Software Engineering

Lecture 2

Methodology & Tools, Testing

David Baelde

ENS Paris-Saclay & MPRI

September 22, 2017

Outline

Principles

- Previous lecture: Rigor, Change, Modularity, Abstraction

Methodology

Tools

Outline

Principles

- Previous lecture: Rigor, Change, Modularity, Abstraction

Methodology

- Reviews: code reviews, pair programming
- Documentation: in / out of the code; present / past / future
- Tests: unit / integration; white / black box; regression
- Modeling, retrospective, refactoring, etc.

Tools

Outline

Principles

- Previous lecture: Rigor, Change, Modularity, Abstraction

Methodology

- Reviews: code reviews, pair programming
- Documentation: in / out of the code; present / past / future
- Tests: unit / integration; white / black box; regression
- Modeling, retrospective, refactoring, etc.

Tools

- Compiler, build automation, versioning system, bug tracker, documentation generator, installers, dependency managers...

Counter-examples

Real stories

- Student codes, compiles by hand, runs, repeats.

Counter-examples

Real stories

- Student codes, compiles by hand, runs, repeats.
  Eventually a compilation is forgotten, things don't make sense.

Counter-examples

Real stories

- Student codes, compiles by hand, runs, repeats.
  Eventually a compilation is forgotten, things don't make sense.
- Student does not manage to use project's build system,
  builds his file(s) in isolation, commits them.

Counter-examples

Real stories

- Student codes, compiles by hand, runs, repeats.
  Eventually a compilation is forgotten, things don't make sense.
- Student does not manage to use project's build system, builds his file(s) in isolation, commits them.
  Useless code, present and future problems go unnoticed.

Counter-examples

Real stories

- Student codes, compiles by hand, runs, repeats.
  Eventually a compilation is forgotten, things don't make sense.
- Student does not manage to use project's build system,
  builds his file(s) in isolation, commits them.
  Useless code, present and future problems go unnoticed.
- Developer fixes a bug, breaks something else.
- Developer fixes a bug, another developer re-introduces it.

Slogans

- Don't repeat yourself. Don't trust yourself.
- Systematically look for bugs. Automate as much as possible.

Build System

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

Build automation

We keep changing and rebuilding software ⇒ 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 ⇒ 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`."

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

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

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
```

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`.

Other tools

The GNU autotools are frightening and criticized.

Alternatives

- `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

- Bedwyr

In practice

Example

- Bedwyr

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.

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.

Test

Tests

What?

- Explicit spec and/or “good behavior”.

Why?

- Detect problems earlier.
- Facilitate identification of root cause.
- Reproduce.

Tests

What?

- Explicit spec and/or “good behavior”.

Why?

- Detect problems earlier.
- Facilitate identification of root cause.
- Reproduce.
- Prevent problems from reappearing.

Tests

What?

- Explicit spec and/or "good behavior".

Why?

- Detect problems earlier.
- Facilitate identification of root cause.
- Reproduce.
- Prevent problems from reappearing.

How?

- Unit testing on . . . basic units.
- Integration testing, complete system testing.

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
the testing suite must “explore” as many behaviors as possible.

Citeria: lines,

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
 the testing suite must “explore” as many behaviors as possible.

Citeria: lines, control flow, conditions,

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
the testing suite must “explore” as many behaviors as possible.

Citeria: lines, control flow, conditions, values, states, etc.
Tests are not proofs!

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
the testing suite must “explore” as many behaviors as possible.

Citeria: lines, control flow, conditions, values, states, etc.
Tests are not proofs!

Choosing test values, based on code and spec:
partitions, equivalence classes, boundaries...
Example: `triangle.ml`

White box

Goal: relevant tests based on the structure of the code.

Idea of coverage:
the testing suite must “explore” as many behaviors as possible.

Citeria: lines, control flow, conditions, values, states, etc.
Tests are not proofs!

Choosing test values, based on code and spec:
partitions, equivalence classes, boundaries. . .
Example: `triangle.ml`

By hand, or using the machine. . .

Pex 1 (C#)

Generate "interesting" test values, by symbolic execution and constraint solving. Demo: `http://www.pexforfun.com`

```
public class Point  {
  public readonly int X, Y;
  public Point(int x, int y) { X = x; Y = y; }
}
public class Program  {
  public static void Puzzle(Point p)
  {
    if (p.X * p.Y == 42)
      throw new Exception("Bug!");
  }
}
```

Pex 1 (C#)

Generate “interesting” test values, by symbolic execution and constraint solving. Demo: `http://www.pexforfun.com`

```
public class Point  {
  public readonly int X, Y;
  public Point(int x, int y) { X = x; Y = y; }
}
public class Program  {
  public static void Puzzle(Point p)
  {
    if (p.X * p.Y == 42)
      throw new Exception("Bug!");
  }
}
```

Propose 3 inputs: `null`, `(0,0)` and `(3,14)`.

Pex 2 (C# + contracts)

```
public class Program  {
  public static string Puzzle(string value) {
    Contract.Requires(value != null);
    Contract.Ensures(Contract.Result<string>() != null);
    Contract.Ensures(
      char.IsUpper(Contract.Result<string>()[0]));
    return char.ToLower(value[0]) + value.Substring(1);
  }
}
```

Find inputs that trigger bugs. . .

Pex 2 (C# + contracts) fixed

```
public class Program  {
  public static string Puzzle(string value) {
    Contract.Requires(value != null);
    Contract.Requires(value=="" ||
                      char.IsLower(value[0]));
    Contract.Ensures(Contract.Result<string>() != null);
    Contract.Ensures(
      Contract.Result<string>()=="" ||
      char.IsUpper(Contract.Result<string>()[0]));
    if (value=="") return value;
    return char.ToUpper(value[0]) + value.Substring(1);
  }
}
```

Pex 3 (C# + contracts)

```
using System;

public class Program {
  static int Fib(int x) {
    return x == 0 ? 0 : x == 1 ? 1 :
           Fib(x - 1) + Fib(x - 2);
  }
  public static void Puzzle(int x, int y)
  {
    if (Fib(x + 27277) + Fib(y - 27277) == 42)
      Console.WriteLine("puzzle solved");
  }
}
```

Black box

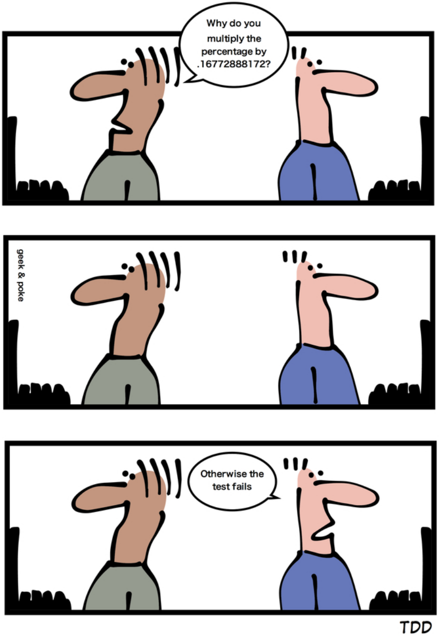
What if we cannot / don't want to rely on the code?

Black box: TDD

Test driven development:
write tests first, then code
that passes them.

Black box: TDD

Test driven development: write tests first, then code that passes them.



Black box: test & spec

Tests cannot replace specs, but allow to exploit it more.

Generate tests from specs:
  spec coverage, *e.g.*, cause/consequence, clauses

Black box: randomness and stress

Randomized tests

- `Quickcheck`, `Scalacheck` (demo):
  test predicates on random input values

Black box: randomness and stress

Randomized tests

- `Quickcheck`, `Scalacheck` (demo):
  test predicates on random input values
- `Csmith`: compare C compilers on random code samples
  ⇝ no need for a spec (phew!)

Black box: randomness and stress

Randomized tests

- `Quickcheck`, `Scalacheck` (demo):
  test predicates on random input values
- `Csmith`: compare C compilers on random code samples
  $\rightsquigarrow$ no need for a spec (phew!)

Stress

- Flood a server with requests
- Execution with constrained resources (memory, disk)
- Create latency (network)

Black box: randomness and stress

Randomized tests

- `Quickcheck`, `Scalacheck` (demo):
  test predicates on random input values
- `Csmith`: compare C compilers on random code samples
  ⇝ no need for a spec (phew!)

Stress

- Flood a server with requests
- Execution with constrained resources (memory, disk)
- Create latency (network)

Fuzz testing

- Mainly for file formats and protocols
- Test on (partly) randomly generated/modified data
- `zzuf` (demo), `LibFuzzer`, `afl-fuzz`, . . .

Objection 1

Writing tests $=$ wasting time ?

Objection 1

Writing tests = wasting time ?

When coding, you're already writing tests:
maybe in an interpreter,
often in temporary `printf` checkes, visual verification,
etc.

The goal is to preserve such tests, so as to fully exploit them.

Objection 1

Writing tests = wasting time ?

When coding, you're already writing tests:
maybe in an interpreter,
often in temporary `printf` checkes, visual verification,
etc.

The goal is to preserve such tests, so as to fully exploit them.

Regression test

Good practice integrating testing and debugging:
before debugging, turn minimized bug into a test;
the test will validate the fix and prevent future regressions.

Objection 2

> “That’s easy for a sorting function,
> but another story for a server. . . ”

Often, hard to test = poorly designed !

Examples

- Interaction with the filesystem, a database, etc.: sandboxing
- Graphical interface: possibility to script or capture (`xnee`)
  beware: testing the interface or the underlying logic?
- Non-functional aspects (time, space): profiling

Testing environment

Librairies to write tests more easily:
`xUnit`, `Scalacheck`, `Scalatest`, etc.

Testing environment

Librairies to write tests more easily:
`xUnit`, `Scalacheck`, `Scalatest`, etc.

Infrastructure for (selectively) executing tests
and producing reports, *e.g.*, `SBT+Scalacheck/test`

Testing environment

Librairies to write tests more easily:
`xUnit`, `Scalacheck`, `Scalatest`, etc.

Infrastructure for (selectively) executing tests
and producing reports, *e.g.*, `SBT+Scalacheck/test`

Systematic exploitation:

- Hooks on commits
- Continuous integration (Jenkins, Travis CI, etc.)

Summary

Tools

Understand their purpose, get the most out of them.

- Compiler, build system, doc generator
- Unit testing frameworks, random testing, etc.
- Version control, continuous integration
- Bug / issue tracker, discussions about changesets
- Several other tools depending on project

Summary

Tools

Understand their purpose, get the most out of them.

- Compiler, build system, doc generator
- Unit testing frameworks, random testing, etc.
- Version control, continuous integration
- Bug / issue tracker, discussions about changesets
- Several other tools depending on project

Principles

- Rigor, Change, Modularity, Abstraction

Summary

Tools

Understand their purpose, get the most out of them.

- Compiler, build system, doc generator
- Unit testing frameworks, random testing, etc.
- Version control, continuous integration
- Bug / issue tracker, discussions about changesets
- Several other tools depending on project

Principles

- Rigor, Change, Modularity, Abstraction
- Laziness, Impatience, Hubris (Larry Wall)
- Humility (Edsger W. Dijkstra)

Summary

Tools

Understand their purpose, get the most out of them.

- Compiler, build system, doc generator
- Unit testing frameworks, random testing, etc.
- Version control, continuous integration
- Bug / issue tracker, discussions about changesets
- Several other tools depending on project

Principles

- Rigor, Change, Modularity, Abstraction
- Laziness, Impatience, Hubris (Larry Wall)
- Humility (Edsger W. Dijkstra)
- "Talk is cheap. Show me the code." (Linus Torvalds)