Build Systems

Martin Kellogg

Build Systems

Today’s agenda:

- Finish slides on Languages
  - performance, team and process factors, when to rewrite
- Reading Quiz
- What is a build system? How does one work?
- How to choose a build system + best practices

Build Systems

Today’s agenda:

- Finish slides on Languages
  - performance, team and pr
- Reading Quiz
- What is a build system? How does one work?
- How to choose a build system + best practices

Announcements:

- project plan grades are finalized (see Canvas)
- IP2 due today
- revised project plan due in one week (3/7)

Build Systems

Today’s agenda:

- **Finish slides on Languages**
    - performance, team and process factors, when to rewrite
- Reading Quiz
- What is a build system? How does one work?
- How to choose a build system + best practices

What impacts performance

- #1: **safety features enforced at run time**
  - dynamic type checking: type safety
  - **garbage collection**: memory safety
  - exceptions: segfault safety
- Also relevant: **optimizations**
  - **interpreted** languages almost always slower: no optimizing compiler
  - JITs (***just-in-time compilers***) can produce surprisingly fast code
    - e.g., Java Virtual Machine

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**
- So, why not **enforce safety at compile time** instead?

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**
- So, why not **enforce safety at compile time** instead?
    - requires **static analysis** (= there will be false positives)

Trade-off: safety features

- #1 performance problem: **safety features enforced at run time**
- So, why not **enforce safety at compile time** instead?
  - requires **static analysis** (= there will be false positives)
  - harder for programmers (trades off against **effort**)

Trade-off: safety features

- #1 performance problem: safety features enforced at run time
- So, why not **enforce safety at compile time** instead?
  - requires **static analysis** (= there will be false positives)
  - harder for programmers (trades off against effort)
    - the garbage collector in Java/Go/etc. is automatic
    - but writing Rust code requires follows its (complex) type discipline

Trade-off: safety features

- #1 performance problem: safety features enforced at run time
- So, why not **enforce safety at compile time** instead?
  - requires **static analysis** (= there will be false positives)
  - harder for programmers (trades off against effort)
    - the garbage collector in Java/Go/etc. is automatic
    - but writing Rust code requires follows its (complex) type discipline
  - bottom line: statically safe languages **can be faster**, but are **generally harder to program in**

How can programming languages differ?

- programming paradigm
- whether they have a type system
  - and, if they do, what kind of type system they have
- library support
  - the standard library is especially important
- performance
- **team/process factors**
  - how well do you know the language
  - how easy it'll be to hire other developers who do

Team/process factors

- **Learning** a new programming language takes time

Team/process factors

- **Learning** a new programming language takes time
  - Becoming productive shouldn’t take that long
    - but, this scales with how hard the language is to program in (+ access to mentors, etc.)

Team/process factors

- **Learning** a new programming language takes time
    - Becoming productive shouldn't take that long
        - but, this scales with how hard the language is to program in (+ access to mentors, etc.)
    - Becoming an expert takes a long time!

Team/process factors

- **Learning** a new programming language takes time
  - Becoming productive shouldn't take that long
    - but, this scales with how hard the language is to program in (+ access to mentors, etc.)
  - Becoming an expert takes a long time!
- If you need performance, you usually need **at least one expert**
  - cf. AWS employs some JVM experts to tune the garbage collector for AWS services that use Java

Team/process factors

- Learning a new programming language takes time
  - program

Implication: if you’re going to need an expert, make sure you have one! This often seriously limits your choice of languages in practice :(

  - Becoming an expert takes a long time!
- If you need performance, you usually need **at least one expert**
  - cf. AWS employs some JVM experts to tune the garbage collector for AWS services that use Java

Team/process factors

- Because learning a new language takes time, the **popularity** of a language is also a plus:

Team/process factors

- Because learning a new language takes time, the **popularity** of a language is also a plus:
    - it’s **easier to hire** new engineers who already know the language, and therefore can ramp up faster

Team/process factors

- Because learning a new language takes time, the **popularity** of a language is also a plus:
    - it’s **easier to hire** new engineers who already know the language, and therefore can ramp up faster
    - but this impact is relatively small over a typical engineer’s tenure at a company

Team/process factors

- Because learning a new language takes time, the **popularity** of a language is also a plus:
    - it’s **easier to hire** new engineers who already know the language, and therefore can ramp up faster
    - but this impact is relatively small over a typical engineer’s tenure at a company
- Implication: if all else is equal, **choose the more popular** language

When to rewrite

- the reading talked about moving a service from Go to Rust
  - why?

When to rewrite

- the reading talked about moving a service from Go to Rust
  - why? **Performance problems**.
- This is usually a risky thing to do:
  - you’re not building new features
  - integration problems
  - will the benefits be worth it?

When to rewrite

- the reading talked about moving a service from Go to Rust
  - why? **Performance problems**.
- This is usually a **risky thing** to do:
  - you’re not building new features
  - integration problems
  - will the benefits be worth it?

Implication: rewriting is a good idea if you’re confident that the benefits of the new language are worthwhile, but be cautious: it can expensive!

Takeaways

- there is a wider world of languages than just imperative and object-oriented (but those are the most popular)
  - learning to write functional code can make you a better programmer
- different programming languages have different trade-offs
  - performance vs safety vs ease of use vs …
- when starting a new project, think carefully about the requirements before choosing a language
- rewrite a project in a new language only after careful consideration

Build Systems

Today’s agenda:

- Finish slides on Languages
  - performance, team and process factors, when to rewrite
- **Reading Quiz**
- What is a build system? How does one work?
- How to choose a build system + best practices

Reading quiz: build systems

Q1: the “F5 key” in the title refers to which of the following:

A. a shortcut key in an IDE (integrated development environment)
B. a shortcut key used to refresh an email client
C. a shortcut key in the Gradle build system

Q2: **TRUE** or **FALSE**: the author argues that you don’t need to worry about how long it takes a new developer to start working productively on your project, because the productivity of team members with long tenure is increased by a good build system

Reading quiz: build systems

Q1: the “F5 key” in the title refers to which of the following:

A. a shortcut key in an IDE (integrated development environment)
B. a shortcut key used to refresh an email client
C. a shortcut key in the Gradle build system

Q2: **TRUE** or **FALSE**: the author argues that you don’t need to worry about how long it takes a new developer to start working productively on your project, because the productivity of team members with long tenure is increased by a good build system

Reading quiz: build systems

Q1: the "F5 key" in the title refers to which of the following:

A. a shortcut key in an IDE (integrated development environment)
B. a shortcut key used to refresh an email client
C. a shortcut key in the Gradle build system

Q2: **TRUE** or **FALSE**: the author argues that you don't need to worry about how long it takes a new developer to start working productively on your project, because the productivity of team members with long tenure is increased by a good build system

Reading quiz: build systems

Q1: the “F5 key” in the title refers to which of the following:

A. a shortcut key in an IDE

B. a shortcut key used to re

C. a shortcut key in the Gr

Q2: **TRUE** or **FALSE**: the aut

about how long it takes a nev

productively on your project, because the productivity of team members with long tenure is increased by a good build system

> **“How long does it take for you to get a new team member working productively on your project?** If the answer is more than one day, *you have a problem*. Specifically, you don't have a proper build process in place.”

Build Systems

Today's agenda:

- Finish slides on Languages
  - performance, team and process factors, when to rewrite
- Reading Quiz
- **What is a build system? How does one work?**
- How to choose a build system + best practices

What does a developer do?

What does a developer do?

- Get the source code
- Install dependencies
- Compile the code
- Run static analysis
- Generate documentation
- Run tests
- Create artifacts for customers
- Ship!

What does a developer do?

- Get the source code
- Install dependencies
- Compile the code
- Run static analysis
- Generate documentation
- Run tests
- Create artifacts for customers
- Ship!

Which should be handled manually?

What does a developer do?

- Get the source code
- Install dependencies
- Compile the code
- Run static analysis
- Generate documentation
- Run tests
- Create artifacts for customers
- Ship!

Which should be handled manually?

NONE!

From the reading

“Here's how most clients I work with build a project:

1. Open the IDE
2. Load the solution
3. Get latest
4. Press F5 (or CTRL+SHIFT+B)”

From the reading

“Here's how most clients I work with build a project:

1. Open the IDE
2. Load the solution
3. Get latest
4. Press F5 (or CTRL+SHIF

> “**The F5 key is not a build process**. It's a quick and dirty substitute. If that's how you build your software, I regret that I have to be the one to tell you this, but *your project is not based on solid software engineering practices.*”

From the reading

“Here's how most clients I wo

1. Open the IDE
2. Load the solution
3. Get latest
4. Press F5 (or CTRL+SHIF

“**The F5 key is not a build process**. It's a quick and dirty substitute. If that's how you build your software, I regret that I have to be the one to tell you this, but *your project is not based on solid software engineering practices.*”

Key objective of a build system: avoid this problem!

What to do instead?

What to do instead?

Orchestrate with a build system!

What is a build system?

What is a build system?

Definition: A ***build system*** is a tool for orchestrating software engineering tasks

What is a build system?

Definition: A ***build system*** is a tool for orchestrating software engineering tasks

- Getting the source code
- Installing dependencies
- Compiling the code
- Running static analysis
- Generating documentation
- Running tests
- Creating artifacts for customers
- Shipping!

What is a build system?

Definition: A ***build system*** is a tool for orchestrating software engineering tasks

- Getting the source code
- Installing dependencies
- Compiling the code
- Running static analysis
- Generating documentation
- Running tests
- Creating artifacts for customers
- Shipping!

A good build system handles all these

Tasks

Definition: a ***task*** is anything that the build system can do

Tasks

Definition: a *task* is anything that the build system can do

- Getting the source code
- Installing dependencies
- Compiling the code
- Running static analysis
- Generating documentation
- Running tests
- Creating artifacts for customers
- Shipping!

Tasks

Definition: a *task* is anything that the build system can do

- Getting the source code
- Installing dependencies
- Compiling the code
- Running static analysis
- Generating documentation
- Running tests
- Creating artifacts for customers
- Shipping!

All tasks!

Tasks

- #1 thing to know about tasks: **tasks are code**, too!

Tasks

- #1 thing to know about tasks: **tasks are code**, too!
  - Should be checked into version control
  - Should be code-reviewed
  - Should be tested

Tasks

- #1 thing to know about tasks: **tasks are code**, too!
  - Should be checked into version control
  - Should be code-reviewed
  - Should be tested
- Tasks also commonly have **dependencies**

Tasks

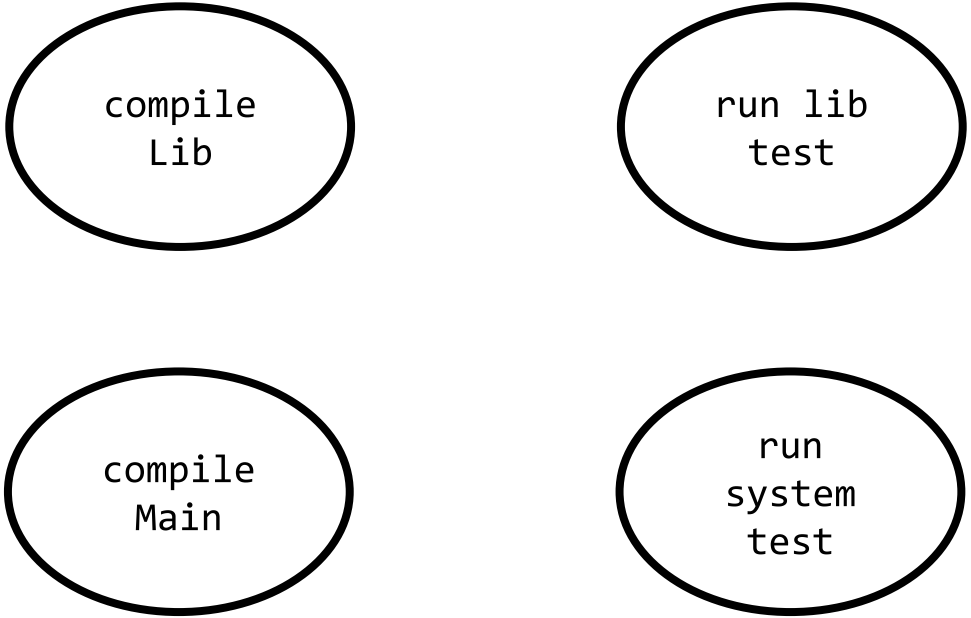
- #1 thing to know about tasks: **tasks are code**, too!
  - Should be checked into version control
  - Should be code-reviewed
  - Should be tested
- Tasks also commonly have **dependencies**
  - Dependency management is a key build system responsibility!

Dependencies between tasks

```
> ls src/

Lib.java   LibTest.java   Main.java     SystemTest.java
```

Dependencies between tasks



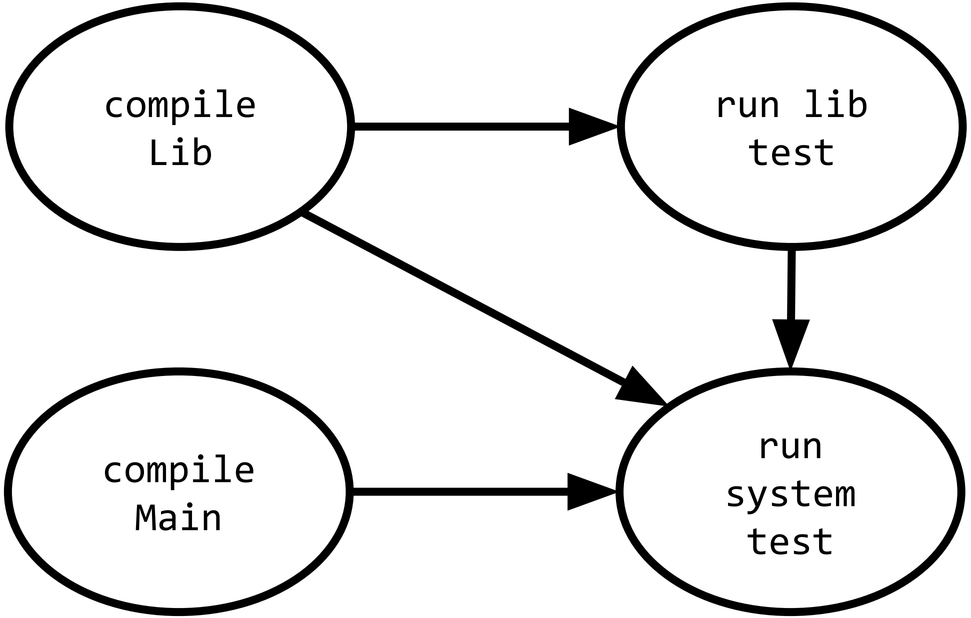
compile Lib

run lib test

compile Main

run system test

Dependencies between tasks



Dependencies between tasks

- A large project may have thousands of tasks

Dependencies between tasks

- A large project may have thousands of tasks
  - What order to run in?
  - How to speed up?

Dependencies between tasks

- A large project may have thousands of tasks
  - **What order to run in?**
  - How to speed up?

Determining task ordering

- Dependencies between tasks form a directed acyclic graph

Determining task ordering

- Dependencies between tasks form a directed acyclic graph

Topological sort!

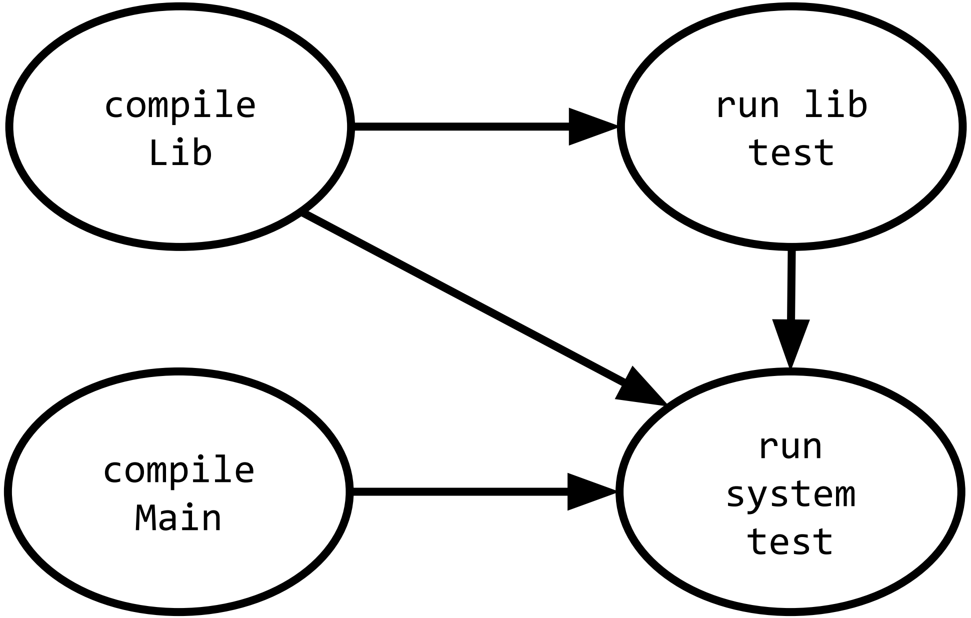
Topological sort

- Any ordering on the nodes such that all dependencies are satisfied

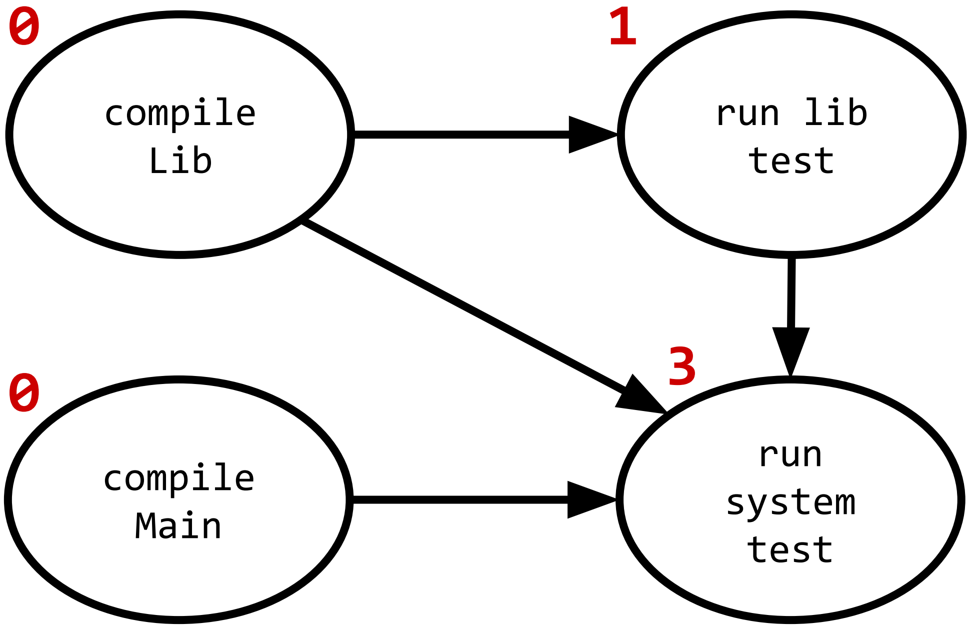
Topological sort

- Any ordering on the nodes such that all dependencies are satisfied
- Implement by computing ***indegree*** (number of incoming edges) for each node

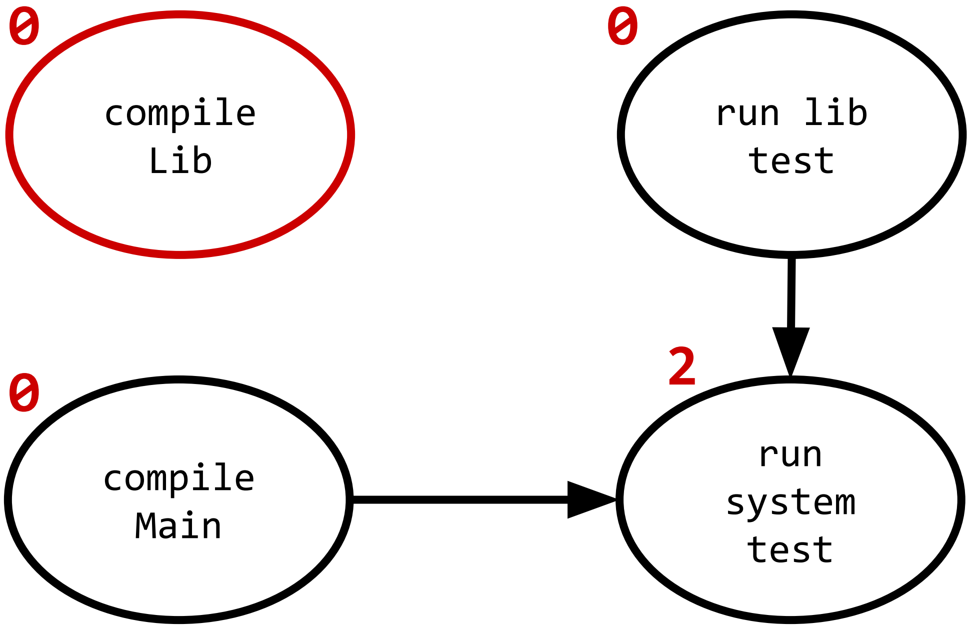
Topological sort



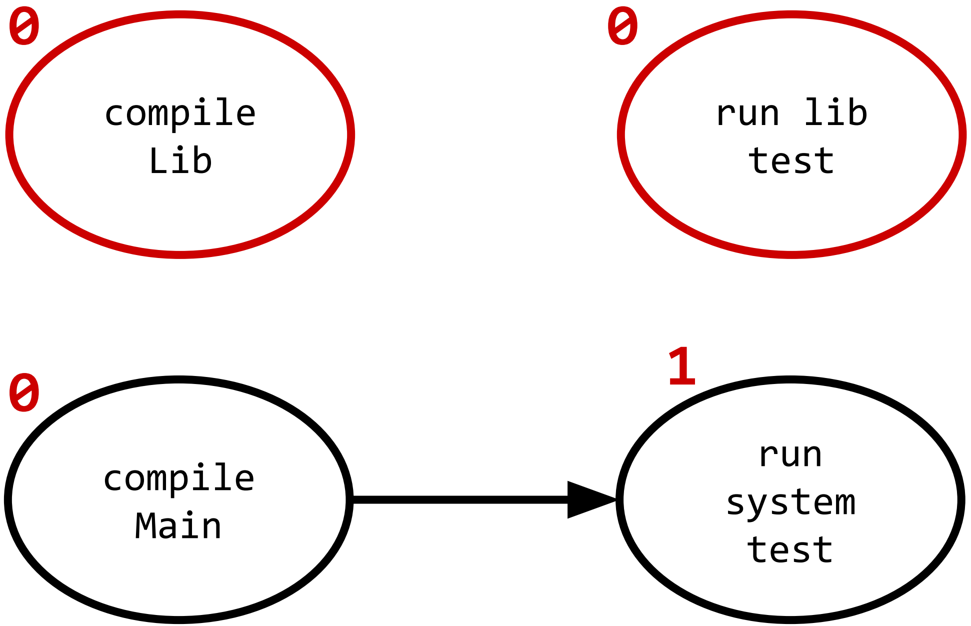
Topological sort



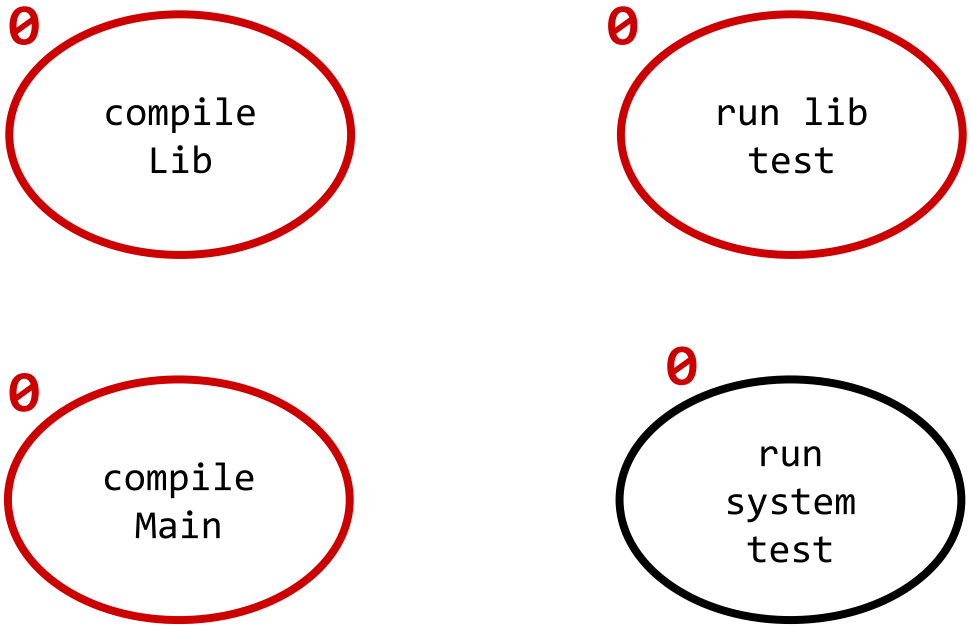
Topological sort



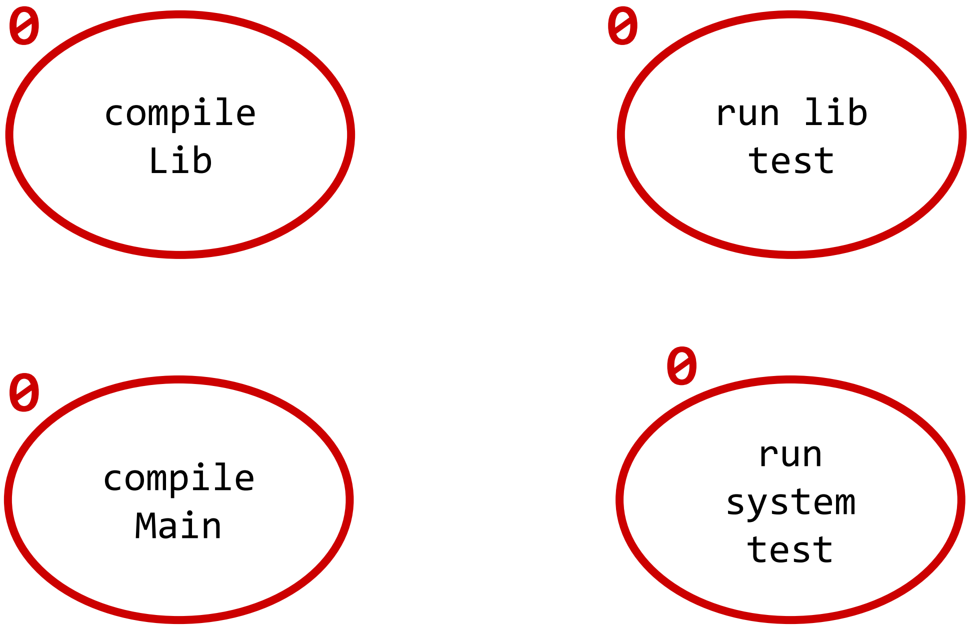
Topological sort



Topological sort



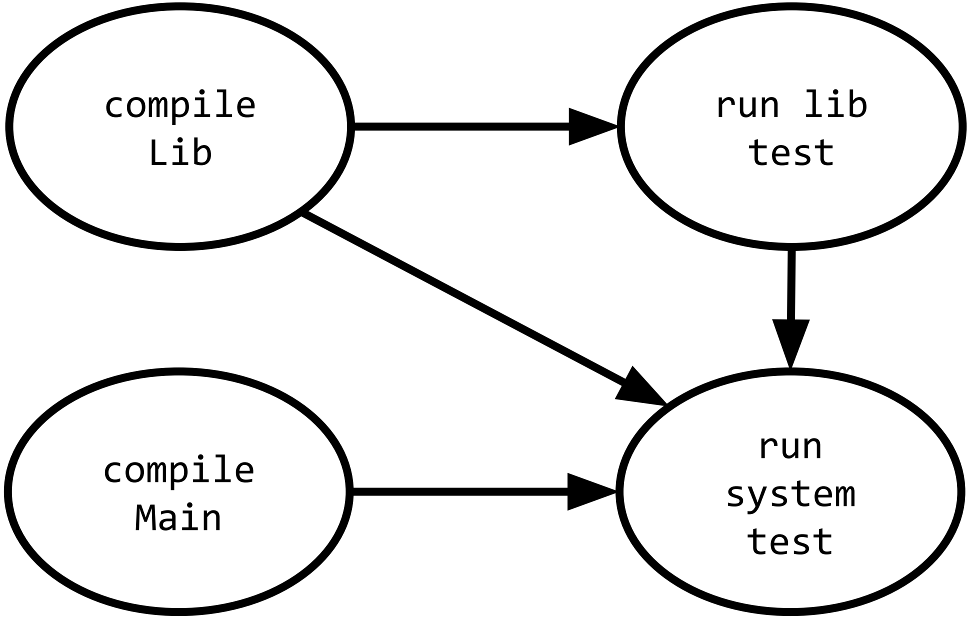
Topological sort



Topological sort

Valid sorts:

1. compile Lib, run lib test, compile Main, run system test

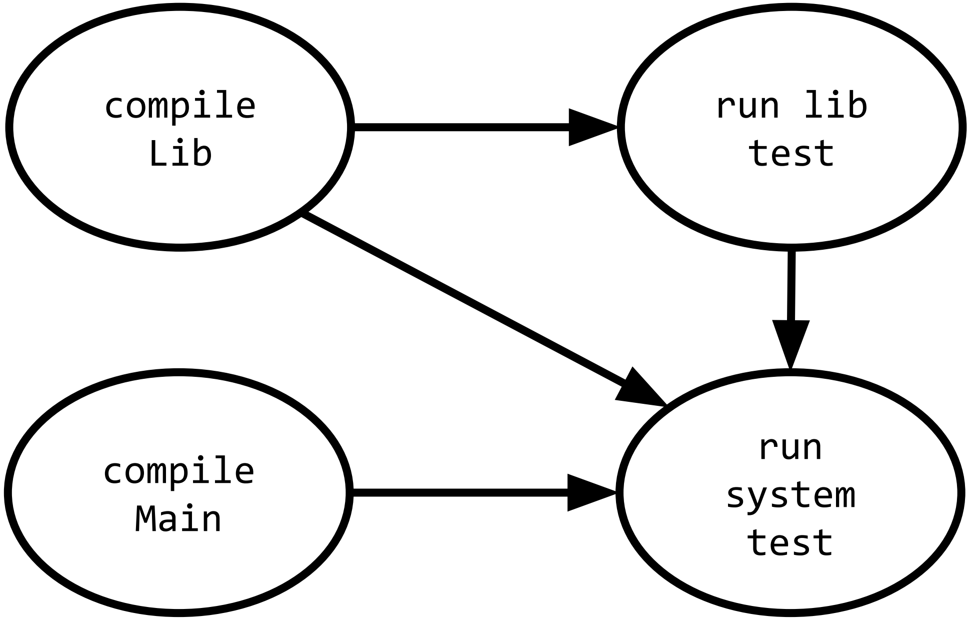


Topological sort

Valid sorts:

1. compile Lib, run lib test, compile Main, run system test

2. compile Main, compile Lib, run lib test, run system test



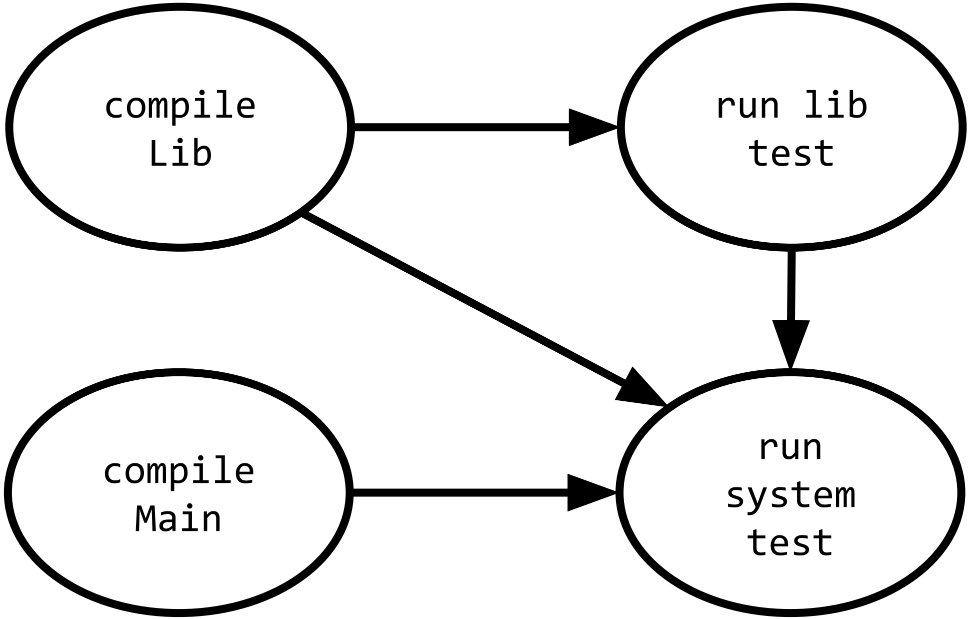
Topological sort

Valid sorts:

1. compile Lib, run lib test, compile Main, run system test

2. compile Main, compile Lib, run lib test, run system test

3. compile Lib, compile Main, run lib test, run system test



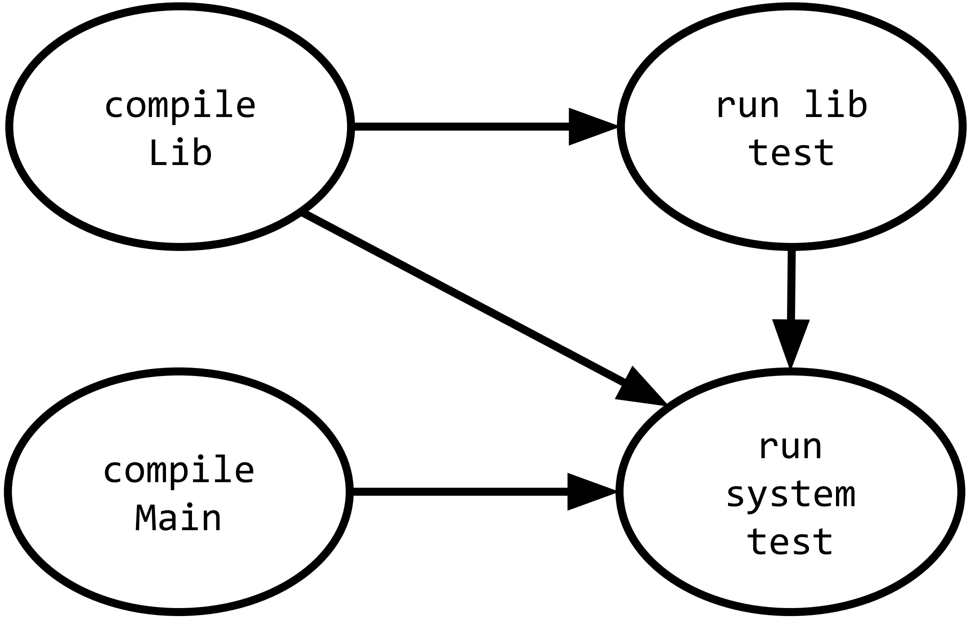
Topological sort

Valid sorts:

1. compile Lib, run lib test, compile Main, run system test

2. compile Main, compile Lib, run lib test, run system test

3. compile Lib, compile Main, run lib test, run system test



Why is this order silly?

Examples of modern build systems

`gradle`

https://gradle.org/

Apache’s open-source successor to ant, maven

`bazel`

https://www.bazel.build/

Google’s internal build tool, open-sourced

Example task: `gradle`

```
task reformat(type: Exec, dependsOn: getCodeFormatScripts, group: 'Format') {
   description 'Format the Java source code'
   // jdk8 and checker-qual have no source, so skip
   onlyIf { !project.name.is('jdk8') && !project.name.is('checker-qual') }
   executable 'python'
   doFirst {
       args += "${formatScriptsHome}/run-google-java-format.py"
       args += "--aosp" // 4 space indentation
       args += getJavaFilesToFormat(project.name)
   }
}
```

Example task: `gradle`

```
task reformat(type: Exec, dependsOn: getCodeFormatScripts, group: 'Format') {
   description 'Format the Java source code'
   // jdk8 and checker-qual have no source, so skip
   onlyIf { !project.name.is('jdk8') && !project.name.is('checker-qual') }
   executable 'python'
   doFirst {
       args += "${formatScriptsHome}/run-google-java-format.py"
       args += "--aosp" // 4 space indentation
       args += getJavaFilesToFormat(project.name)
   }
}
```

kind of rule

Example task: gradle

```
task reformat(type: Exec, dependsOn: getCodeFormatScripts, group: 'Format') {
    description 'Format the Java source code'
    // jdk8 and checker-qual have no source, so skip
    onlyIf { !project.name.is('jdk8') && !project.name.is('checker-qual') }
    executable 'python'
    doFirst {
        args += "${formatScriptsHome}/run-google-java-format.py"
        args += "--aosp" // 4 space indentation
        args += getJavaFilesToFormat(project.name)
    }
}
```

explicitly specified dependencies

Example task: gradle

```
task reformat(type: Exec, dependsOn: getCodeFormatScripts, group: 'Format') {
   description 'Format the Java source code'
   // jdk8 and checker-qual have no source, so skip
   onlyIf { !project.name.is('jdk8') && !project.name.is('checker-qual') }
   executable 'python'
   doFirst {
       args += "${formatScriptsHome}/run-google-java-format.py"
       args += "--aosp" // 4 space indentation
       args += getJavaFilesToFormat(project.name)
   }
}
```

code!

Example task: bazel

```
java_binary(
    name = "dux",
    main_class = "org.dux.cli.DuxCLI",
    deps = ["@google_options//:compile",
            "@checker_qual//:compile",
            "@google_cloud_storage//:compile",
            "@slf4j//:compile",
            "@logback_classic//:compile"],
    srcs = glob(["src/org/dux/cli/*.java",
                 "src/org/dux/backingstore/*.java"),
)
```

Example task: bazel

```
java_binary(
    name = "dux",
    main_class = "org.dux.cli.DuxCLI",
    deps = ["@google_options//:compile",
            "@checker_qual//:compile",
            "@google_cloud_storage//:compile",
            "@slf4j//:compile",
            "@logback_classic//:compile"],
    srcs = glob(["src/org/dux/cli/*.java",
                 "src/org/dux/backingstore/*.java"),
)
```

kind of rule

Example task: bazel

```
java_binary(
    name = "dux",
    main_class = "org.dux.cli.DuxCLI",
    deps = ["@google_options//:compile",
            "@checker_qual//:compile",
            "@google_cloud_storage//:compile",
            "@slf4j//:compile",
            "@logback_classic//:compile"],
    srcs = glob(["src/org/dux/cli/*.java",
                 "src/org/dux/backingstore/*.java"),
)
```

explicitly specified dependencies

Example task: `bazel`

```
java_binary(
    name = "dux",
    main_class = "org.dux.cli.DuxCLI",
    deps = ["@google_options//:compile",
            "@checker_qual//:compile",
            "@google_cloud_storage//:compile",
            "@slf4j//:compile",
            "@logback_classic//:compile"],
    srcs = glob(["src/org/dux/cli/*.java",
                 "src/org/dux/backingstore/*.java"]),
)
```

explicitly specified
dependencies
(also bazel tasks)

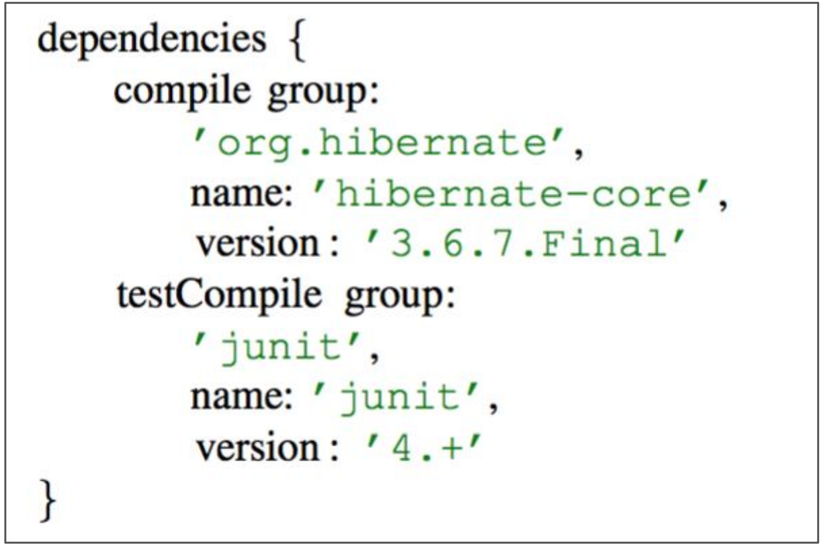
External and internal dependencies

- A list of tasks (internal) or libraries (external)

External and internal dependencies

- A list of tasks (internal) or libraries (external)

```
deps = ["@google_options//:compile",
        "@checker_qual//:compile",
        "@google_cloud_storage//:compile",
        "@slf4j//:compile",
        "@logback_classic//:compile"],
```



```
dependencies {
    compile group:
        'org.hibernate',
        name: 'hibernate-core',
        version : '3.6.7.Final'
    testCompile  group:
        'junit',
        name: 'junit',
        version : '4.+'
}
```

https://docs.gradle.org/current/userguide/artifact_dependencies_tutorial.html

Why list dependencies?

- Reproducibility!

Why list dependencies?

- Reproducibility!
- ***Hermetic builds***: “they are insensitive to the libraries and other software installed on the build machine”[1]

[1] https://landing.google.com/sre/sre-book/chapters/release-engineering/

Why list dependencies?

- Reproducibility!
- ***Hermetic builds***: “they are insensitive to the libraries and other software installed on the build machine”[1]
  - critical if you want to get new developers working quickly (remember the reading!)
  - useful for debugging problems users encounter with old versions (can always get back to exactly the code they’re using)
  - prevents “it works on my machine” syndrome

[1] https://landing.google.com/sre/sre-book/chapters/release-engineering/

Dependencies between tasks

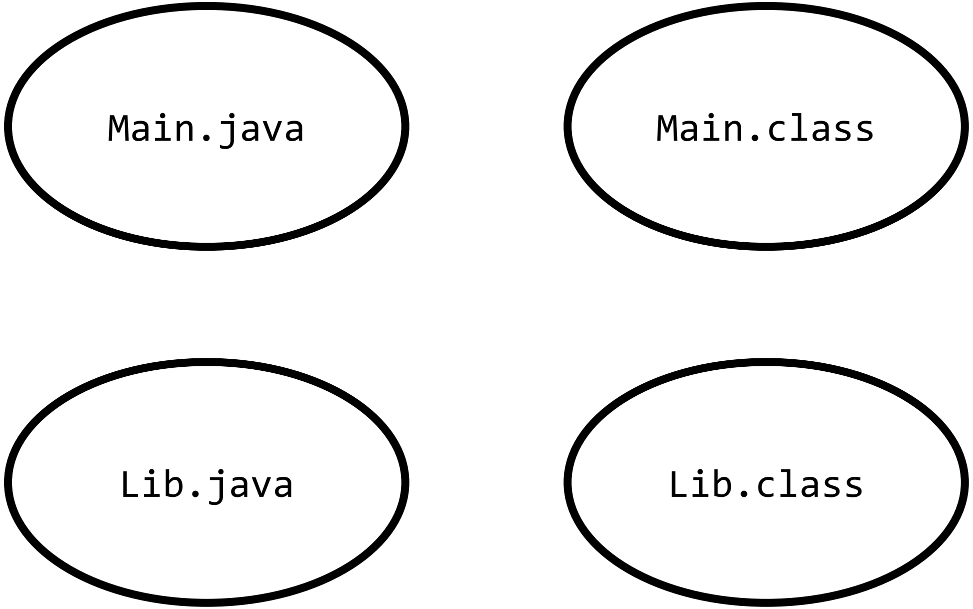
- A large project may have thousands of tasks
  - What order to run in?
  - **How to speed up?**

How to speed up builds?

How to speed up builds?

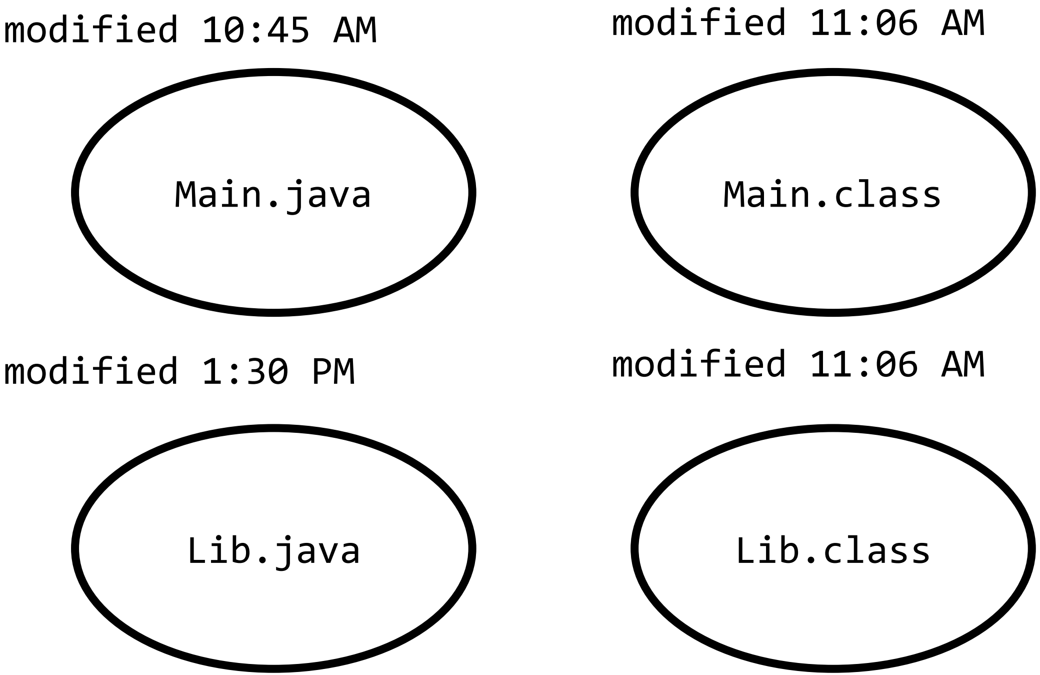
- **Incrementalize** - only rebuild what you have to

Incrementalization



Incrementalization

1:31 PM



modified 10:45 AM

Main.java

modified 11:06 AM

Main.class

modified 1:30 PM

Lib.java

modified 11:06 AM

Lib.class

Incrementalization

- Compute hash codes for inputs to each task
- When about to execute a task, check input hashes - if they match the last time the task was executed, skip it!

How to speed up builds?

- **Incrementalize** - only rebuild what you have to
- Execute many tasks in **parallel**
- **Cache** artifacts in the cloud

How do build systems differ

How do build systems differ

- Scheduling algorithm

How do build systems differ

- Scheduling algorithm
    - We’ve already seen topological scheduling (used by e.g. make), which is a **static** scheduling algorithm

How do build systems differ

- Scheduling algorithm
  - We’ve already seen topological scheduling (used by e.g. make), which is a **static** scheduling algorithm
  - **Dynamic** scheduling algorithms are also possible

How do build systems differ

- Scheduling algorithm
  - We’ve already seen topological scheduling (used by e.g. make), which is a **static** scheduling algorithm
  - **Dynamic** scheduling algorithms are also possible
    - **Key idea**: compute what dependencies are necessary as you go

How do build systems differ

- Scheduling algorithm
    - We’ve already seen topological scheduling (used by e.g. make), which is a **static** scheduling algorithm
    - **Dynamic** scheduling algorithms are also possible
        - **Key idea**: compute what dependencies are necessary as you go
        - this is how e.g., Bazel actually schedules tasks

How do build systems differ

- Rebuilding strategy

How do build systems differ

- Rebuilding strategy
  - We’ve seen two:

How do build systems differ

- Rebuilding strategy
  - We’ve seen two:
    - a ***dirty bit*** strategy (make’s timestamps)

How do build systems differ

- Rebuilding strategy
  - We’ve seen two:
    - a ***dirty bit*** strategy (make’s timestamps)
    - a ***verifying trace*** strategy (storing hashes of each object)

How do build systems differ

- Rebuilding strategy
    - We’ve seen two:
        - a ***dirty bit*** strategy (make’s timestamps)
        - a ***verifying trace*** strategy (storing hashes of each object)
    - Other options:
        - ***constructive traces***: store all intermediate objects (usually in the cloud) along with the hashes of the **inputs** used to produce them. If we ever see the same input hashes again, just return the intermediate object

How do build systems differ

- How are tasks expressed?

How do build systems differ

- How are tasks expressed?
    - traditionally **declarative** (e.g., make, Ant, Maven)
        - “declarative” = you tell the build system what you want, it figures out how to build that thing

How do build systems differ

- How are tasks expressed?
    - traditionally **declarative** (e.g., make, Ant, Maven)
        - “declarative” = you tell the build system what you want, it figures out how to build that thing
    - most modern build systems have **scripting languages**
        - e.g., Groovy in Gradle, Starlark in Bazel, etc.
        - enables us to write tasks as if they are other code

How to choose a build system

How to choose a build system

High level idea: same rules apply to choosing a language

How to choose a build system

High level idea: same rules apply to choosing a language

- **don’t change what’s already there** unless there is a good reason

How to choose a build system

High level idea: same rules apply to choosing a language

- **don’t change what’s already there** unless there is a good reason
- **follow convention** and prefer the tooling that’s “idiomatic” to your language
  - e.g., use Gradle or Maven when working in Java

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
    - common causes include:

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven's per-module incremental compilations)

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven’s per-module incremental compilations)
    - lack of support for artifact caching (= **cloud builds**)

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven’s per-module incremental compilations)
    - lack of support for artifact caching (= **cloud builds**)
    - build has become too complex for a declarative task language

When to switch build systems

- developers rarely choose to change build systems **except** when **build performance** is a problem
  - common causes include:
    - poor incrementalization (e.g., Maven’s per-module incremental compilations)
    - lack of support for artifact caching (= **cloud builds**)
    - build has become too complex for a declarative task language
  - most projects keep the same build system **forever**

Best practices

- Automate everything

Best practices

- Automate everything
- Always use a build tool

Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)

Best practices

Your CI server is a good place to test that your build is hermetic. **Standard practice**: spin up a new CI server for **each build**.

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)

Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)
- Don’t depend on anything that’s not in the build file (hermetic)

Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)
- Don’t depend on anything that’s not in the build file (hermetic)
- Don’t break the build

Best practices

- Automate everything
- Always use a build tool
- Have a build server that builds and tests your code on every commit (continuous integration)
- Don’t depend on anything that’s not in the build file (hermetic)
- Don’t break the build

A **common mistake to avoid**: allowing the CI server to fail for a long time because “we know what the problem is.” Don’t do this: leads to complacency, missing real bugs.