CHECKLIST

Class Design Checklist

BY GIORGIO SIRONI

✓ NAMING

- ○ Does the class (or interface) name describe what an instance of this class (or interface) does?
  - o *Usually a name or an adjective plus a noun are good for a class, while an adjective is more appropriate for an interface. Some interfaces have names composed by one or more nouns.*
- ○ Does the name contain unnecessary implementation details?
  - o *Interface and abstract classes should not contain any reference to a particular implementation, but you should analyze this issue in context. For instance, XmlParser is not correct if at least a possible parser implementation does not work with Xml, while for a family of Xml parsers that differ in performance XmlParser is appropriate. In the same vein, class names should not contain private implementation details of the class that may change – only the class-special trait in respect to the other implementations (e.g. XmlParser, HtmlParser, YamlParser.)*
- ○ Is the fully-qualified name of the class or interface correct? (Or: is this artifact placed in the right package or namespace?)
  - o *You can make a guess based on the number of dependencies that this new artifact introduces.*
  - o *Naming conventions and best practices are also valid for method names, parameters, local variables, inline comments.*
- ○ Is the name consistent with the rest of the object model? Is it part of the Ubiquitous Language?
  - o *The more public the named entity is (in ascending order: private, protected, [package where applicable,] public, published), the more important it is to assign a valid name immediately.*
- ○ Should the name be influenced by a standard Ubiquitous Language?
  - o *This is usually true for implementations of design patterns, where leveraging the role names communicates a great deal about the code structure. Other examples of a standard Ubiquitous Language are framework and programming language conventions.*

✓ STRUCTURE

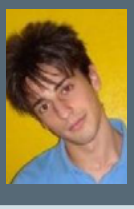
- ○ How many levels of indentation are there in your code?
  - o *Supposing that the first level is dedicated to methods, two other levels are acceptable, with one (thus two in total) being the norm. Extract Method will help break up the complexity into different, orthogonal methods.*
- ○ Have you inserted switch constructs, especially similar ones? This structure is usually a smell, which can be refactored with a State or Strategy pattern.
  - o *If-elseif chains or even if-else constructs, when repeated, are equivalent to switch, with the latter being its two-fold substitute.*

○	Are there new operators mixed with business logic? This is a no-brainer.
○	Are there any controversial constructs in the code, such as the static keyword, or goto?
○	If the code artifact is a subclass, does it extend the right parent class? If it is an implementation, does it implement the right interface? Check the semantics and remember: An instance of [entity] is always an instance of [parent], too.

✓ LENGTH

○	Does a class want to implement only part of this interface? Segregate it in different pieces as much as possible.
○	Is the class longer than the standard size for your project? The suggested length varies with the programming language and the particular application, but a long class may be the sign that you need to apply an Extract Class refactoring.
○	Is the class size of the same order of magnitude as other similar implementations?
○	How many characters long are the most complicated lines? You may want to introduce intermediate methods or data structures to keep such lines readable. A common rule of thumb is the 80 characters of the original text terminal.
○	Method length is also a useful metric. The rule of thumb is a method should be fully visible in a single screen, but this doesn't mean that with a 30" LCD monitor you should write longer methods. The original screen was 25 lines high, but you may want to extend it a bit for practical purposes and refactor later: Extract Local Method is one of the simplest refactorings and it has a very limited impact on the rest of the codebase.

ABOUT THE AUTHOR



Giorgio Sironi is a developer at Onebip. He searches for the harmony between form and context, which is a fancy way of saying he builds software to fit in the world he's in and keep up with its rapid changes. Giorgio's specific areas of expertise are testing, OOP design and distributed computing.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776