IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# Automated software construction

Fabien Dagnat & J.-C. Bach

DCL – build – 2025-2026

# Work under Creative Commons BY-SA license


You are **free**:

- to **use**, to **copy**, to **distribute** and to **transmit** this creation to the public;
- to **adapt** this work.

Under the following terms:

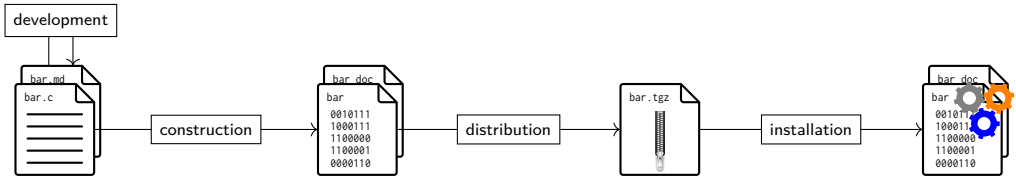
- **Attribution** (**BY**): you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **ShareAlike** (**SA**): if you modify, transform, alter, adapt or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

See `http://creativecommons.org/licenses/by-sa/2.0/`

# Important notes

- If you do not understand something, please ask your questions. *We cannot answer the questions you do not ask...*
- If you disagree with us, please say it (politely)
- People don't learn computer science by only reading few academic slides: practicing is fundamental

# Software deployment




- We're looking for automation to deliver better and faster
- Several possible actions
  - compilation, tests, code generation, execution, formating...
  - archive and installer, documentation, release...
  - combinations of several actions

# Different kinds of tools

*System Package Manager*

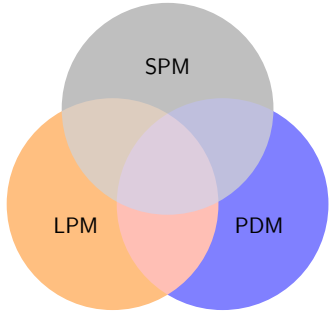
package (parts of application) management (installation, upgrade, ...), often distributed as binaries

ex: `apt`, macos app store, homebrew

*Language Package Manager*

building of language-specific applications and libraries

ex: go tools, npm, opam, cargo, ...

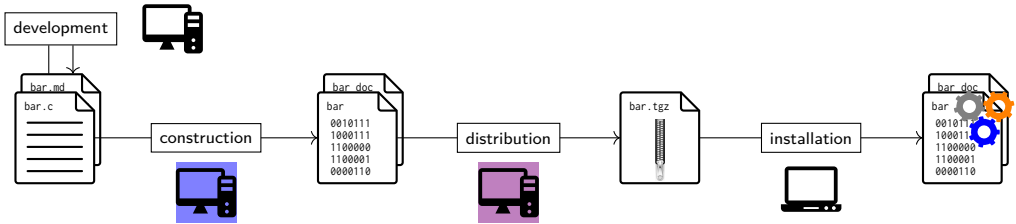



*Project Dependency Manager*

building of a project

ex: maven, gradle, dune, ...

# Different machines




- Several machines are involved, they may have various OS
- The build machine, must be able to build binaries for the user's machine
  - notion of cross-compilation

# Different uses

Production can be

- on demand
  - by invoking the build tool explicitly
  - for instance by the developer, by a package maintainer, ...
- scheduled
  - a server launches a build at a certain frequency : every night, ...
  - for instance, for the regular production of intermediate versions (ex : *nightly versions*)
- triggered
  - construction is launched by an external event
  - for example, build each new `commit` in a certain branch

# The notion of internal dependency

- Complete rebuilding can be costly

⇒ The aim is to minimize unnecessary execution of actions

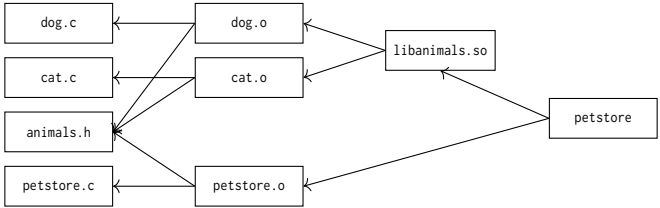
- For each action
  - the inputs must be known
  - the action is re-executed only if one of its inputs has changed
- One must know
  - the dependencies between artifacts (often files), this is called a *dependency graph*
  - the actions that produce these artifacts

⇒ The actions to be performed and their (partial) order can be deduced from this.

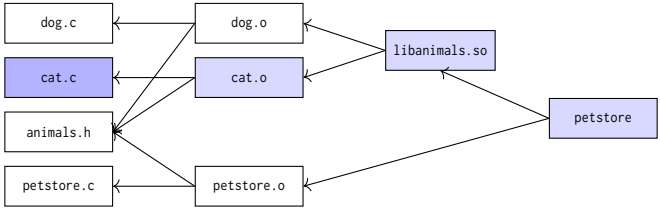
- Generally a dependency graph **without circuit**

Algo: circuit search and topological sorting (DFS-based algorithms)

# An example of a dependency graph

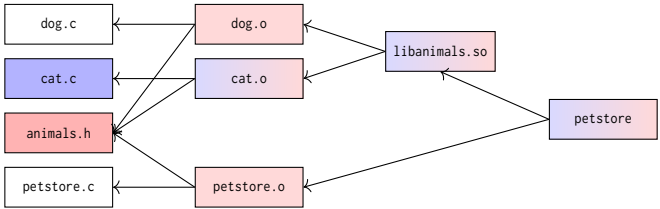


# An example of a dependency graph




- If `cat.c` is modified
- Then one must rebuild `cat.o`, `libanimals.so` and `petstore` in this order

# An example of a dependency graph




- If `cat.c` is modified
- Then one must rebuild `cat.o`, `libanimals.so` and `petstore` in this order
- If `animals.h` is modified
- Then one must rebuild
  - `dog.o`, `cat.o` et `petstore.o` independently
  - then `libanimals.so`
  - and finally `petstore`

# An example of a tool: make

- Is configured with a `Makefile` file

Product

```
dog.o: dog.c animals.h
    gcc -fPIC -c dog.c
```

Depends on

Actions

# An example of a tool: make

- Is configured with a `Makefile` file

Product

```
dog.o: dog.c animals.h          Depends on
    gcc -fPIC -c dog.c          Actions
cat.o: cat.c animals.h
    gcc -fPIC -c cat.c
petstore.o: petstore.c animals.h
    gcc -c petstore.c
libanimals.so:
    gcc -shared -o libanimals.so dog.o cat.o
petstore: animals.so petstore.o
    gcc -o petstore petstore.o -L. -lanimals
```

- Uses file change dates
- Portability of complex actions

# Differents tool categories

- A la `make`
  - action = shell script
- A la `ant`
  - specific action language + Java *framework*

  ⇒ actions are portable but it is a dedicated language
- Build files generation: `automake`, `CMake`

  ⇒ higher level and portable language, reuse of a well-known tool
- Embedded in a script language: `gradle`, `scons`, ...

  ⇒ easier to write tailored and portable actions
- IDE: eclipse, ...

  ⇒ well-integrated for the developer, but a fixed set of actions
- Sometimes, combinations are possible: `gradle`/eclipse

# Expected properties of building tools

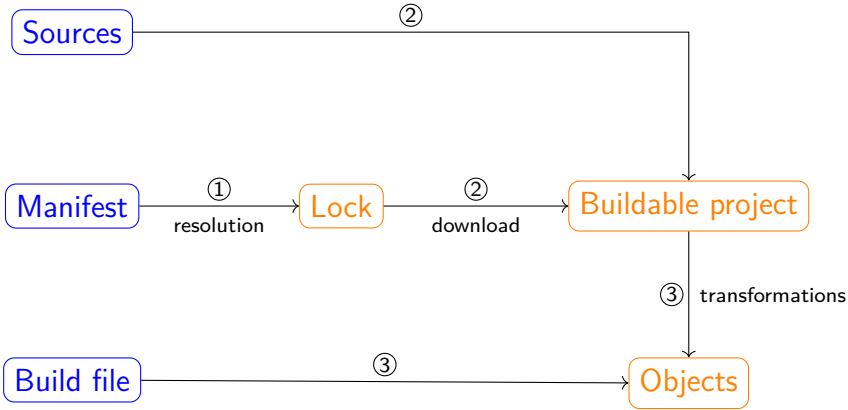
- Correction
  - the artifacts produced must correspond to what is expected
- Efficiency
  - optimal re-execution of actions
  - dependency management is not too costly
- Flexibility
  - the tool can easily produce several versions of artifacts
  - the construction machines can be changed easily
- Reproductibilty (possibly)
  - Identical action executions produce identical artifacts

# Building components

- Project elements (*sources*)
  - source code, resources (images, icons, ...)
- Specification of the external dependencies (*manifest*)
  - other source code, libraries, ...
- Specification of the build tasks (*build file*)
- Exact specification of external dependencies (*lock*)
- The buildable project
  - project elements
  - all dependencies necessary for the build
- The built project (*objects*)

Elements in blue are typically produced by the developer
Elements in orange are typically produced by the build tool
Sometimes, the *manifest* is in the build file

# The build process



# Example

- Sources
  1. Java files, ...
  2. database model
  3. initial database data
  4. test cases
- Objects
  - A generated Java code for DB access
  - B compiled Java classes (`.class` files)
  - C a `jar` archive of Java code
  - D database schema
  - E initialized database
  - F test execution report
- Transformations

$2 \rightarrow$ A,E generation of DB schema & Java code

1,A $\rightarrow$ B Java code compilation

B $\rightarrow$ C production of a code archive

D,3 $\rightarrow$ E DB initialization

C,4,E $\rightarrow$ F test execution and report generation

# Difference between *manifest* and *lock*

- The *manifest*
  - uses a language to define flexible dependencies
  - e.g. Gradle : `'junit:junit:4.+'`
- The *lock*
  - contains the exact dependency choices
  - e.g.: the version `4.12` of JUnit
- Flexible dependencies are necessary to manage transitive dependencies
  - for example, we depend on `A` and we depend on `B`, which depends on `A`
  - if the version of `A` is forced and `B` changes version of `A`
  - there may be a conflict
- The exact version is necessary to be able to reproduce the build

# Packaging

- For distribution to the user, there are several approaches
  - Standard archives: `zip`, `tar.gz`, `jar`, ...
  - SPM : `deb`, `rpm`, ...
  - Graphical installation tools
  - Specific installation tools

# Continuous Integration (CI)

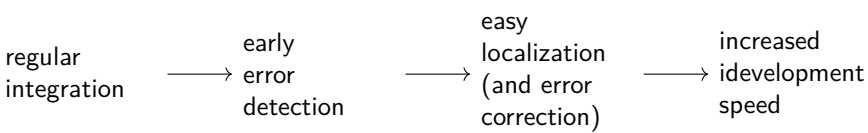
## Definition

Development practice that frequently integrates developers' work into a shared repository and that ensures the repository is in a workable state.

- Each integration triggers build and verification tasks
- Several iterative and incremental integrations *vs* a final integration

$\Rightarrow$ early detection of problems

$\Rightarrow$ improves quality



# Continuous *

- integration : version management + automated build and testing
- delivery : continuous integration continue + delivery (*releases*)
- deployment : continuous delivery with automated deployment

# CI toolset requirements

CI requirements

- sharing code and other artefacts
- integrating often
- testing integration
- communicating build results

Needed tools

- control version systems (Git, Subversion, ...)
- test frameworks (JUnit, ...), analysers (PMD, ...)
- build systems (Make, Ant, Gradle, ...)
- dashboard, emails, RSS feeds or chat systems with notifications, bug reports, ...

# CI systems: *one tool to unite them all*

- aggregate other tools
- run the integration processes
- distribute the integration tasks to workers
- present build results


# CI IRL

Demo: Jenkins or Gitlab-ci (if time permits)

# Some references

- https://en.wikipedia.org/wiki/List_of_build_automation_software
- https://medium.com/@sdboyer/so-you-want-to-write-a-package-manager-4ae9c17d9527
- https://media.ccc.de/v/camp2015-6657-how_to_make_your_software_build_reproducibly

# Gentle reminder

## ⚠ Important notes

- If you do not understand something, please ask your questions. *We cannot answer the questions you do not ask...*
- If you disagree with us, please say it (politely)
- People don't learn computer science by only reading few academic slides: practicing is fundamental