Metaprogramming

What is Metaprogramming?

- **Metaprogramming** is about the *process* rather than just writing code.
- Involves systems for:
  - Building and testing code.
  - Managing dependencies.
- Essential in large codebases and real-world applications.
- In this context, not about "programs that operate on programs".

Build Systems

- **Build systems** automate the process from inputs to outputs.
- They consist of:
  - **Dependencies**: What is needed.
  - **Targets**: What to produce.
  - **Rules**: How to get from dependencies to targets.
- `make`: A common build system found on UNIX-based systems.

Example `Makefile`

```
paper.pdf: paper.tex plot-data.png
    pdflatex paper.tex

plot-%.png: %.dat plot.py
    ./plot.py -i $*.dat -o $@
```

- Defines how to produce targets from dependencies.
- Patterns match dependencies with targets.

Running `make`

```
$ make
make: *** No rule to make target 'paper.tex', needed by 'paper.pdf'.  Stop.
```

- `make` checks for the needed files and rules.
- If files or rules are missing, `make` will not proceed.

Creating Necessary Files

```
$ touch paper.tex
$ make
make: *** No rule to make target 'plot-data.png', needed by 'paper.pdf'.  Stop.
```

- Even with a rule for `plot-data.png`, missing source files (`data.dat`) halt the process.

Populating Source Files

```
$ cat paper.tex
\documentclass{article}
\usepackage{graphicx}
\begin{document}
\includegraphics[scale=0.65]{plot-data.png}
\end{document}
```

Populating Source Files (cont.)

```
$ cat plot.py
#!/usr/bin/env python
... # Python plotting script
```

```
$ cat data.dat
1 1
2 2
3 3
4 4
5 8
```

- With source files in place, `make` can produce the PDF.

Re-running `make`

```
$ make
make: 'paper.pdf' is up to date.
```

- `make` does nothing if no dependencies have changed.
- Modifying a dependency triggers a rebuild of the related target.

Dependency Management

- Software projects often depend on other projects.
- Dependencies are managed through:
    - **Repositories**: Centralized locations for dependencies.
    - **Tools**: Vary by programming language and system (e.g., `apt`, RubyGems, PyPi).

Versioning

- Projects use *version numbers* to manage compatibility and updates.
- **Semantic Versioning**: major.minor.patch
    - **Major**: Incompatible API changes.
    - **Minor**: Add functionality in a backwards-compatible manner.
    - **Patch**: Backwards-compatible bug fixes.

Lock Files and Vendoring

- **Lock files**: Specify exact versions of dependencies.
- **Vendoring**: Including all dependency code within your project.

Continuous Integration Systems

- **CI**: Automates tasks such as testing, deployment, and documentation updates.
- Common CI providers: Travis CI, Azure Pipelines, GitHub Actions.
- Responds to events like pushes or pull requests.

GitHub Pages Example

- GitHub Pages is a CI action that:
    - Runs Jekyll to build the site.
    - Deploys the site on GitHub domain.
- Simplifies website updates: commit and push, CI does the rest.

A Brief Aside on Testing

- **Test suite**: Collection of all tests.
- **Unit test**: Tests specific features in isolation.
- **Integration test**: Tests how different features work together.
- **Regression test**: Ensures previously fixed bugs don't resurface.
- **Mocking**: Replacing parts of the system to focus on the component being tested.