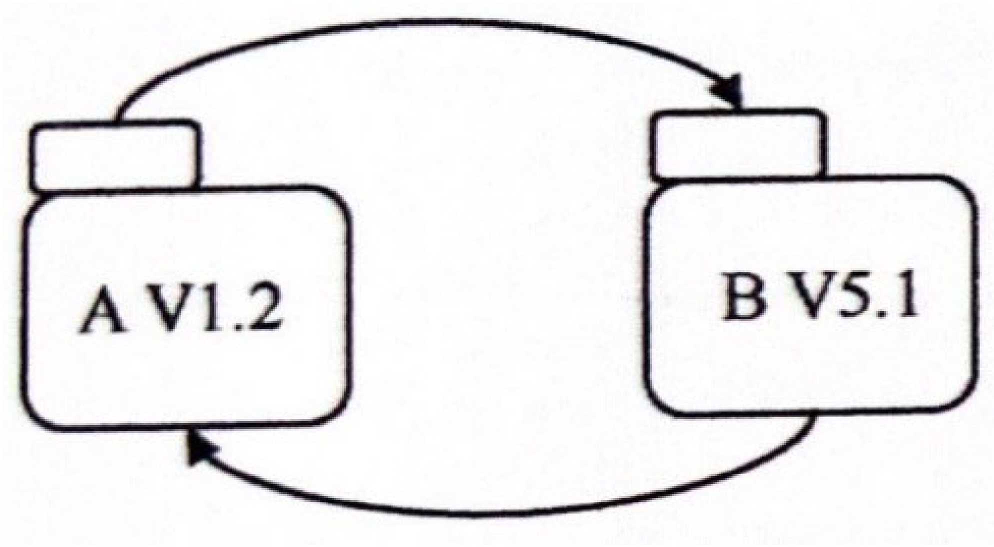
When two dependencies require conflicting versions of a common transitive dependency, you either have a conflict that needs solving, or you have duplication.

However, duplication is error-prone. Companies (e.g., Google) may not allow this solution.

https://www.tedinski.com/2018/03/27/maven-design-case-study.html

DEPENDENCY PROBLEM 4

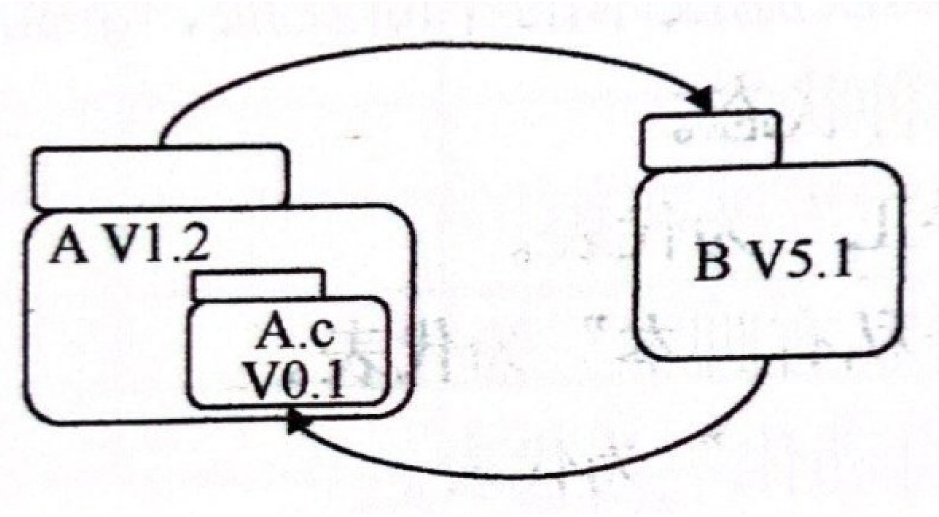
Circular Dependency

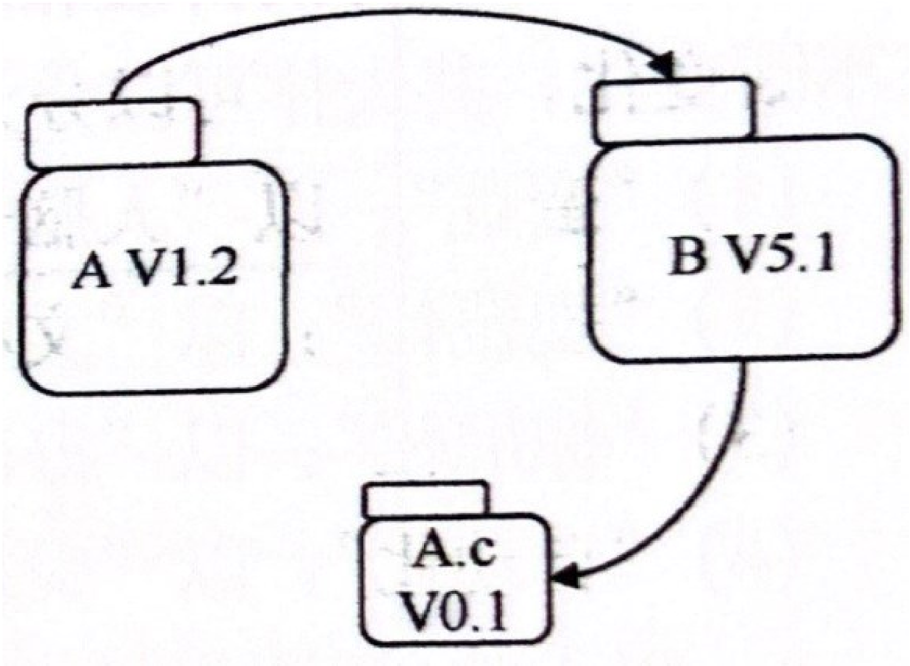


Build systems like Maven do not even allow circular dependency, as it cannot determine which component should be built first

DEPENDENCY PROBLEM 4

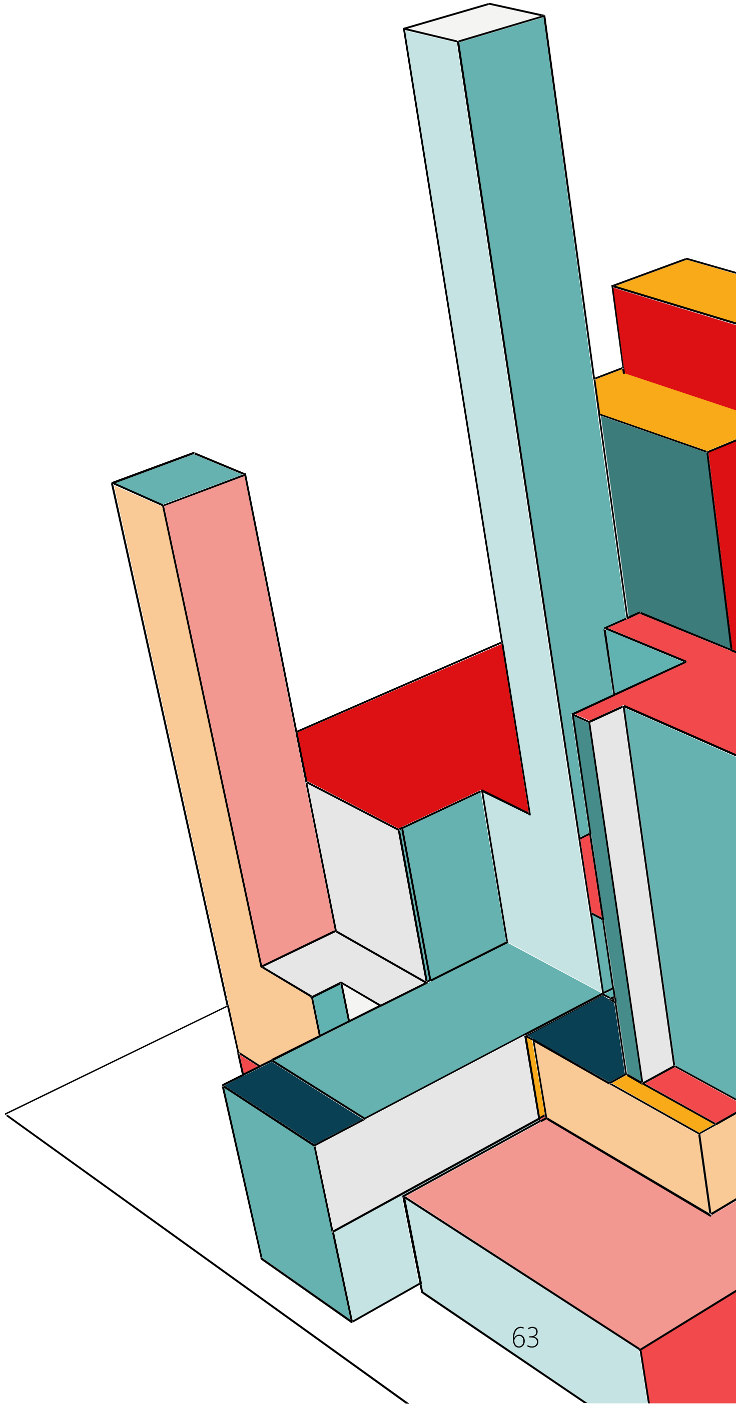
Solution: extract part of A to be another package





READINGS

- Chapter 18, 21. Software Engineering at Google by Titus Winters et al.
- Chapter 25.2. Software Engineering by Ian Sommerville, 10th edition.
- 《持续交付2.0》乔梁，第11.2，11.3章



NEXT

- Code Quality