04834580 Software Engineering (Honor Track) 2025-26

UNIX Environment

Sergey Mechtaev
mechtaev@pku.edu.cn
School of Computer Science, Peking University

Small Tools That Compose

Doug McIlroy summarised the UNIX philosophy as [1]:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

This lecture surveys the classic tools that embody this philosophy:

- `diff` / `patch` — comparing and updating files,
- `make` — automating builds,
- `sed`, `sort`, `uniq`, `cut`, `find`, `xargs` — text and file plumbing,
- `awk` — pattern-action programming on records,
- `jq` — the same idea, applied to JSON.

Composition By Pipes

A typical UNIX one-liner is a pipeline of small programs. Example: list the five most frequent shell commands from the user's history:

```
$ history | awk '{print $2}' | sort | uniq -c | sort -rn |
    head -5
     142 git
      97 cd
      63 ls
      41 vim
      38 make
```

Each utility is small and reusable; the power comes from *composition*, not from a monolithic "history analyser" program.

What is the `diff` utility?

北京大学

- `diff` [2] compares two text files *line by line* and prints the smallest set of changes that turns the first into the second.
- It is the foundation of code review, version control, and software distribution: every `git diff`, every patch posted to a mailing list, every code-review comment is built on this idea.
- Output formats: *normal* (default), *context* (`-c`), *unified* (`-u`), *ed-script* (`-e`), *side-by-side* (`-y`).

Syntax of `diff`

北京大学

Syntax

diff [options] file1 file2

Frequently used options:

- –u — unified format (preferred for code review).
- –c — context format (older, more verbose).
- –r — recursive comparison of two directories.
- –N — treat absent files as empty (useful with `-r`).
- –w — ignore all whitespace.
- –i — ignore case.
- ––color — colourise the output for the terminal.

Default Output: A Tiny Example

北京大學

Let `file1.txt` and `file2.txt` be:

file1.txt

```
apple
banana
cherry
date
```

file2.txt

```
apple
blueberry
cherry
date
```

Plain diff file1 . txt file2 . txt produces an *ed-script*:

```
2c2
< banana
---
> blueberry
```

Read it as: "**c**hange line 2 of file 1 into line 2 of file 2".

Why Unified Diff?

北京大学

- Compact: only changed lines plus a few lines of context.
- Symmetric: removals (−) and additions (+) line up visually.
- Universal: this is what Git, GitHub, Gerrit, mailing-list patches and code reviewers all use.

Hunk header

`@@ -l,s +l,s @@`

- `-l,s`: starting line and length of the hunk in the original file.
- `+l,s`: starting line and length in the new file.

Unified Diff: Single-Hunk Example

北京大学

Running `diff -u file1.txt file2.txt` on the previous files:

```
--- file1.txt   2026-05-14 09:00:00
+++ file2.txt   2026-05-14 09:00:01
@@ -1,4 +1,4 @@
 apple
-banana
+blueberry
 cherry
 date
```

- `---`/`+++`: old/new file with timestamps.
- `@@ -1,4 +1,4 @@`: this hunk starts at line 1 in both files and spans 4 lines.
- Lines starting with ` ` (space) are unchanged *context*.

Unified Diff: Multi-Hunk Example

北京大学

file1.txt

```
apple
banana
cherry
date
elderberry
fig
grape
```

file2.txt

```
apple
blueberry
cherry
dragonfruit
elderberry
grape
```

diff —u file1.txt file2.txt

```
@@ -1,7 +1,6 @@
 apple
-banana
+blueberry
 cherry
-date
+dragonfruit
 elderberry
-fig
 grape
```

From `diff` to `patch`

- `diff` *produces* a patch (a description of changes).
- `patch` *applies* a patch to recreate the new file from the old.
- Together they let people exchange small text descriptions instead of whole files — the basis of distributed development.

Producing and Applying a Patch

Generate the patch

```
$ diff -u original.txt modified.txt > change.patch
```

Apply it (in place)

```
$ patch original.txt < change.patch
```

Useful options:

- patch --dry-run < change.patch — simulate without modifying files.
- patch -R < change.patch — reverse a previously applied patch.
- patch -p1 < change.patch — strip one leading directory component from filenames (Git-style patches).

Patch Resilience

北京大学

`patch` is intentionally fuzzy: it locates each hunk by its context rather than by absolute line numbers.

- If the original file has been edited slightly, `patch` can still find the right place using the surrounding context lines.
- If a hunk cannot be applied cleanly, the rejected portions are saved in a `.rej` file for manual resolution.

```
$ patch < change.patch
patching file original.txt
Hunk #1 succeeded at 12 (offset 2 lines).
Hunk #2 FAILED at 47.
1 out of 2 hunks FAILED -- saving rejects to file original.txt
    .rej
```

The Diff Problem

Given two sequences $A = a_1 \dots a_N$ and $B = b_1 \dots b_M$ (lines of two files), find the **shortest edit script** — the smallest set of insertions and deletions that transforms A into B.

Two important properties:

- Output should be *minimal* so the patch is small.
- It should "look right" to humans — preserve large unchanged regions.

Eugene Myers (1986) [3] gave an algorithm with running time $\mathcal{O}((N + M) \cdot D)$, where D is the length of the shortest edit script. For two nearly identical files, D is tiny — so it is fast in practice.

The Edit Graph

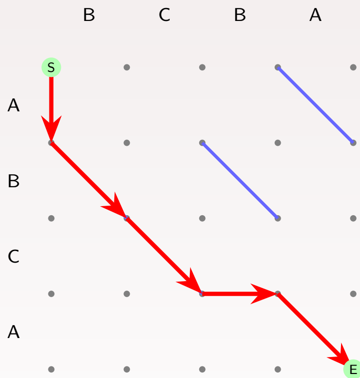
北京大学

Build a grid with $N + 1$ rows (positions in A) and $M + 1$ columns (positions in B).
Edges:

- $\rightarrow$: horizontal step — "insert b_j", cost 1.
- $\downarrow$: vertical step — "delete a_i", cost 1.
- $\searrow$: diagonal step, *allowed only if* $a_i = b_j$ — "match", cost 0.

A path from $(0, 0)$ to (N, M) corresponds to an edit script; its cost equals the number of insertions and deletions.

Goal: find the shortest such path.



Shortest path ($D = 2$)

Match (diagonal, free)

→ insert, ↓ delete

Script: `-A, =B, =C, +B, =A`

Each non-diagonal step costs 1; diagonals (matches) are free. Myers' algorithm finds the path that uses the fewest non-diagonal steps.

Why a Build Tool?

北京大學

As soon as a project has more than one source file, building it “by hand” becomes painful:

- Compile each source file in the right order.
- Recompile only what actually changed.
- Re-link when any object file changes.
- Run tests, generate documentation, package releases. . .

`make` [4] (1976, Stuart Feldman) is the canonical UNIX build tool. It encodes the project as a graph of files (*targets*) and dependencies, and rebuilds the smallest necessary subset.

The Anatomy of a Rule

```
target: prerequisites
<TAB>recipe
```

- **target** — the file (or phony name) to be built.
- **prerequisites** — the files this target depends on.
- **recipe** — the shell commands that build the target.

Crucial pitfall: the recipe must be indented with a literal `TAB`, not spaces. Editors that "helpfully" replace tabs with spaces cause cryptic "missing separator" errors.

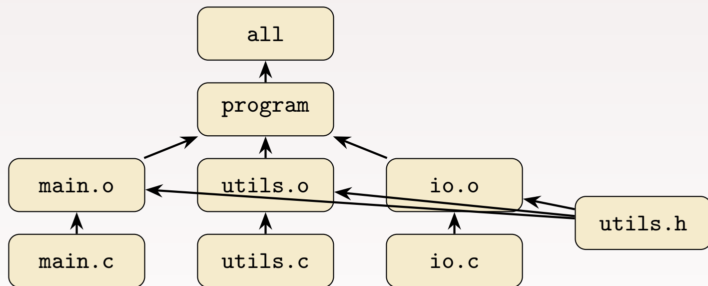
When Does make Rebuild?

- make compares the modification time of the target with the modification times of its prerequisites.
- If any prerequisite is newer than the target (or the target does not exist), the recipe is run.
- Otherwise, make prints `'target' is up to date.`

```
output.txt: input.txt
	cat input.txt > output.txt
```

`output.txt` is rebuilt only when `input.txt` is newer than it.

`make` treats the project as a directed acyclic graph of targets:



Editing `utils.h` forces three `.o` files to be rebuilt and the whole program to be relinked. Editing `io.c` only rebuilds `io.o` and relinks.

A Realistic Makefile


```
CC      = gcc
CFLAGS  = -Wall -O2
OBJS    = main.o utils.o io.o

all: program

program: $(OBJS)
        $(CC) $(CFLAGS) -o $@ $^

main.o: main.c utils.h
utils.o: utils.c utils.h
io.o:   io.c   utils.h

.PHONY: clean
clean:
        rm -f *.o program
```

Two ideas pulling weight here:

- *Variables* — CC, CFLAGS, OBJS.
- *Automatic variables* — $@ (target), $^ (all prerequisites), $< (first prerequisite).

Pattern Rules and Implicit Rules

北京大学

Pattern rules generalise over many similar targets:

```
%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@
```

This single rule replaces a per-file rule for every `.c` file.

`make` also has *built-in implicit rules*: if you have only `hello.c`, you can run

```
$ make hello
cc      hello.c   -o hello
```

with no Makefile at all — `make` infers the recipe from its knowledge of common compilers.

Phony and Special Targets

Some “targets” are not files but command-aliases:

```
.PHONY: all clean test install

test: program
        ./run-tests.sh

install: program
        cp program /usr/local/bin/
```

- Without `.PHONY`, `make test` would do nothing if a file named `test` happened to exist in the directory.
- `.PHONY` tells `make` “always run this recipe”.

Parallel Builds

Because `make` knows the dependency graph, it can build independent targets concurrently:

```
$ make -j8          # build with 8 jobs in parallel
$ make -j$(nproc) # use all CPU cores
```

- Speeds up large builds dramatically.
- Reveals *missing dependencies* — a target that "works" serially may fail in parallel because a prerequisite was not declared.
- This is one of the strongest reasons to be precise about dependencies.

Make Beyond C

`make` is not C-specific. It is used to:

- Compile LaTeX documents (the build script for *these slides*).
- Generate code from grammars, schemas, or protobuf files.
- Run reproducible data-science pipelines.
- Coordinate Docker image builds and deployments.

Modern alternatives (CMake, Bazel, Ninja, Gradle, just, ...) inherit the same core idea: *declare the dependency graph, and the tool figures out what to do*.

One Tool, One Job

Most everyday data wrangling on UNIX uses a handful of small text-processing utilities:

Tool	Job
`cat`	concatenate / dump files
`head`, `tail`	first / last N lines (`tail -f` follows logs)
`cut`	extract columns from delimited text
`tr`	translate or delete characters
`sort`	sort lines (`-n` numeric, `-k` key, `-u` unique)
`uniq`	collapse adjacent duplicates (`-c` count)
`wc`	count lines, words, bytes
`grep`	filter lines matching a regex
`sed`	stream-edit (substitute, delete, insert)
`find`	locate files by name, size, time
`xargs`	turn standard input into command-line arguments

Most common use: regex substitution.

```
$ sed 's/foo/bar/g' input.txt > output.txt
```

- `s/pattern/replacement/flags` — `g` = global (every match on the line), `i` = case-insensitive.
- `-i` edits files *in place* (`-i.bak` keeps a backup).

Other useful operations:

```
$ sed -n '10,20p' file.txt          # print lines 10..20
$ sed '/^#/d' file.txt              # delete comment lines
$ sed 's/[ \t]*$//' file.txt        # strip trailing whitespace
```

find walks a directory tree and applies tests:

```
$ find . -name '*.py' -mtime -7        # Python files modified
    in the last week
$ find . -size +100M                   # files over 100 MB
$ find . -type f -name '*.tmp' -delete # delete temp files
```

xargs converts standard input into command-line arguments — useful when a command does not natively read from stdin:

```
$ find . -name '*.c' | xargs wc -l
$ find . -name '*.c' -print0 | xargs -0 grep -l TODO
```

Use -print0/-0 when filenames may contain spaces or newlines.

A Realistic Pipeline

Find the top three IPs hitting a web log:

```
$ cut -d' ' -f1 access.log | sort | uniq -c | sort -rn | head
    -3
   8421 192.168.1.42
   2103 10.0.0.7
    915 172.16.5.18
```

Reading left to right:

1. `cut` extracts the first whitespace-separated field (the IP).
2. `sort` groups identical IPs together so `uniq` can collapse them.
3. `uniq -c` counts each group.
4. `sort -rn` sorts numerically by count, descending.
5. `head -3` keeps the top three.

What is `awk`?

- `awk` [5] (Aho, Weinberger, Kernighan, 1977) is a tiny programming language for processing *records of fields* — tabular text such as logs, CSVs, and command output.
- Each input line is a *record*, automatically split into *fields* on whitespace (or a chosen separator).
- A program is a list of `pattern` { `action` } pairs: for every record, every matching pattern fires its action.

Syntax

awk 'pattern1 { action1 } pattern2 { action2 } ...' file ...

- Either `pattern` or `action` may be omitted:
  - missing `action` ⇒ { `print` } (print the line);
  - missing `pattern` ⇒ action runs on every line.
- Patterns can be regexes (`/re/`), comparisons (`$3 > 100`), or the special markers `BEGIN` / `END`.

- $0 — the whole record (line).
- $1, $2, . . . — the 1st, 2nd, . . . field.
- NR — number of records read so far.
- NF — number of fields in the current record.
- FS — input field separator (default: whitespace).
- OFS — output field separator (default: space).
- FILENAME — name of the current input file.

```
$ awk -F: '{ print NR, $1, $NF }' /etc/passwd
1 root /bin/bash
2 daemon /usr/sbin/nologin
...
```

-F: sets FS to colon. $NF is the *last* field.

Example: Filter and Project

北京大学

data.txt

```
John    25   5000
Alice   30   6000
Bob     22   4500
```

Print name and salary for everyone older than 23:

```
awk '$2 > 23 { print $1, $3 }' data.txt
```

```
John 5000
Alice 6000
```

Example: BEGIN, END, Aggregation

Total payroll, average age:

```
awk 'BEGIN { print "Summary:" }
     { sum += $3; ages += $2; n++ }
     END   { printf "  total: %d\n  avg age: %.1f\n", sum,
         ages/n }' data.txt
```

```
Summary:
  total: 15500
  avg age: 25.7
```

- BEGIN runs before any input is read.
- END runs after the last record.
- Variables are auto-initialised (numbers to 0, strings to "").

Example: Associative Arrays

Count requests per HTTP method in a web log:

```
awk '{ method = $6; gsub("\"", "", method); count[method]++ }
     END { for (m in count) print count[m], m }' access.log \
    | sort -rn
```

Sample output:

```
8421 GET
 192 POST
  47 HEAD
   3 DELETE
```

Associative arrays make `awk` a surprisingly capable analytics tool.

When to Reach for `awk`

- One-liners over column-oriented text — almost always more concise than a Python script.
- Quick aggregates: sums, averages, frequency counts.
- Light parsing of log files, system reports, command output.

When the task grows — multi-file state, complex data structures, JSON — graduate to a proper scripting language. `awk`'s sweet spot is "one screenful of code or less".

What is `jq`?

- `jq` [6] is a command-line JSON processor: it queries, transforms, and reformats JSON with a small functional expression language.
- Modern web APIs and tools (`kubectl`, `aws`, `gh`, `docker inspect`) emit JSON — `jq` is the standard way to drill into it from the shell.
- Mental model: a JSON value flows through a pipeline of *filters* that produce zero or more output values.

Basic Access

```
$ echo '{"name":"Alice","age":25}' | jq '.name'
"Alice"

$ echo '{"name":"Alice","age":25}' | jq '.age'
25

$ echo '[{"name":"Alice"},{"name":"Bob"}]' | jq '.[0].name'
"Alice"
```

- `.key` — access an object field.
- `.[i]` — index into an array.
- `.[]` — iterate over array (or object) values.
- `.` alone — the identity filter (pretty-prints input).

Composition: the Pipe Operator

jq's `|` works exactly like the shell's `|`:

```
$ echo '[{"name":"Alice","age":25},{"name":"Bob","age":17}]' \
  | jq '.[] | .name'
"Alice"
"Bob"
```

Combine with `select` for filtering:

```
$ echo '[{"name":"Alice","age":25},{"name":"Bob","age":17}]' \
  | jq '.[] | select(.age >= 18) | .name'
"Alice"
```

Building Output: map, Object Construction

北京大学

```
$ echo '[1,2,3,4]' | jq 'map(. * 2)'
[2, 4, 6, 8]
```

Reshape JSON into a different schema:

```
$ echo '[{"first":"Ada","last":"Lovelace"},
         {"first":"Alan","last":"Turing"}]' \
  | jq 'map({full: "\(.first) \(.last)"})'
[
  {"full": "Ada Lovelace"},
  {"full": "Alan Turing"}
]
```

`"\(...)"` is string interpolation.

From JSON to a Spreadsheet

北京大学

Convert a JSON array to CSV:

```
$ cat users.json
[{"name":"Alice","age":25},{"name":"Bob","age":30}]

$ jq -r '.[] | [.name, .age] | @csv' users.json
"Alice",25
"Bob",30
```

- `-r` (raw) drops surrounding quotes around string outputs.
- `@csv`, `@tsv`, `@json`, `@sh`, `@uri` are formatting filters — one of the most useful features for shell glue.

```
$ echo '[1,2,3]' | jq 'length'
3

$ echo '{"a":1,"b":2}' | jq 'keys'
["a","b"]

$ echo '[3,1,2]' | jq 'sort'
[1,2,3]

$ echo '[{"x":2},{"x":1}]' | jq 'sort_by(.x)'
[{"x":1},{"x":2}]

$ echo '[1,2,3,4]' | jq 'add'        # sum
10
```

Combined with shell pipelines, `jq` replaces dozens of ad-hoc Python scripts.

Take-Aways

- **Composition over monoliths.** Small tools that read text and write text combine into solutions for problems their authors never imagined.
- **Patches are the lingua franca of source code.** Understanding unified diff is essential for code review and version control.
- **Declarative builds.** `make` (and its descendants) encode *what* to build, not *how* — the tool figures out the minimal work.
- **Pattern–action** languages (`sed`, `awk`, `jq`) are an underappreciated middle ground between one-off shell commands and full scripts.

Further reading: *UNIX Power Tools* [1], *The Missing Semester* [7].

References I

北京大学

[1] Jerry Peek, Tim O'Reilly, Dale Dougherty, Mike Loukides, Chris Torek, Bruce Barnett, Jonathan Kamens, Gene Spafford, and Simson Garfinkel.
UNIX power tools.
Bantam Books, Inc., 1993.

[2] Free Software Foundation.
Gnu diffutils.
https://www.gnu.org/software/diffutils/, 2025.

[3] Eugene W Myers.
An o (nd) difference algorithm and its variations.
Algorithmica, 1(1):251–266, 1986.

[4] Free Software Foundation.
Gnu make.
https://www.gnu.org/software/make/, 2025.

References II

[5] Free Software Foundation.
Gawk.
https://www.gnu.org/software/gawk/, 2025.

[6] jq Developers.
jq is a lightweight and flexible command-line json processor.
https://jqlang.org/, 2025.

[7] Elaine Mello, Jim Cain, Anthony Zolnik, and Brandi Adams.
The missing semester of your cs education.
https://missing.csail.mit.edu/, 2020.
[Online; accessed 27-Jan-2025].