Software Engineering

Lecture 3: defensive programming

David Baelde
baelde@lsv.ens-cachan.fr

MPRI

September 30, 2015

Problem

What is good code / coding?
(From a software engineering perspective.)

Citeria

Problem

What is good code / coding?
(From a software engineering perspective.)

Citeria

- No crash, memory leak, undefined behavior, etc.
- Conform to the functional specification

Problem

What is good code / coding?

(From a software engineering perspective.)

Citeria

- No crash, memory leak, undefined behavior, etc.
- Conform to the functional specification
- Performance: time, space, reactivity
- Ease of bug fixing and code evolution

Problem

What is good code / coding?
(From a software engineering perspective.)

Citeria

- No crash, memory leak, undefined behavior, etc.
- Conform to the functional specification
- Performance: time, space, reactivity
- Ease of bug fixing and code evolution

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. — Brian Kernighan

Principles

Rigor

- Think

Principles

Rigor

- Think

Principles

Rigor

- Think
- Review and peer-review
- Systematize

Principles

Rigor

- Think
- Review and peer-review
- Systematize

Paranoïa

There ARE bugs in your code.

Problem: (re)produce, minimize, understand and fix them.
Might as well be early and in simple circumstances.

Principles

Rigor

- Think
- Review and peer-review
- Systematize

Paranoïa

There ARE bugs in your code.

Problem: (re)produce, minimize, understand and fix them.
Might as well be early and in simple circumstances.

A few concrete ideas in this lecture...

Static analysis

Formal methods

Theory

Model-checking, deductive verification, abstract interpretation, certified code generation, etc.

Practice

- Why, Frama-C, etc.
- At Microsoft
  - MSR Tools: SLAM, Boogie, Z3...
  - Every code change checked by verification tools.
    Not necessarily complete nor correct!
    - correct: finds all bugs (and false alerts)
    - complete: finds only real bugs (but misses some)
- Sparse: check annotations in the Linux kernel

What if we cannot / do not know how to use those tools?

Avoid the worst

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)

Avoid the worst

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.

In addition to the compiler (but still static)

Check style and good practices

Tools such as `lint`, `cpplint`, `scalastyle`, etc.

- Long lines, spacing, naming conventions
- Enforce type annotations on public methods
- Impose block delimiters for `if`
- Avoid `return` and `var`

Simple analyzers

Look for memory leaks , unchecked errors, trivial tests, library misuse, etc. in a more or less syntactic way.

- `cppcheck`, clang static analyzer, PVS studio, etc.
- Scala `linter` (demo)

Contracts

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

Focus on debugging, not conformance to requirements.
Remember defensive (offensive?) programming.

Goals

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

Tests

Focus on debugging, not conformance to requirements.
Remember defensive (offensive?) programming.

Goals

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

Tests

Focus on debugging, not conformance to requirements.
Remember defensive (offensive?) programming.

Goals

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

Testing granularity

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

Which properties? Explicit spec and/or "good behavior".

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.

> *Program testing can be used to show the presence of bugs, but never to show their absence! — E.W.Dijkstra*

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.

> *Program testing can be used to show the presence of bugs, but never to show their absence! — E.W.Dijkstra*

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.

> *Program testing can be used to show the presence of bugs, but never to show their absence! — E.W.Dijkstra*

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

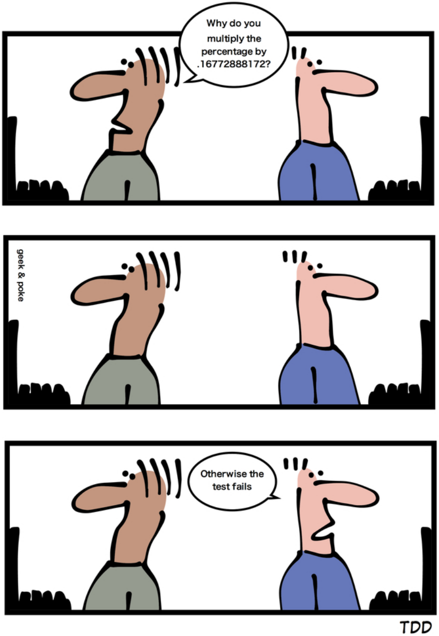
What if we cannot / don't want to rely on the code?

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
  $\rightsquigarrow$ 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
  $\rightsquigarrow$ 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`, ...

In practice

Objection

Writing tests = wasting time?

Objection

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

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.

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.)

Objection

"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

Conclusion

Remember

- Be a humble, paranoid programmer
- Think debugging even from the design phase
- Provoke bugs

Tools are your friends (tutorials coming):

- Your programming language
- A strict compiler, linter, etc.
- Unit testing infrastructure, continuous integration
- Logging system
- Git(Hub) for code reviews and debugging