Do you check input parameters of methods for validity? Well, maybe you shouldn't, and instead you should use validating decorators.

Do you check the input parameters of your methods for validity? I don't. I used to, but not anymore. I just let my methods crash with a null pointer and other exceptions when parameters are not valid. this may sound illogical, but only in the beginning. i'm suggesting you use **validating decorators** instead.

Let's take a look at this rather typical java example:

```
class report {
  void export(file file) {
    if (file == null) {
      throw new illegalargumentexception(
                "file is null; can't export.");
    }
    if (file.exists()) {
      throw new illegalargumentexception("file already exists." );
    }
    // export the report to the file
  }
}
```

If we remove these validations, the code will be much shorter, but it will crash with rather confusing messages if null is provided by the client. moreover, if the file already exists, our report will silently overwrite it. pretty dangerous, right?
yes, we must protect ourselves, and we must be defensive.
but not this way, not by bloating the class with validations that have nothing to do with its core functionality. instead, we should use decorators to do the validation. here is how.

First, there must be an interface report:

```
interface report {
  void export(file file);
}
```

Then, a class that implements the core functionality:

```
class defaultreport implements report {
  @override void export(file file) {
    // export the report to the file
  }
}
```

And, finally, a number of decorators that will protect us:

```
class nowriteoverreport implements report {
  private final report origin;
  nowriteoverreport(report rep) {
    this.origin = rep;
  }
  @override void export(file file) {
    if (file.exists()) {
      throw new illegalargumentexception("file already exists.");
    }
    this.origin.export(file);
```

```
  }
}
```

Now, the client has the flexibility of composing a complex object from decorators that perform their specific tasks. the core object will do the reporting, while the decorators will validate parameters:

```
report report = new nonullreport(new nowriteoverreport(
                        new defaultreport()));
report.export(file);
```

What do we achieve with this approach? first and foremost: smaller objects. and smaller objects always mean higher maintainability. our defaultreport class will always remain small, no matter how many validations we may invent in the future. the more things we need to validate, the more validating decorators we will create. all of them will be small and cohesive. and we'll be able to put them together in different variations.

Besides that, this approach makes our code much more reusable, as classes perform very few operations and don't defend themselves by default. while being defensive is an important feature, we'll use validating decorators. but this will not always be the case. sometimes validation is just too expensive in terms of time and memory, and we may want to work directly with objects that don't defend themselves.

I also decided not to use the java validation Api anymore for the same reason. its annotations make classes much more verbose and less cohesive. I’m using validating decorators instead.