Clean Code

All of the quotes in the topics below are by Robert C. Martin.

Object Anti-Symmetry

- Data-Structures:
  - expose their data
  - have no meaningful functions to operate on the data
- Objects
  - hide their data behind abstractions
  - expose functions to operate on the data

> *Procedural code (code using data structures) makes it easy to add new functions without*
> *changing the existing data structures. OO code, on the other hand, makes it easy to add*
> *new classes without changing existing functions.*

therefore

> *Procedural code makes it hard to add new data structures because all the functions must*
> *change. OO code makes it hard to add new functions because all the classes must change.*

Object Anti-Symmetry ist ein Konzept in der objektorientierten Programmierung (OOP), das sich mit der Balance zwischen der Wiederverwendbarkeit von Code und der Anpassbarkeit an neue Anforderungen beschäftigt.

Diese Anti-Symmetrie besagt, dass es einen trade-off gibt zwischen Wiederverwendbarkeit und Anpassbarkeit: Je mehr Code wiederverwendbar ist, desto weniger anpassbar ist er an neue Anforderungen, und umgekehrt.

Um diesen trade-off zu minimieren, sollte OOP-Code so gestaltet sein, dass er so wiederverwendbar wie möglich ist, ohne dass die Anpassbarkeit an neue Anforderungen beeinträchtigt wird. Dies kann erreicht werden, indem man eine saubere und gut strukturierte Architektur für das System schafft und durch die Verwendung von Techniken wie Polymorphie, Abstraktion und Encapsulation.

Zusammenfassend ist Object Anti-Symmetry ein Konzept, das sich mit dem trade-off zwischen Wiederverwendbarkeit und Anpassbarkeit befasst und dabei hilft, eine saubere und gut strukturierte Architektur für softwarebasierte Systeme zu schaffen.

Train Wrecks

```
final String outputDir = ctx.getOptions().getScratchDir().getAbsolutePath();
```

 Analogy ⇒ code looks like a bunch of coupled train carts

Better representation:

```
Options opts = cxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

The calling context knows a lot about the structure of the `ctx`-object. If it's a violation of LoD depends on the fact if `ctx`, `Options` and `ScratchDir` are data structures or objects. If they are objects the LoD is clearly violated since the calling function knows about the inner workings of `ctx`.

The example stated above is a bit confusing due to the accessor functions. If they are data structures and would expose their data directly as stated below, the violation of LoD would be out of question.

```
final String outputDir = ctx.options.scratchDir.absolutePath;
```

Boy Scout Rule

A codebase must be kept clean over time. In order to achieve that one can apply a simple rule found with the Boy Scouts of America:

> Leave the campground cleaner than you found it

In other terms this means:

> One should check-in cleaner code than one has checked-out beforehand

However the clean-up mustn't be something massive, a simple refactoring of a variable name or the resolving of a deeply nested `if` can already make the difference.

Meaningful Names

Choosing good names takes time but saves more than it takes

The name of a variable, function or class should reveal:

- why it exists
- what it does
- how it is used

If a variable requires a comment, its name isn't sufficient.

Be vary when including words with contextual meaning e.g. if `accountList` is effectively an `Array` instead of a `List` one could be mislead to believing it is a `List`-Instance ⇒ the simpler name `accounts` would be better suited here.

Declarative names are generally something positive however maintain a healthy amount of common sense to identify when it becomes unreadable.