Clean Code Notes

**General rules**

1. Follow standard conventions.
2. Keep it simple stupid. Simpler is always better. Reduce complexity as much as possible.
3. Boy scout rule. Leave the campground cleaner than you found it.
4. Always find root cause. Always look for the root cause of a problem.

**Design rules**

1. Keep configurable data at high levels.
2. Prefer polymorphism to if/else or switch/case.
3. Separate multi-threading code.
4. Prevent over-configurability.
5. Use dependency injection.
6. Follow Law of Demeter. A class should know only its direct dependencies.

**Understandability tips**

1. Be consistent. If you do something a certain way, do all similar things in the same way.
2. Use explanatory variables.
3. Encapsulate boundary conditions. Boundary conditions are hard to keep track of. Put the processing for them in one place.
4. Prefer dedicated value objects to primitive type.
5. Avoid logical dependency. Don't write methods which works correctly depending on something else in the same class.
6. Avoid negative conditionals.

**Names rules**

1. Choose descriptive and unambiguous names.
2. Make meaningful distinction.
3. Use pronounceable names.
4. Use searchable names.
5. Replace magic numbers with named constants.
6. Avoid encodings. Don't append prefixes or type information.

**Functions rules**

1. Small.
2. Do one thing.
3. Use descriptive names.
4. Prefer fewer arguments.
5. Have no side effects.
6. Don't use flag arguments. Split method into several independent methods that can be called from the client without the flag.

**Comments rules**

1. Always try to explain yourself in code.
2. Don't be redundant.
3. Don't add obvious noise.
4. Don't use closing brace comments.
5. Don't comment out code. Just remove.
6. Use as explanation of intent.
7. Use as clarification of code.
8. Use as warning of consequences.

**Source code structure**

1. Separate concepts vertically.
2. Related code should appear vertically dense.
3. Declare variables close to their usage.
4. Dependent functions should be close.
5. Similar functions should be close.
6. Place functions in the downward direction.
7. Keep lines short.
8. Don't use horizontal alignment.
9. Use white space to associate related things and disassociate weakly related.
10. Don't break indentation.

**Objects and data structures**

1. Hide internal structure.
2. Prefer data structures.
3. Avoid hybrids structures (half object and half data).
4. Should be small.
5. Do one thing.
6. Small number of instance variables.
7. Base class should know nothing about their derivatives.
8. Better to have many functions than to pass some code into a function to select a behavior.
9. Prefer non-static methods to static methods.

**Tests**

1. One assert per test.
2. Readable.
3. Fast.
4. Independent.
5. Repeatable.

**Code smells**

1. Rigidity. The software is difficult to change. A small change causes a cascade of subsequent changes.
2. Fragility. The software breaks in many places due to a single change.
3. Immobility. You cannot reuse parts of the code in other projects because of involved risks and high effort.
4. Needless Complexity.
5. Needless Repetition.
6. Opacity. The code is hard to understand.

Longer version

**What Is Clean Code?**

* The code can be measured with either "good" or "bad" in the code review or by how many minutes it takes you to talk about it.
* A clean code should be elegant, efficient, readable, simple, without duplications, and well-written.
* You should add value to the business with your code.
* Clean code offers quality and understanding when we open a class.
* It is necessary that your code is clean and readable for anyone to find and easily understand. Avoid wasting others' time.

**Meaningful Names**

* Names of the classes, variables, and methods must be meaningful and clearly indicate what a method does or what an attribute is.
* Create pronounceable names to facilitate communication.
* Avoid acronyms and avoid confusing names, which may bring anyone who reads the code to the wrong conclusions.
* Use names that reflect the system domain, the context, and the problems that must be solved.

**Functions**

* The method should be easy to read and understand.
* The method should convey its intention.
* The methods should be small. Another rule for small methods is that they should be even lower.
* They must have up to 20 lines. (I think they should have up to 10 lines.)
* Methods should only do one thing: they should do it the right way and just do it.
* You should use names with words that say what it really does.
* The optimal number of parameters of a method is zero, after one and two.
* Three should be avoided, but if you think it should be used, have a good justification.
* Parameters of the Boolean type as a parameter already clearly states that it does more than one thing.
* Methods must do something and return something.
* Avoid duplication.

**Comments**

* [](http://4.bp.blogspot.com/-y67AMEukWcc/V5E3oC3giMI/AAAAAAAACiw/JtVOHDj89ssj27WldhuFx1GA_lKZdEAZACK4B/s1600/bla.jpeg)One of the most common reasons for the comments is because the code is bad.
* If you're thinking about writing a comment, then the code should be refactored.
* Comments do not save a bad code.
* Try to explain what the code causes to happen.
* Comments can be useful when placed in certain places.
* Create method names and informative variables instead of explaining the code with comments.
* Comments can be used to express the importance of certain points in the code.
* The best comment is one that needs to be written because your code already explained.
* Do not write comments with redundant, useless, or false information.
* They should not be used to indicate who changed or why, for that already exists in versioning.
* Don’t comment code that will not be used, remove, it just pollutes the code and leaves no doubt in anyone reading.

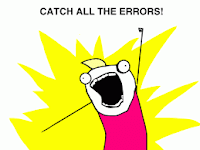
**Formatting**

* Formatting should indicate things of importance since it is a developer of communication form.
* A messy code is hard to read.
* The readability of the code will take effect on all of the changes that will be made.
* Try to write a class with a maximum of 500 lines. Smaller classes are easier to understand.
* Set a limit of characters per line of code.
* A good character limit on a line is 120.
* Try to keep more next related concepts vertically to create a code stream.
* Use spaces between operators, parameters, and commas.

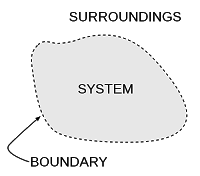
**Objects and Data Structure**

* Follow the Law of Demeter, which says that one *M* method of an object *O* can only consume services of the following types of objects:
  + The object itself, *O*.
  + The *M* parameters.
  + Any object created or instantiated by *M*.
  + Direct components of *O*.
* Make good abstraction and encapsulation.
* Do not make dumb objects.
* Objects hide the data abstraction and expose methods that operate the data.
* Data structures expose your data and do not have significant methods.

**Error Handling**

* [](http://4.bp.blogspot.com/-5Hpre4XxAbE/V5E346YFPNI/AAAAAAAACi4/VThj9NwVVXAPtR2vi1W5f0o46vXK3ugcwCK4B/s1600/erro.png)Error handling should be planned carefully by all programmers.
* When wrong things occur, we have to get it to do the right things.
* We should give preference to launching an exception than treating it just to hide.
* Create messages with information about the error.
* Mention that it failed. Where was this failure? If possible, mention why it failed.
* Look at separate business rules for errors and error handling.
* Avoid returning a NULL in methods, preferably to return an empty object.
* Avoid passing NULL to the methods; this can generate NullPointerExceptions.

**Boundary**

* [](http://2.bp.blogspot.com/-jGD9qO_9BgQ/V5E4Qpmck7I/AAAAAAAACjA/J8YPALYSvQksz_FNUTprKSSnyxBCKGrWgCK4B/s1600/500px-System_boundary.png)In third-party code, to avoid passing objects, APIs look forward in order to keep things in the same class.
* Performs tests on the API's third party.
* Study the documentation and test the third API before you start using it.
* Check well the features you will use.
* These steps can help increase yield when there are new updates to the API and you can only run your tests to check for this update.
* Create tests the functionality of the API.

**Unit Tests**

* Make sure each piece of code is doing what you expect it to do.
* Follow the TDDs law:
  + Don't create code before you have a failing test.
  + Don't create more tests than necessary to fail.
  + You cannot write more code than enough to pass the test that is failing.
* [](http://2.bp.blogspot.com/-YfgbMERjkX0/V5E4gcUVeII/AAAAAAAACjI/oPw4OqWQK3kBPwXV3vy3tl5O-fouZhxNQCK4B/s1600/test.jpeg)Keep your test clean.
* The tests must undergo changes in the same way that the code.
* The dirtier the code, the more difficult test will be to maintain.
* Use the F.I.R.S.T rule for testing:
  + The test is **fast**-running.
  + The tests are **independent** of other.
  + The test is **repeatable** in various environments.
  + The test is **self-validating**.
  + The test is **timely**.
* The test is as important as the production code.

**Classes**

* By default, Java classes should start with the variables:
  + Static and constantly public.
  + Static and variable private.
  + Instances and variables privates.
  + Soon after comes the functions.
* The class name should represent your responsibility.
* The class must have only one responsibility.
* To know the size of the class is ideal or we should not measure her responsibility.
* You should try to make a brief description of the class.
* The methods should be:
  + Small...
  + ...and even lower.
  + They must have only one responsibility.

**Systems**

* It is important to recognize and separate responsibilities of a system.
* It should be separate and modularize the logic execution, allowing an independent strategy for solving application dependency.
* Is important to take care of dependency injections and to allow only objects to take care of the business of logic.
* It is very difficult to create a system properly first. It must be made available to the story, then refactored, and then expanded to continue implementing new stories.
* To get to the point that TDD is necessary, you need refactoring and clean code.
* We must build POJOs-based logic through testing and evolve from simple to interconnect the various aspects necessary.

**Emergence**

Here are the rules that are given by Kent Beck to create good designs:

* **Run all tests.** They verify that the system behaves as expected.
* **Eliminate duplication** because duplicate code brings additional work.
* **To express the intention of the programmer**, use more expressive code to facilitate maintenance. Choose good names for functions, classes and tests shouldn’t be small and well written.
* **Minimize the number of classes and methods.** Following this pattern can ignore it if the classes are very small.
* **Apply all knowledge to improve the design during refactoring.** Increase cohesion, reduce coupling, separate responsibilities, reduce classes and methods, choose the best names.

Even applying it once, you will not be able to have good software. You need to do this over and over again to achieve continuous improvement.

**Concurrence**

* The concurrency is an aspect that may be present in the codes.
* Uncoupling allows for improving the yield and structure of an application.
* The concurrency can improve response times and application efficiency.
* You should consider the following ideas about the concurrence:
  + It injects a certain overload.
  + It can be complex to operate.
  + Errors caused by it can be difficult to reproduce.
  + It usually requires design changes.
* The concurrence problem is that different segments of an application may be following tangled multi-threading, which can cause unexpected problems in normal situations.
* For concurrence reasons, it is important that each class has a unique responsibility.
* Create sections that are synchronized and minimized. Running tests often is the best way to find any errors in the code. However, it is difficult to do when there are concurrence tests.
* A good way to test is to insert codes for testing in the middle of the implemented code.

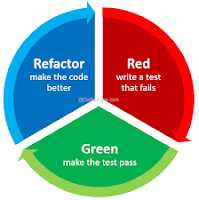
**Successive Refinement**

* The code-only work is not enough to have a good code.
* Professionals who care only about the code that works cannot be considered professional.
* We should ignore that we have no time to refactor to one code. The code that was not taken care of today can become a problem after becoming a problem for the team because no one will want to mess with it.
* Try not to let the code rot. It is much cheaper to create a clean code than cleaning a rotten code, as a move in a tangle can be an arduous task.
* The solution, then, comes down to maintaining the cleanest code possible and as simply as possible without ever letting it begin to rot.

**JUnit**

* Look to cover tests each (not every method, but each code line).
* No code is immune to improvement, and each of us has a responsibility to make the code a little better than we found it.
* Refactoring is an iterative process full of trial and error, inevitably converging to something that we feel is worthy of a professional.

**Refactoring**

[](http://4.bp.blogspot.com/-AuIr5GYLRes/V5E6YgMavcI/AAAAAAAACjs/7BMk-DsRDEE53xKybRK_7hSbLS4jA6PWwCK4B/s1600/tdd.png)

* Before making any kind of refactoring, it is important to have good coverage tests.
* After increasing or creating test coverage, you can begin to leave the clearest code and fix some bugs.
* Now, after leaving the code clearer, someone else can probably clean it even more.

**Java**

* Avoid long lists of import using \*.
* Do not inherit constants. Instead, use enums constants.

**Names**

* Choose descriptive names.
* Choose names at the appropriate level of abstraction.
* Use standard nomenclature where possible.
* Use long names for long scopes.
* Avoid encodings.
* Names should describe side effects.

**Conclusion**

Clean code is not written following a set of rules. You do not become a software professional just by learning a list of what you do and what you've done.

Professionalism and craftsmanship come from values and discipline in lists of what you should and should not do when creating a code.