



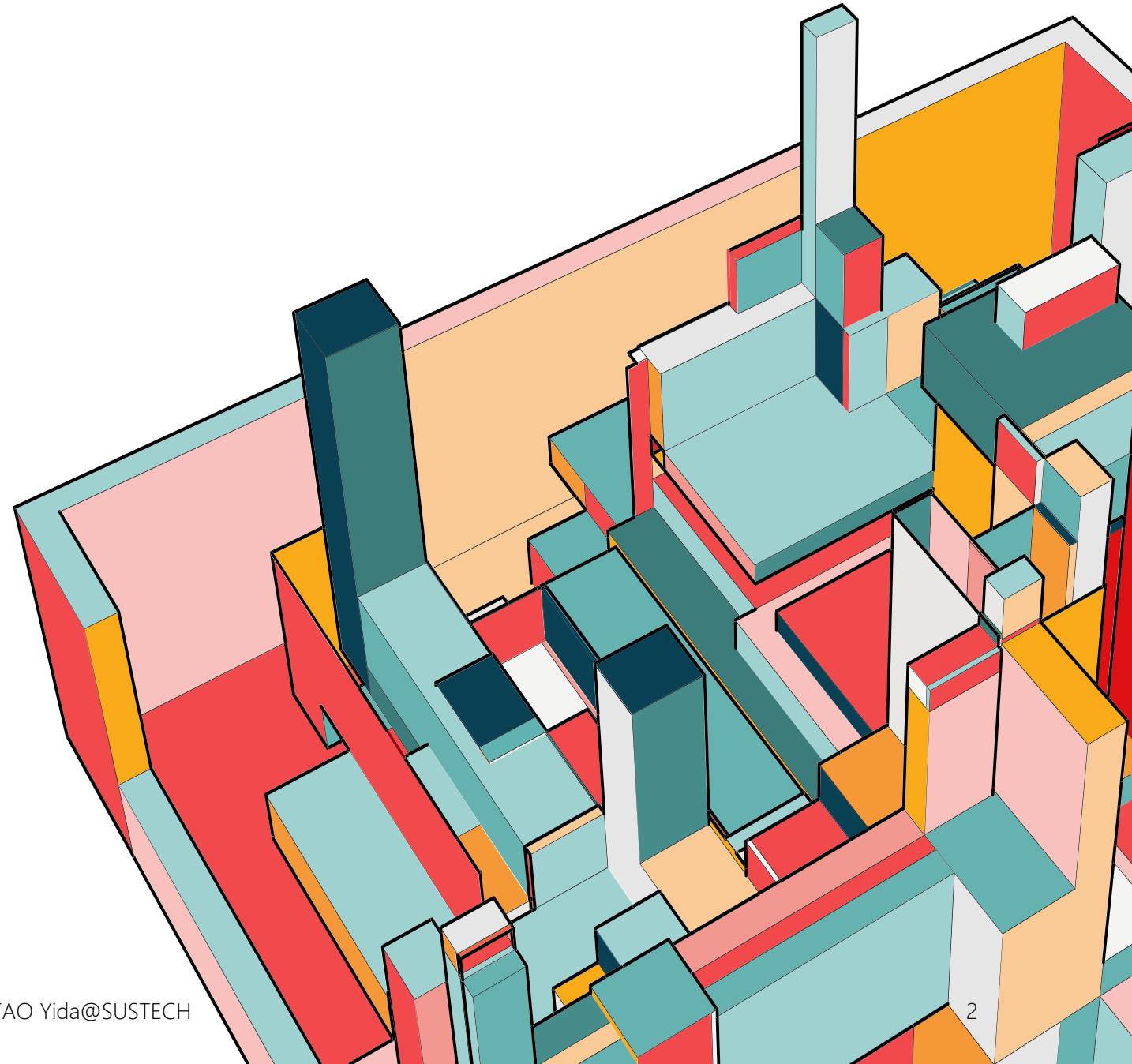
# **CS304 SOFTWARE ENGINEERING**

Yida Tao

[taoyd@sustech.edu.cn](mailto:taoyd@sustech.edu.cn)

# LECTURE 6

- Build system
- Dependency management



# ***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 3<sup>rd</sup> 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- What if the dependent JARs become outdated?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?
- .....

# ***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.

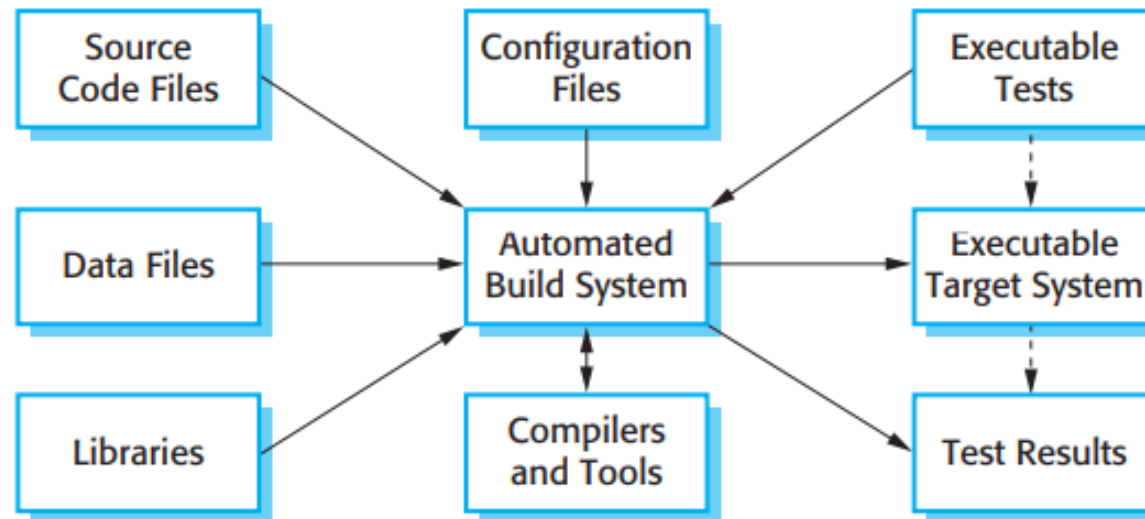
# BUILD SYSTEMS

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

External libraries may have conflict dependencies and may be outdated



Tests may depend on certain frameworks

# MANAGING DEPENDENCY

- Managing code is fairly straightforward, but managing dependencies is much more difficult
- The most fundamental job of build systems: managing dependencies

# DEPENDENCY

“I need that before I can have this”

## Internal vs External Dependency

- **Internal dependency:** depend on another part of your codebase
- **External dependency:** depend on code or data owned by another team (either in your organization or a third party)

## 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

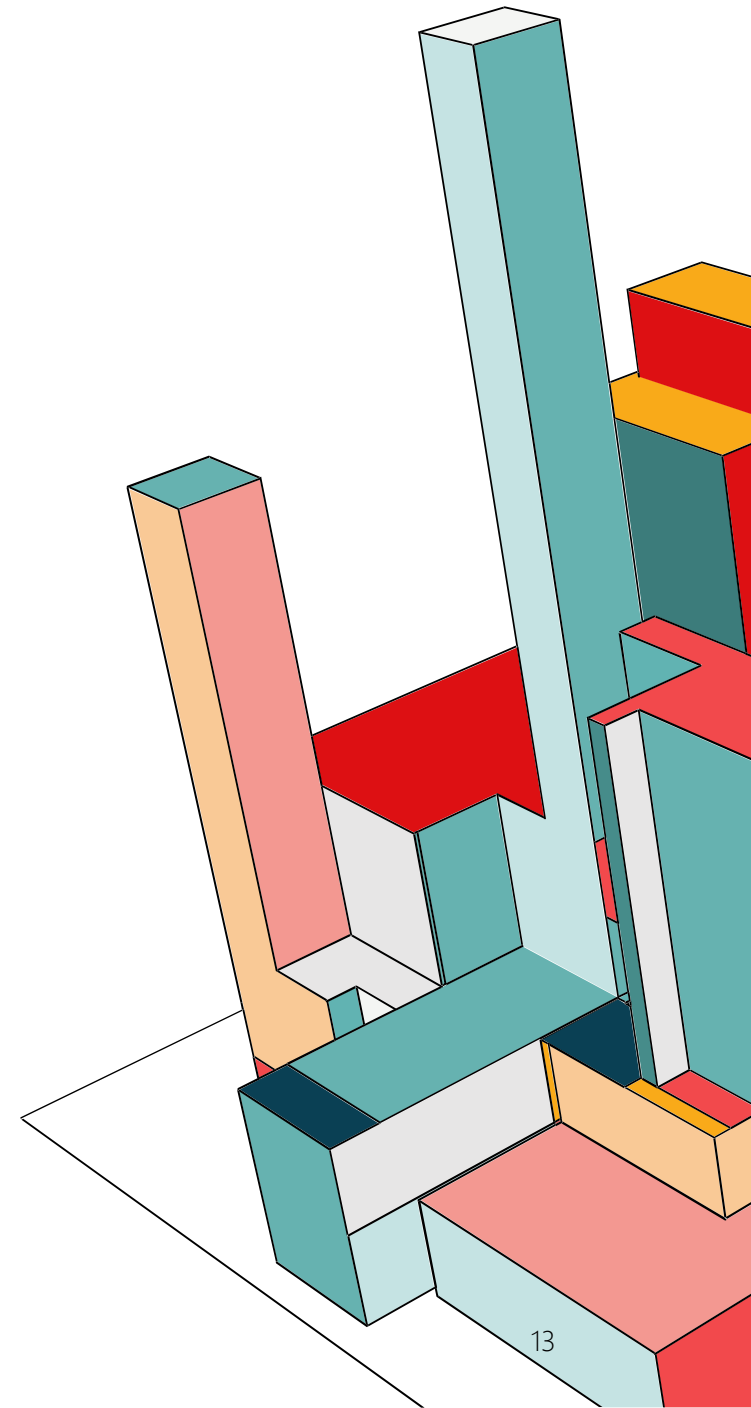
“I need that before I can have this”

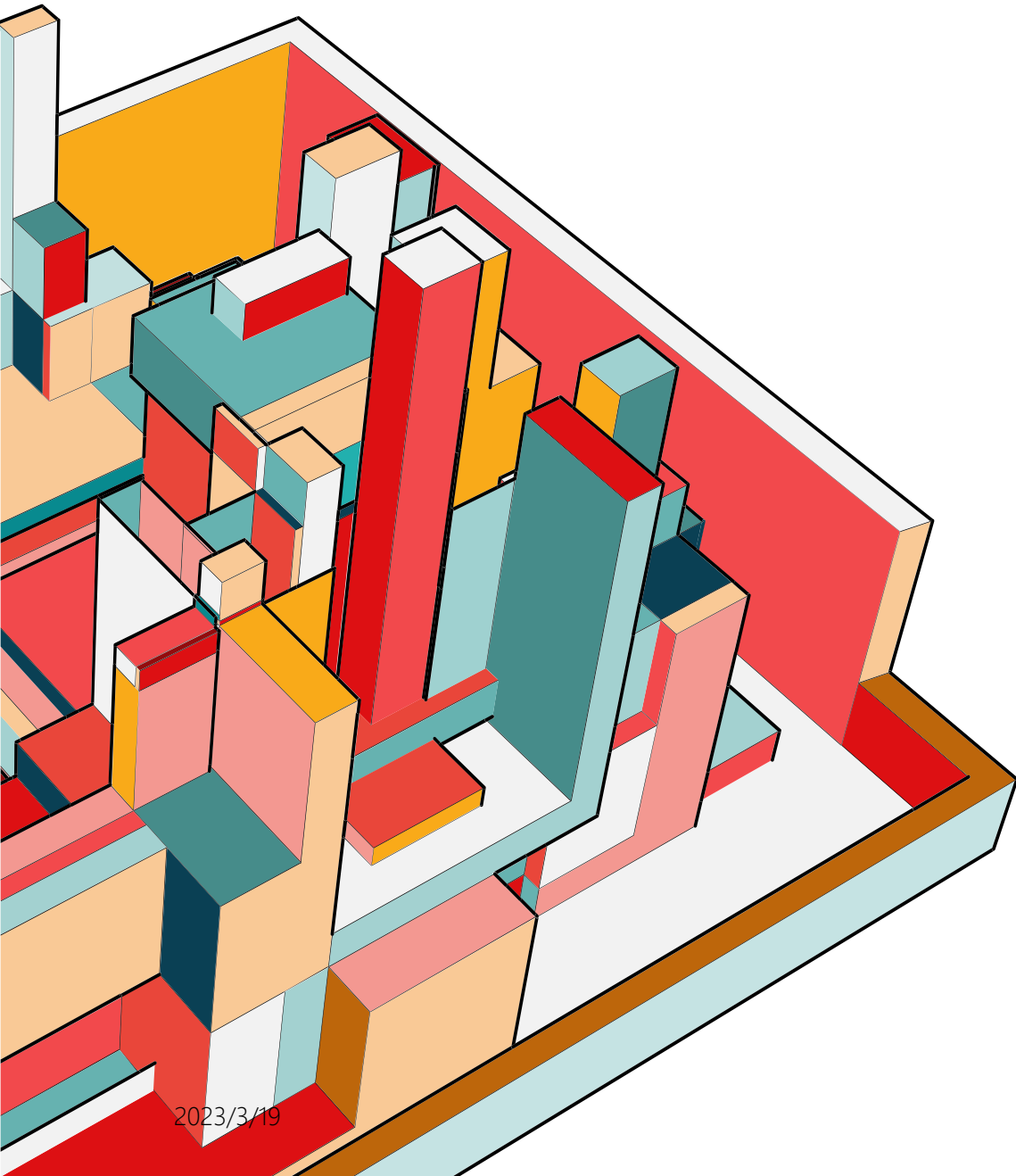
## Dependency Scopes

- **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)

# ***TYPES OF BUILD SYSTEMS***

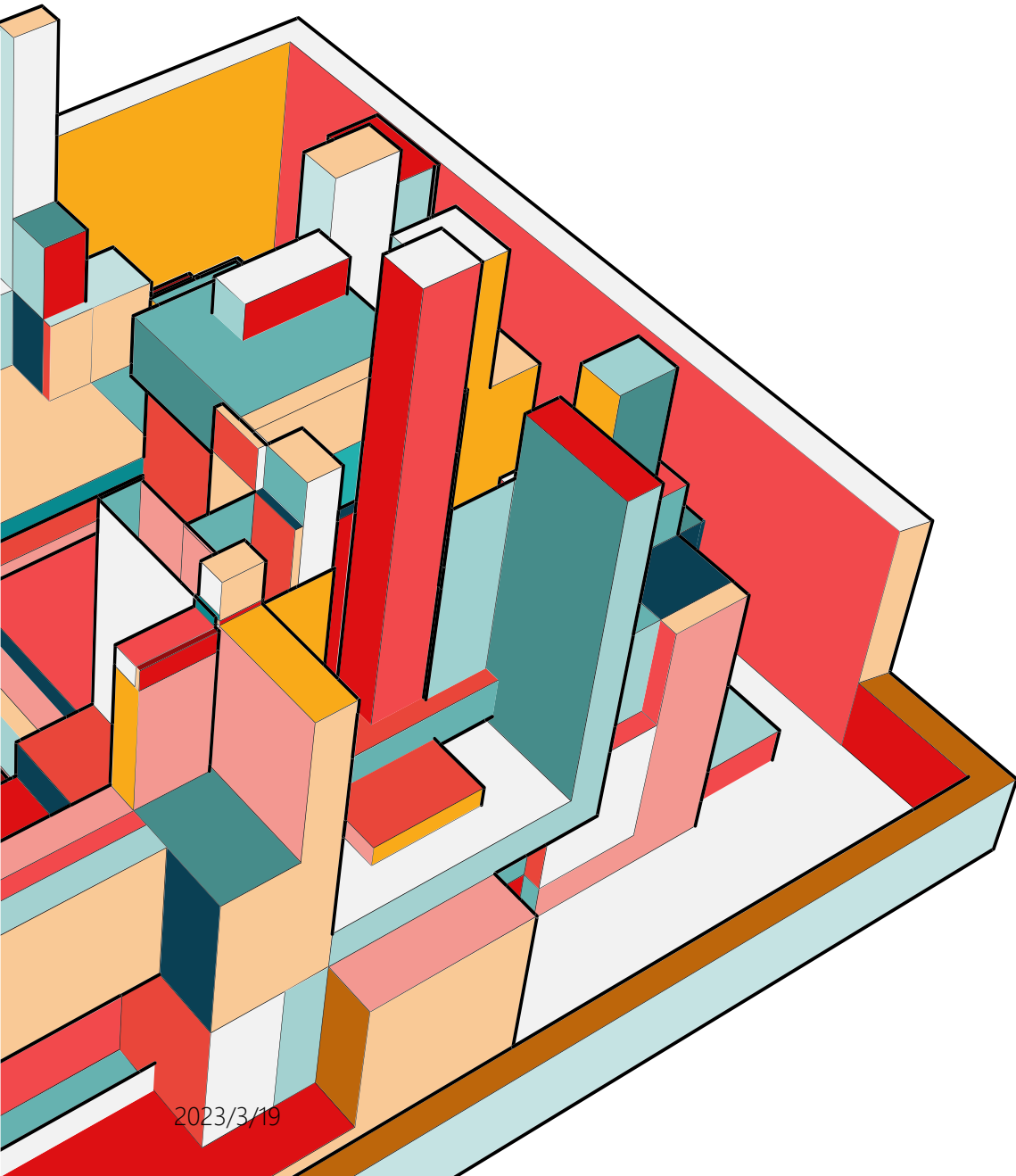
- Task-based Build Systems
- Artifact-based Build Systems





# TASK-BASED BUILD SYSTEMS

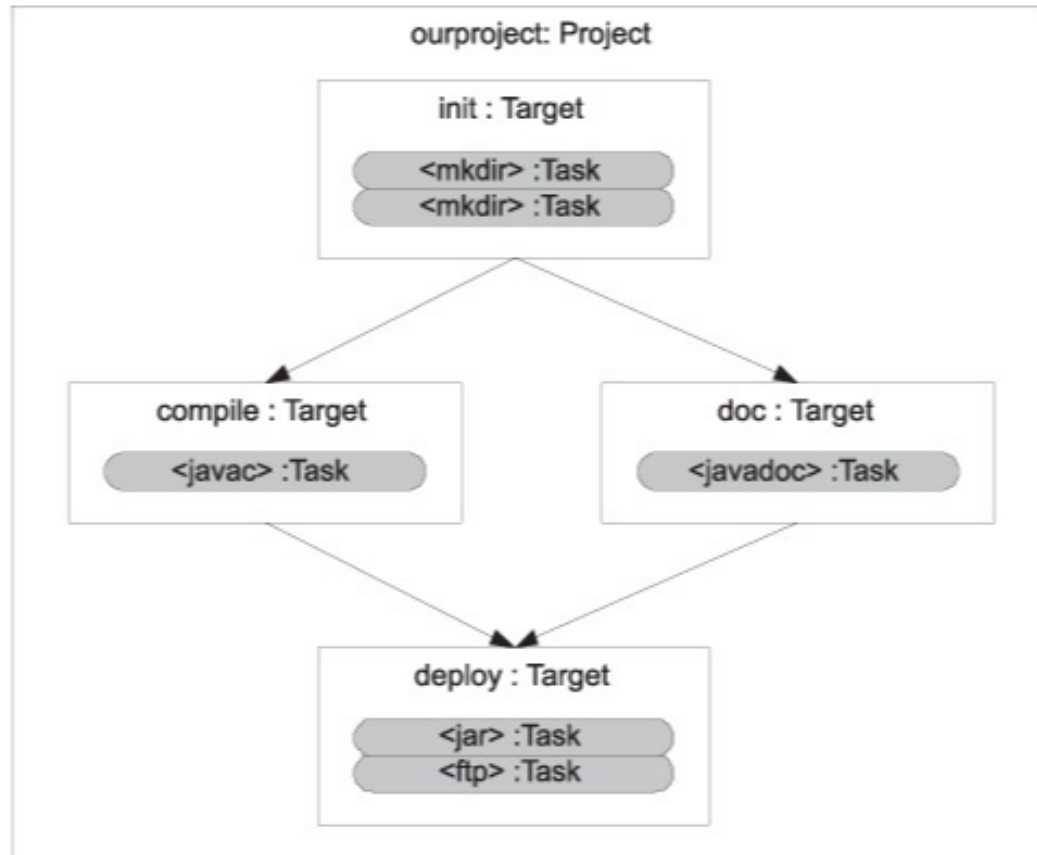
- In a task-based build system, the fundamental unit of work is the **task**
- Each task is a script of some sort that can execute any sort of logics, and tasks can specify other tasks as dependencies that must run before them



# 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)

# EXAMPLE: ANT 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

# 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

## Bug example 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 change 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.

# 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

## Bug example 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.

# 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

## Bug example 3:

The developer of a new task accidentally makes an assumption about the machine running the task, such as the location of a tool or the value of particular environment variables.

The task works on their machine, but fails whenever another developer tries it.



# 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

## Bug example 4:

A task contains a nondeterministic component, such as downloading a file from the internet or adding a timestamp to a build.

Now, people will get potentially different results each time they run the build, meaning that engineers won't always be able to reproduce and fix one another's failures or failures that occur on an automated build system.

# 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

## Bug example 5:

Tasks with multiple dependencies can create race conditions. If task A depends on both task B and task C, and task B and C both modify the same file, task A will get a different result depending on which one of tasks B and C finishes first.

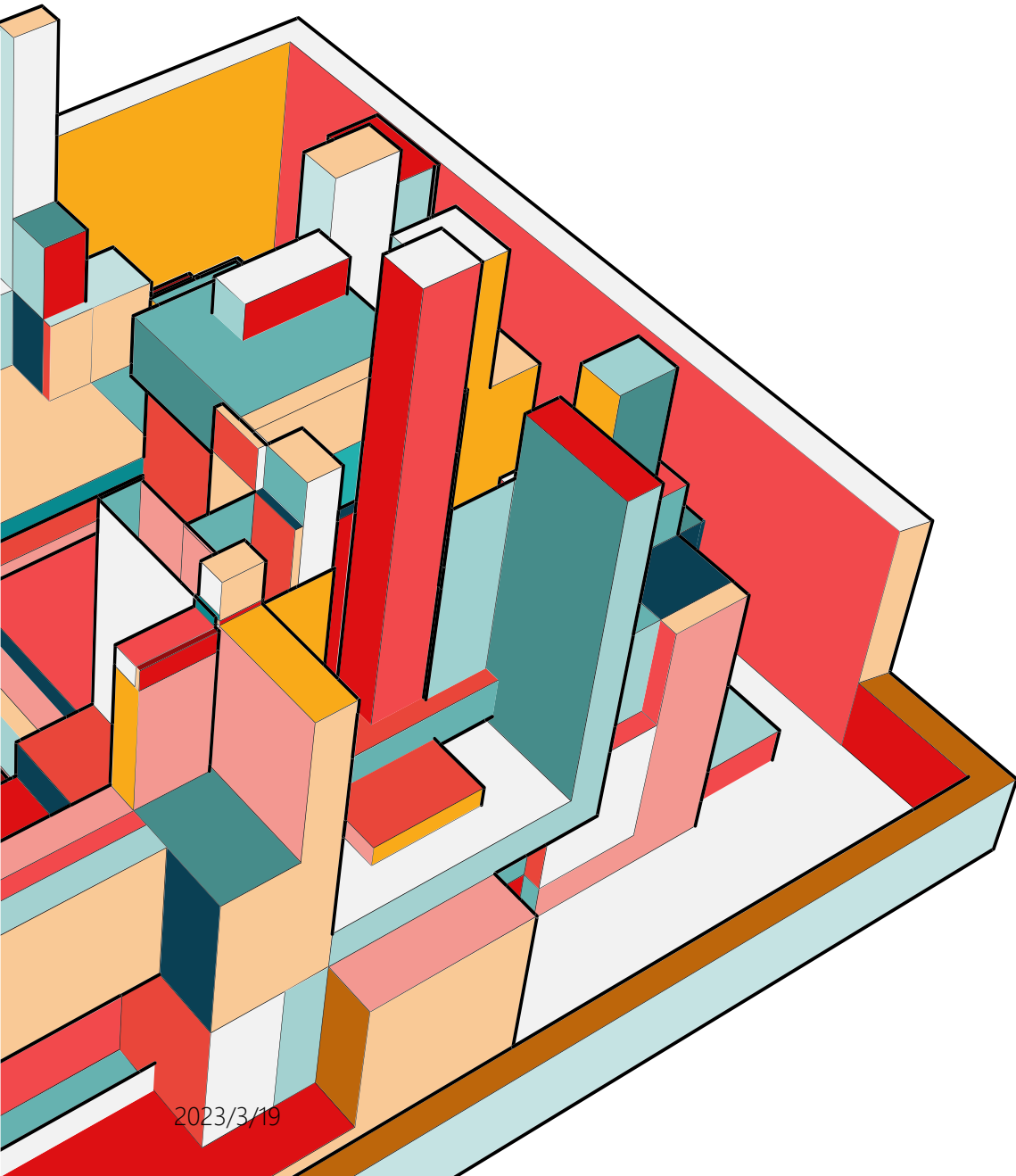
# DRAWBACKS OF TASK-BASED BUILD SYSTEMS

## Difficulty performing incremental builds

- Using a good build system, a small change doesn't require the entire code base to be rebuilt from scratch
- But for task-based build systems, it's hard to determine the change impact
- To ensure correctness, task-based build systems typically must rerun every task during each build

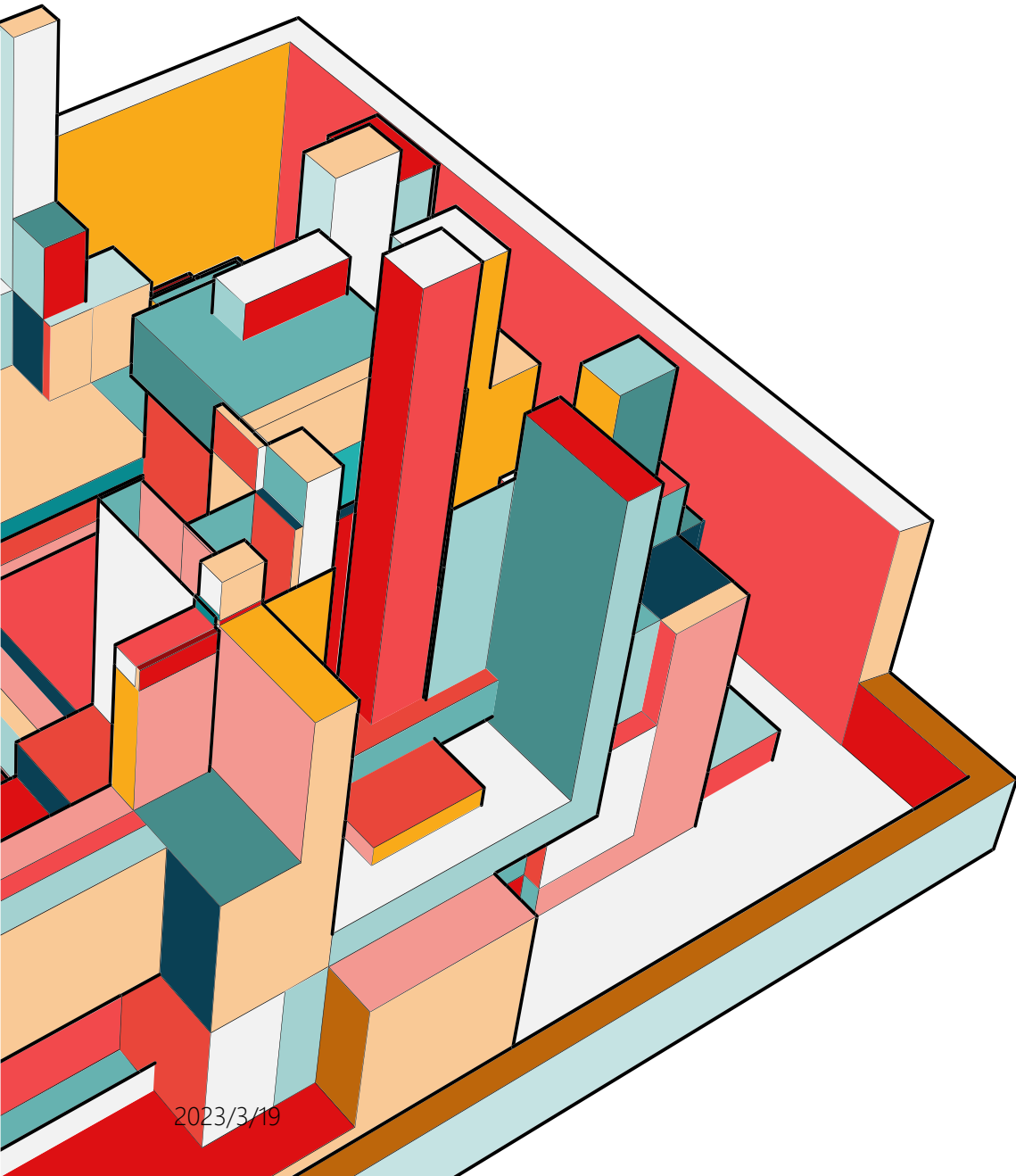
## Difficulty of parallelizing build steps

- For the same reason (hard to determine task dependencies and change impact), task-based build systems are often unable to parallel task executions
- Huge wasting of development resources
- Hard to distribute the build across multiple machines



# TASK-BASED BUILD SYSTEMS

- Engineers can write arbitrary code to execute any tasks during 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 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 system
- The approach that Google takes with **Blaze** (internal version) and **Bazel** (open-source version)

# BUILDFILES

- **Buildfiles** in artifact-based build systems like Blaze are a declarative manifest describing:
  - A set of artifacts to build
  - Their dependencies
  - Limited set of configurations
- Engineers run blaze by specifying a set of targets to build (the “**what**”)
- Blaze 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",  
    ],  
)
```



# FIRST TIME BUILD

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**; that is, every target that **MyBinary** depends on and every target that those targets depend on, recursively.
3. Build (or download for external dependencies) each of those dependencies, **in order**.
  - Bazel starts by building each target that has no other dependencies and keeps track of which dependencies still need to be built for each target.
  - When all of a target's dependencies are built, Bazel starts building that target. This process continues until every one of **MyBinary**'s transitive dependencies have been built.
4. Build **MyBinary** to produce a final executable binary that links in all dependencies that were built in step 3.

```
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

## 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.
- This can produce an **order of magnitude performance improvement** over building targets one at a time on a multicore machine, because 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

## Incremental Builds

- 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.**

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",  
    ],  
)
```

# BENEFITS

## Incremental Builds

- 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**.
- Because Bazel knows about the properties of the tools it runs at every step, it's able to **rebuild only the minimum set of artifacts** each time while guaranteeing that it won't produce stale builds.

```
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",  
    ],  
)
```

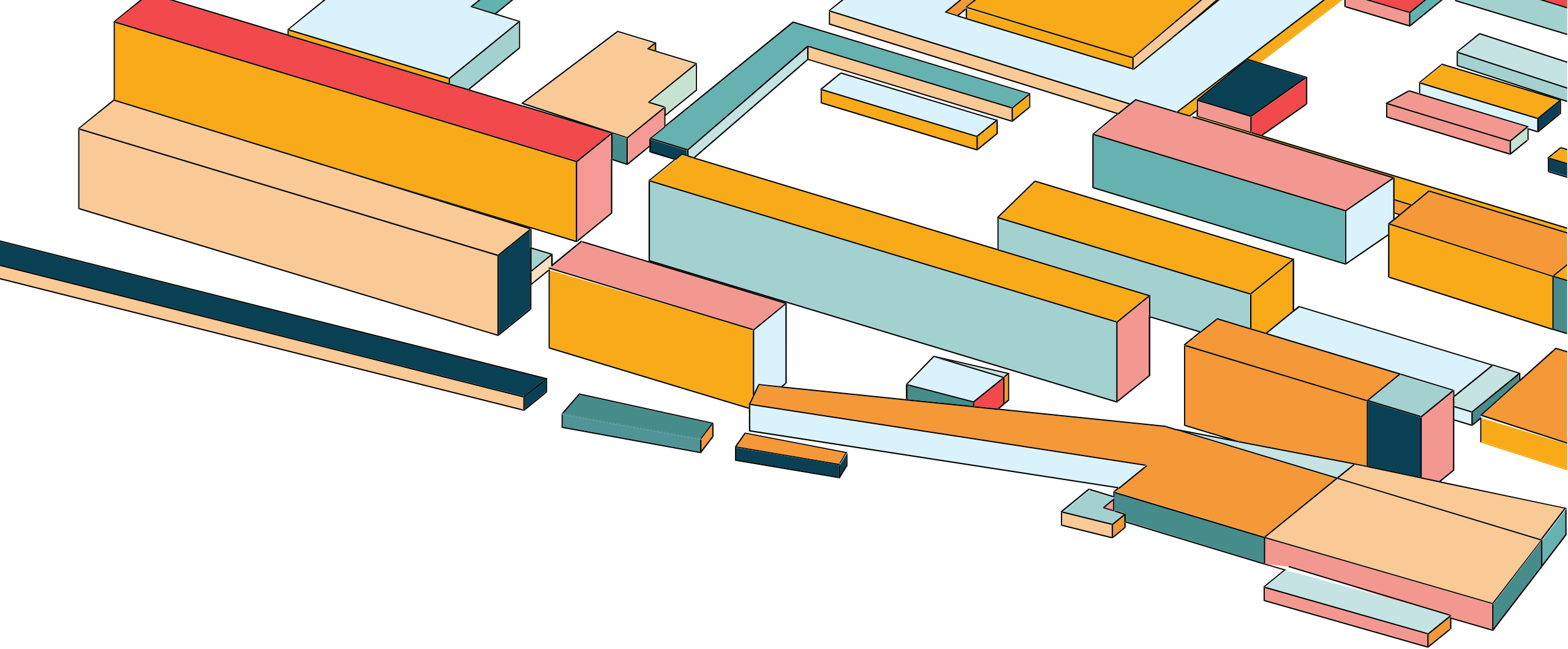
# COMPARISON

## 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

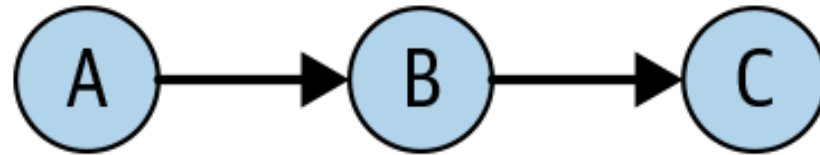


# ***MANAGING DEPENDENCIES***

# INTERNAL DEPENDENCY

A target depends on other targets in the same source repository

Suppose target A depends on target B, which depends on a common library target C. Should target A be able to use classes defined in target C?



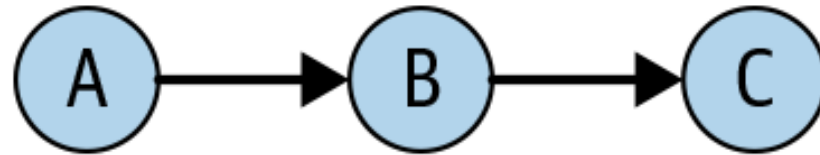
*Figure 18-5. Transitive dependencies*

**Analysis:** Suppose that B was refactored such that it no longer needed to depend on C. If B's dependency on C was then removed, A and any other target that used C via a dependency on B would break.

# INTERNAL DEPENDENCY

A target depends on other targets in the same source repository

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



*Figure 18-5. Transitive dependencies*

- 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.



# EXTERNAL DEPENDENCY

A target depends on artifacts built and stored outside of the project and typically accessed via Internet

## Where are dependencies hosted?

- Maven Central (<https://repo.maven.apache.org/maven2/>)
- Ubuntu Packages (<https://packages.ubuntu.com/>)
- Python Package Index (<https://pypi.org/>)
- NPM Public Registry (<https://registry.npmjs.org/>)

# EXTERNAL DEPENDENCY

A target depends on artifacts built and stored outside of the project and typically accessed via Internet

## Reliability Risks

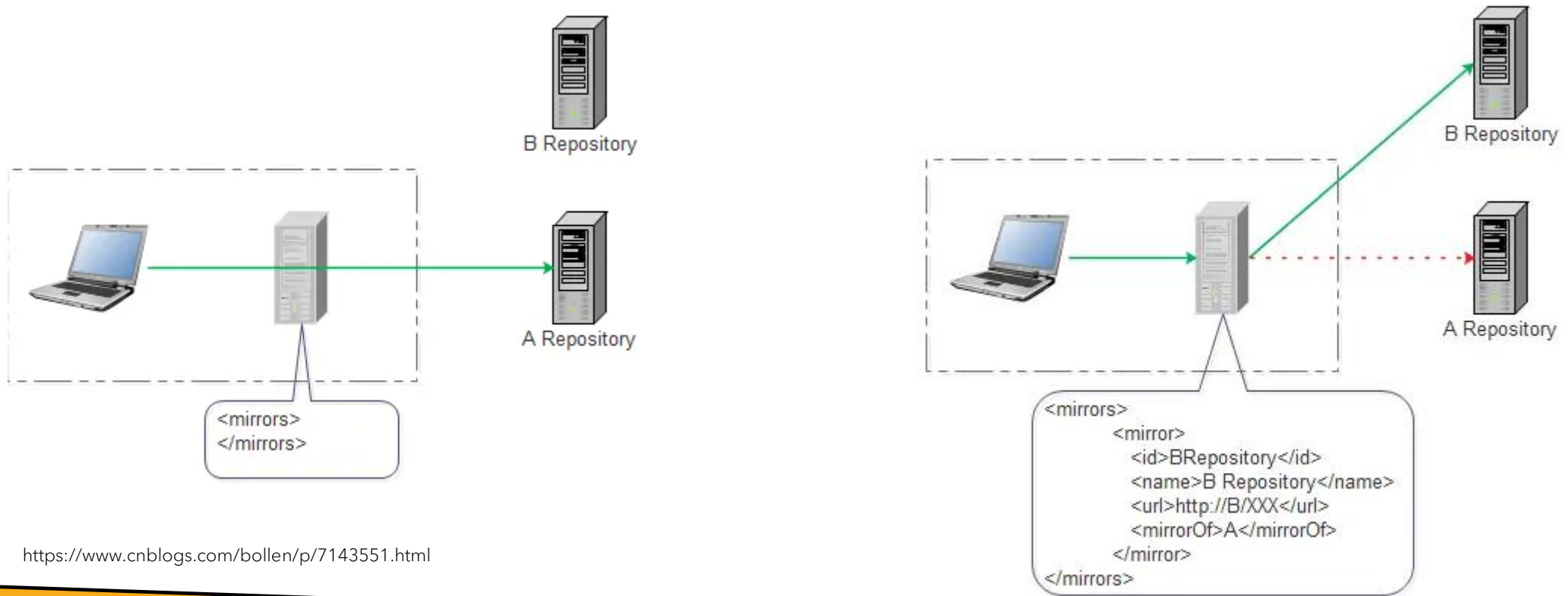
If the third-party source (e.g., a maven repository) goes down, your entire build might grind to a halt if it's unable to download an external dependency.

## How to address

The problem can be mitigated by **mirroring** any artifacts you depend on onto servers you control and blocking your build system from directly accessing third-party artifact repositories like Maven Central.

# EXTERNAL DEPENDENCY

A target depends on artifacts built and stored outside of the project and typically accessed via Internet

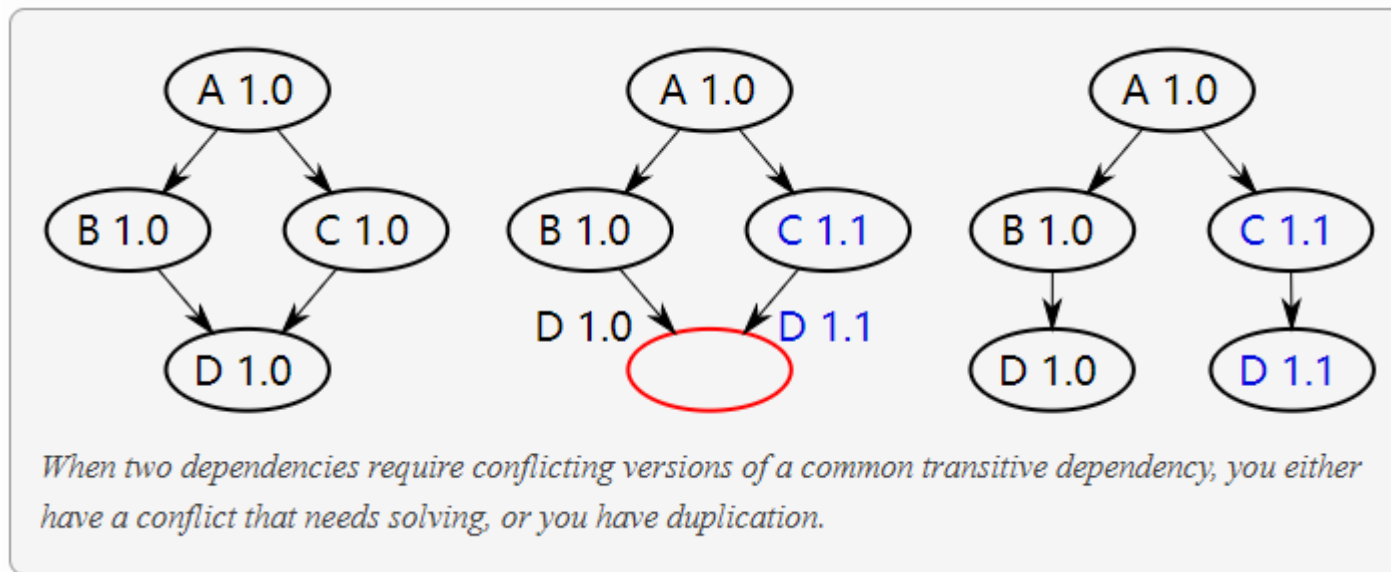


<https://www.cnblogs.com/bollen/p/7143551.html>

# EXTERNAL DEPENDENCY

A target depends on artifacts built and stored outside of the project and typically accessed via Internet

**Diamond Dependency Issues** lead to conflicts and unexpected results



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

# 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
- MINOR version change indicates purely added functionality that should not break existing usage
- 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)

# 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.

# SEMANTIC VERSIONING

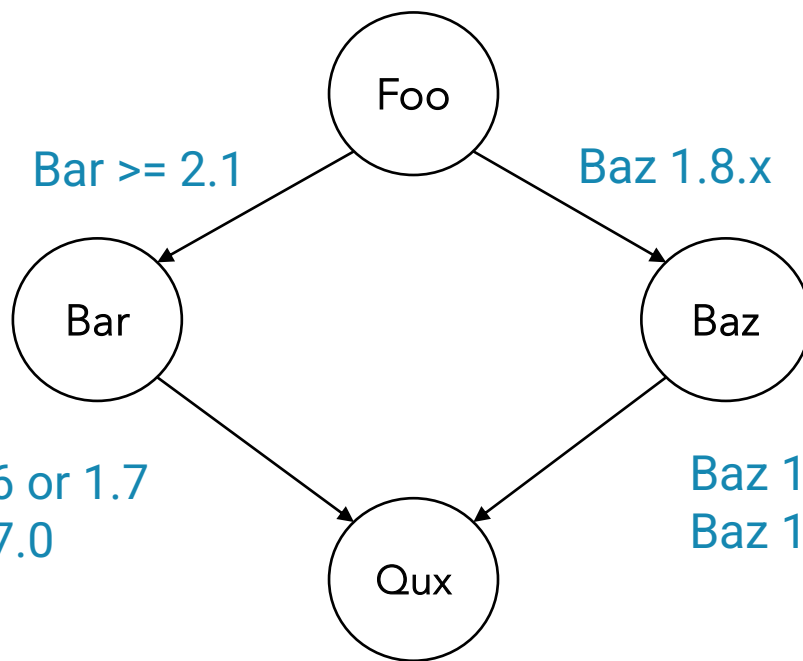
Using SemVer, version requirement can be expressed as “anything newer than”, excluding API-incompatible changes (i.e., major version changes)

Declare dependency on “Bar  $\geq$  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?



**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 2.0.0

## Summary

Given a version number MAJOR.MINOR.PATCH, increment the:

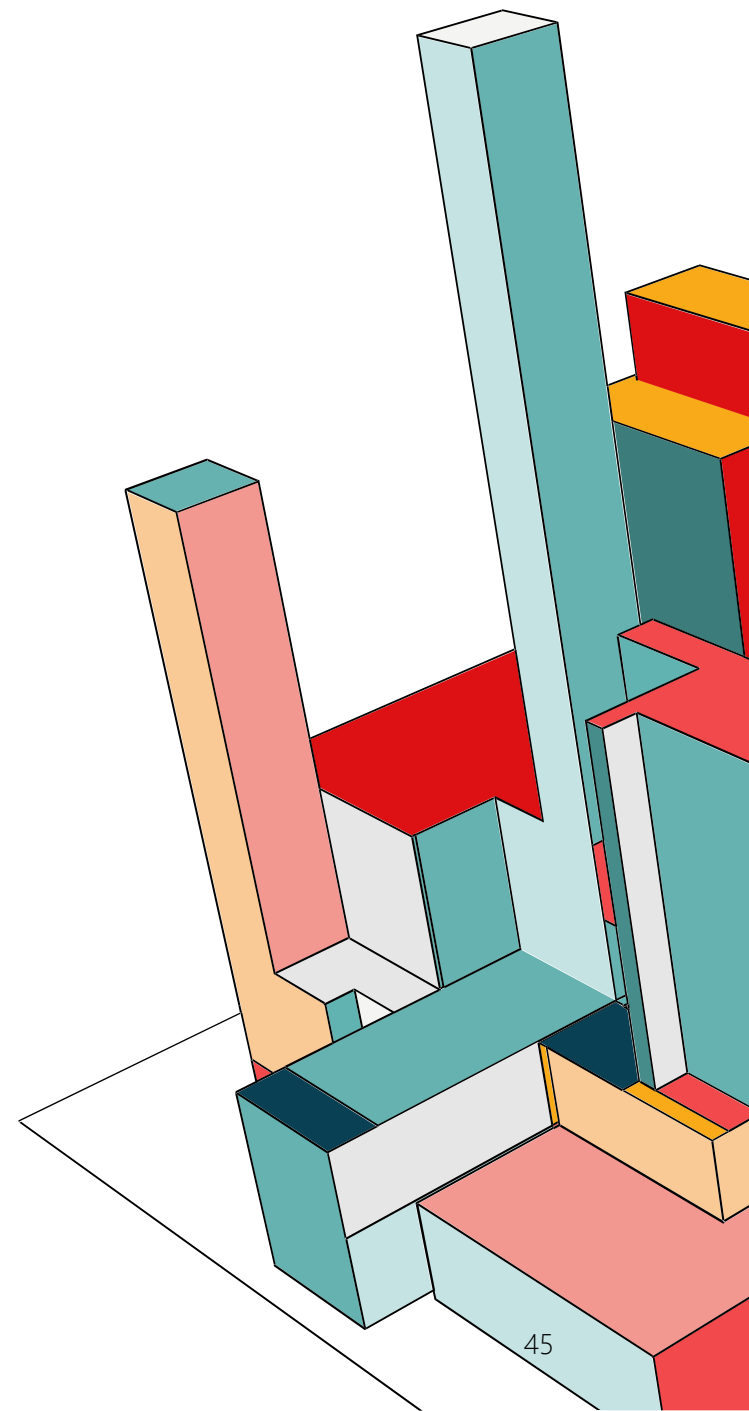
1. MAJOR version when you make incompatible API changes
2. MINOR version when you add functionality in a backwards compatible manner
3. PATCH version when you make backwards compatible bug fixes

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

## Introduction

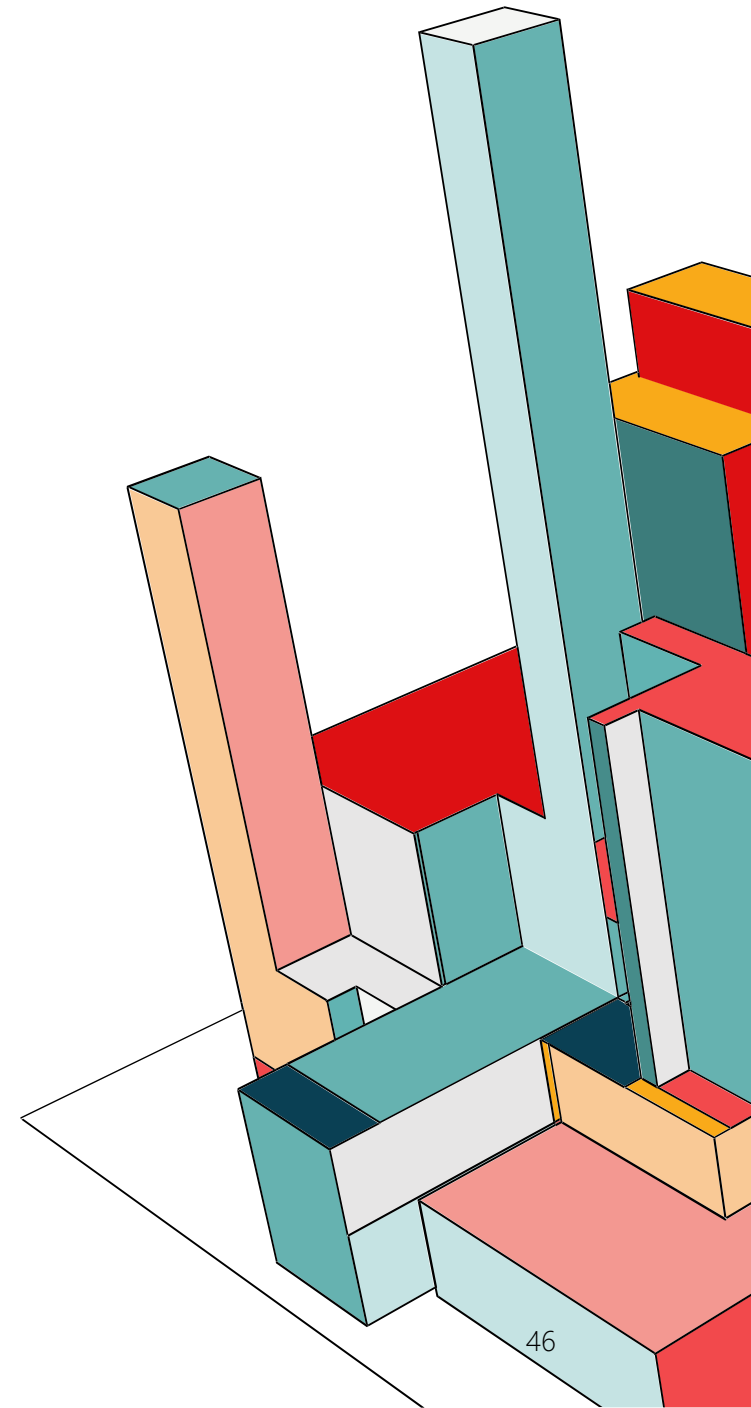
In the world of software management there exists a dreaded place called “dependency hell.” The bigger your system grows and the more packages you integrate into your software, the more likely you are to find yourself, one day, in this pit of despair.

Check out the official website for Semantic Versioning. <https://semver.org/>



# READINGS

- Chapter 18, 21. Software Engineering at Google by Titus Winters et al.
- Chapter 25.2. Software Engineering by Ian Sommerville, 10<sup>th</sup> edition.



# NEXT

- Code Quality
- Code Review