Software Engineering

Lecture 2
Methodology & Tools

David Baelde

ENS Paris-Saclay & MPRI

September 28, 2018

Introduction

Today we will (mostly) rediscover common developer tools...

- Compilers and their extensions, build automation software, installers, dependency managers, versioning system, bug tracker, documentation generator, etc.

Introduction

Today we will (mostly) rediscover common developer tools. . .

- Compilers and their extensions, build automation software, installers, dependency managers, versioning system, bug tracker, documentation generator, etc.

. . . and articulate this with our mottos, in particular:

> *The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. – E.W. Dijkstra*

> *Hubris is the third great virtue of a programmer. — L. Wall*

Introduction

Today we will (mostly) rediscover common developer tools. . .

- Compilers and their extensions, build automation software, installers, dependency managers, versioning system, bug tracker, documentation generator, etc.

. . . and articulate this with our mottos, in particular:

> *The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. – E.W. Dijkstra*

> *Hubris is the third great virtue of a programmer. — L. Wall*

> *Laziness is the first. — L. Wall*

Programming Language

The first line of defense

Choose a disciplined language

- Variable declarations: avoid typos
- Static typing: guarantee simple invariants
  more types $\rightsquigarrow$ more expressible invariants
  - Use enumerations rather than magic numbers
  - More in Prog. 2 (L3)

The first line of defense

Choose a disciplined language

- Variable declarations: avoid typos
- Static typing: guarantee simple invariants
  more types $\rightsquigarrow$ more expressible invariants
  - Use enumerations rather than magic numbers
  - More in Prog. 2 (L3)

Exploit your compiler as much as possible

- Even with a strong and statically typed language,
  the compiler is not necessarily very constraining by default.
- OCaml, C/C++, Scala, etc.: activate options to obtain
  more warnings, and treat them as errors.
- Demo in Scala

Contracts and Assertions

Code contracts

A metaphore for Floyd-Hoare logic:
pre-conditions, post-conditions, invariants, etc.
A design methodology: design by contract

Support

- Native langage support: Eiffel, SpeC#
- Extension (comments): JML

Use

- Proof of programs
- Documentation generation
- Unit test generation
- Runtime verification

Assertions

It may be hard to prove the spec,
but it can often easily be executed.

- Detect anomalies earlier.
- A form of “active” comment.

Assertions

It may be hard to prove the spec,
but it can often easily be executed.

- Detect anomalies earlier.
- A form of "active" comment.

The `assert` function(s)

Take a boolean and raise an error if it's false.

```
let add ?(needs_check=true) x rules kb =
  assert (needs_check || not (mem_equiv x kb)) ;
  if not (needs_check && mem_equiv x kb) then
    add (fresh_statement x) kb
```

Often part of the core language, with an erasing facility:
`ocamlc -noassert ...`, `g++ -DNDEBUG ...`, etc.

Using assertions

No-no

- If assert raises an exception, it should not be caught!
  (At least not permanently.)

```
let main () =
  try ... with _ -> eprintf "Oops!\n" ; main ()
```

- Erasing assertions should not change the behavior of the code!

(Could we systematically detect such problems?)

Using assertions

No-no

- If assert raises an exception, it should not be caught!
  (At least not permanently.)

```
let main () =
  try ... with _ -> eprintf "Oops!\n" ; main ()
```

- Erasing assertions should not change the behavior of the code!

(Could we systematically detect such problems?)

Grey zone

- When is an assertion too costly?

Using assertions

No-no

- If assert raises an exception, it should not be caught!
  (At least not permanently.)

```
let main () =
  try ... with _ -> eprintf "Oops!\n" ; main ()
```

- Erasing assertions should not change the behavior of the code!

(Could we systematically detect such problems?)

Grey zone

- When is an assertion too costly?
  Beware premature optimization.
  Consider multiple assertion levels.

Using assertions

No-no

- If assert raises an exception, it should not be caught!
  (At least not permanently.)

```
let main () =
  try ... with _ -> eprintf "Oops!\n" ; main ()
```

- Erasing assertions should not change the behavior of the code!

(Could we systematically detect such problems?)

Grey zone

- When is an assertion too costly?
  Beware premature optimization.
  Consider multiple assertion levels.
- Should we release software with assertions enabled?

Using assertions

No-no

- If assert raises an exception, it should not be caught!
  (At least not permanently.)

```
let main () =
  try ... with _ -> eprintf "Oops!\n" ; main ()
```

- Erasing assertions should not change the behavior of the code!
  (Could we systematically detect such problems?)

Grey zone

- When is an assertion too costly?
  Beware premature optimization.
  Consider multiple assertion levels.
- Should we release software with assertions enabled?
  Rather not, so as to benefit from precise errors.
  Consider changing them into BIG warnings.

Automation

Build automation

We keep changing and rebuilding software $\Rightarrow$ automate it !

Requirements

- Automatically build software from latest source code.
- Avoid useless recompilations.
- Get the dependencies right, handle subdirectories, multiple languages and targets, code generators, etc.
- Perhaps automatically fetch dependencies, etc.

Build automation

We keep changing and rebuilding software $\Rightarrow$ automate it !

Requirements

- Automatically build software from latest source code.
- Avoid useless recompilations.
- Get the dependencies right, handle subdirectories, multiple languages and targets, code generators, etc.
- Perhaps automatically fetch dependencies, etc.
- All developers should understand its use, and actually use it.

Focus on `make`, but the key concepts are the same for other tools.

Usual make targets

GNU Coding Standards, *The Release Process, Standard Targets*, R. M. Stallman et al., 2016.

`make all`
Compile the entire program. Should be the default.
GNU says "By default, should compile `-g`." Why?

`make test` or `make check`
Test the software, or parts of it.
Meant to be used before installation.

`make doc`
Generate documentation from source code,
relevant only for developpers.

Usual make targets

`make install`
Install applications, libraries, documentation.
Create directories if needed, set the right permissions...
better use utilities such as `install`.

`make clean`
Delete intermediary files built by `make`.

`make distclean`
Cleans all automatically generated files.

`make dist`
Create a tarball for distribution to end users.

Adaptability

Use variables for programs and options that could change.

Compilation

```
CC = gcc
CFLAGS = -g
.c.o:
    $(CC) $(CFLAGS) -c $<
```

Adaptability

Use variables for programs and options that could change.

Compilation

```
CC = gcc
CFLAGS = -g
.c.o:
	$(CC) $(CFLAGS) -c $<
```

Installation, ready for alternative paths and sandboxing

```
prefix = /usr/local
bindir = $(exec_prefix)/bin
libdir = $(prefix)/lib
install: all
	$(INSTALL_PROGRAM) foo $(DESTDIR)$(bindir)/foo
	$(INSTALL_DATA) libfoo.a $(DESTDIR)$(libdir)/libfoo.a
```

These examples use standard variables names: why is it useful?

Configure and beyond

Configuration options

- Compiler, compiler options
- Libraries or library versions
- Enable/disable specific features

`./configure`

- Discover reasonable default values for configuration options and detect missing dependencies,
  using tools such as `pkg-config`, `ocamlfind`, etc.
- Generate (parts) of Makefile or code, perhaps using `automake`.

Writing a portable `configure` script can be quite complex;
the script itself may be generated instead using `autoconf`.

More user-friendly systems?

- `cmake` for C
- `ant` for Java
- `sbt` for Scala
- `xbuild` for .NET
- `ocamlbuild`, `ocp-build` for OCaml
- ...

Evaluate your needs before choosing! Some tools are easy for simple projects, but make more complex cases very hard or impossible.

In practice

Example

- `ocaml-ogg` for autotools
- 2018's `rogue` for parallelization

In practice

Example

- `ocaml-ogg` for autotools
- 2018's `rogue` for parallelization

Exercise

Users may build software from a release or from a code repository. In either case a release or revision number can identify the software version; such information is useful when reporting problems.

- How would you make the information available in the application, e.g. as output of `--version` or in crash reports.

Exercise

When library XYZ is available, you want to provide an entry in your application's menubar for performing something thanks to XYZ.

- How/when would you do this in C ? in OCaml ?

Beyond the build

Code, *compile*, *test*, *commit*: how to enforce this humble workflow?

Beyond the build

Code, *compile*, *test*, *commit*: how to enforce this humble workflow?

Hooks

`.git/hooks/pre-commit` is executed to verify commits:

- extend sample hook to run a simple `make test`.

Limitations:

- the hook's execution should be fast;
- its success may be dependent upon untracked files, a particular configuration, etc.

Beyond the build

Code, *compile*, *test*, *commit*: how to enforce this humble workflow?

Hooks

`.git/hooks/pre-commit` is executed to verify commits:

- extend sample hook to run a simple `make test`.

Limitations:

- the hook's execution should be fast;
- its success may be dependent upon untracked files, a particular configuration, etc.

Continuous integration

For every commit pushed on the main repository, build and test a fresh clone on one/several blank virtual machines.
Public github repositories get free Travis CI servers.

Conclusion

Recap

We've seen some of the most common software development tools.

Your project will rock if you use them well.

Conclusion

Recap

We've seen some of the most common software development tools.

Your project will rock if you use them well.

Keep searching for more techniques to improve your workflow.

Conclusion

Recap

We've seen some of the most common software development tools.

Your project will rock if you use them well.

Keep searching for more techniques to improve your workflow.

Next:

- Choice of projects
- Git tutorial