Teaching Clean Code

Linus W. Dietz, Technical University of Munich

Johannes Manner, University of Bamberg

Dr. Simon Harrer, University of Bamberg

Dr. Jörg Lenhard, Karlstad University

Introduction

Observation

University graduates often lack clean coding proficiency

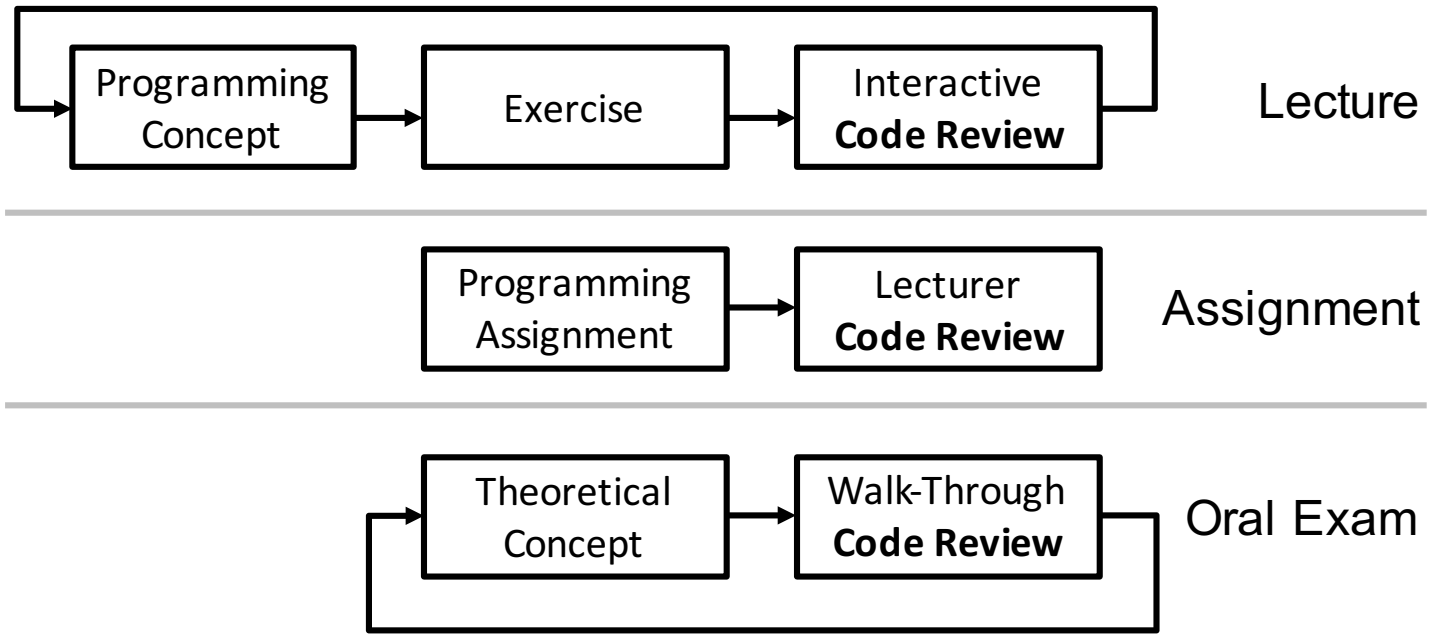
Why?

- Focus on functional correctness
- (Too) many students per course
- Insufficient feedback
- No/short software maintenance period → no awareness

Research questions

1) How to teach clean code at university?
2) How to tech clean code at scale?

Didactic Concept



Developed and evaluated over seven years at the University of Bamberg

- Advanced Java programming courses
- Popular among students, won teaching award

Assignment Corrections are the Bottleneck!

- Time consuming
- Same mistakes over and over
- Corrector must be skilled
- Good feedback is hard to achieve
- Easy to miss small bugs

Knowledge Base

- Most common “Common Issues“
- 70 before / after code examples & explanations
- Actionable names
- For Java learners in their 2nd year and later

Avoid Negations

```
class Laboratory {

    Microscope microscope;

    Result analyze(Sample sample) {
        if (microscope.isInorganic(sample)) {
            return Result.INORGANIC;
        } else {
            return analyzeOrganic(sample);
        }
    }

    private Result analyzeOrganic(Sample sample) {
        if (!microscope.isHumanoid(sample)) {
            return Result.ALIEN;
        } else {
            return Result.HUMANOID;
        }
    }
}
```

Has anyone ever told you to "think positively"? Turns out, positive expressions are better in your code than negative ones because they're often easier to grasp and they take up slightly less space.

The problem code shown here is another variant of the Laboratory component. It provides two methods that take a Sample and return a Result. There's nothing obviously wrong with the code. It'll fulfill its purpose, but it's more complex than necessary.

Consider the conditions of the if statements: Both express a negating condition. The first one tests if a sample isInorganic(). The second one even uses the Boolean operator for negation: the exclamation mark '!'.

In most cases, you'll find it easier to understand positive expressions when you read code. A negated expression adds one extra layer of indirection. Instead of a simple "X does apply," you have to grasp an additional token: "X does *not* apply."

This extra token is often unnecessary. Although it seems like a minor change, every tiny bit of simplification helps for more complex expressions (those that you'll find in real code).

Remember: Everybody likes no negations.

Now, take a look at the simplified version:

```
class Laboratory {

    Microscope microscope;

    Result analyze(Sample sample) {
        if (microscope.isOrganic(sample)) {
            return analyzeOrganic(sample);
        } else {
            return Result.INORGANIC;
        }
    }

    private Result analyzeOrganic(Sample sample) {
        if (microscope.isHumanoid(sample)) {
            return Result.HUMANOID;
        } else {
            return Result.ALIEN;
        }
    }
}
```

We changed the code only slightly, but that already has an effect. Instead of isInorganic(), we're calling isOrganic(), which has a positive wording instead of a negative one. That is, we have to switch the bodies of the if and else blocks.

When it comes to the second part, we're calling the same method, isHumanoid(), but we eliminated the negation. This is a real simplification of the code. Again, it means that we have to switch the bodies of the if and else blocks.

All in all, these changes are quite simple and you might wonder: why should I bother? You should bother because it improves the understandability of the code and comes at virtually no cost. You don't usually have to add further lines of code. All you have to do is re-sort existing parts of the code and you'll see an improvement. It's an opportunity that's too good to let go!

Sometimes, however, you need the appropriate methods, like the isOrganic() instead of isInorganic() here. If the code you're calling comes from a third-party library, you might not have the option to invoke different methods. But when you control it, don't shy away from adding that method to the appropriate class—the few lines of code are worth their characters, because they make your code clearer in other places. In the long run you'll end up with less code, since such methods reduce code duplication and can be reused in other parts of the program. From our experience, it's best to get rid of the negative method completely—no need to maintain two similar methods.

QualityReview: Automated Didactic Code Review

- Static code analysis meta tool
- Based on our teaching experience
- Pre-configured with 107 rules
- Adjustable for specific needs
- Built with Gradle
- Available under MIT license on Github
  https://github.com/LinusDietz/QualityReview

Call to Actions

- Re-think your teaching concepts: Do students need to maintain code?

- Encourage the students to establish workflows with code reviews

- Improve feedback time and quality
  1) Knowledge base: ***Java by Comparison***
  2) Static code analysis: ***QualityReview***

- **Lets collaborate!**
  - How can *Java by Comparison* be of value in your teaching?
  - Experiments and studies with *QualityReview*
  - Integrate *QualityReview* in automated code assessment frameworks