Guidelines for examples

* **Keep examples at unit level.** There should be one to many examples related to the requirement. Each example should be attached to specific behavior of the class. Avoid the temptation to test an entire work-flow.
* **Test the trivial cases too.** It is sometimes recommended that all non-trivial cases should be tested and that trivial methods can be omitted.
  + Trivial is hard to define. It may mean different things to different people.
  + From a black-box perspective there is no way to know which part of the code is trivial.
  + The trivial cases can contain errors too, often as the result of copy-paste operations.

The recommendation is therefore to test everything. The trivial cases are simple to test after all.

* **Cover boundary cases.** Make sure the parameter boundary cases are covered. For numbers, test negatives, 0, positive, smallest, largest, NaN, infinity, etc. For strings test empty string, single character string, non-ASCII string, multi-MB strings etc. For collections test empty, one, first, last, etc. For dates, test January 1, February 29, December 31 etc. The class being tested will suggest the boundary cases in each specific case. The point is to make sure as many as possible of these are tested properly as these cases are the prime candidates for errors.
* **Test each feature once.** There is no need to come up with repetitive examples because it delays the work time
* **Design code with testing in mind.** Writing and maintaining unit tests are costly, and minimizing public API and reducing cyclomatic complexity in the code are ways to reduce this cost and make high-coverage test code faster to write and easier to maintain.

Some suggestions:

* + Make class members immutable by establishing state at construction time. This reduce the need of setter methods.
  + Restrict the use of excessive inheritance and virtual public methods.
  + Reduce the public API by utilizing friend classes (C++) or package scope (Java).
  + Avoid unnecessary branching.
  + Keep as little code as possible inside branches.
  + Make heavy use of exceptions and assertions to validate arguments in public and private API's respectively.
* **Know the limitations.** Unit tests can never prove the correctness of code!! A failing test may indicate that the code contains errors, but a succeeding test doesn't prove anything at all. The most useful appliance of unit tests are verification and documentation of requirements at a low level, and regression testing: verifying that code invariants remains stable during code evolution and refactoring. Consequently, unit tests can never replace a proper up-front design and a sound development process. Unit tests should be used as a valuable supplement to the established development methodologies.
* **Think black-box.** Act as a 3rd party class consumer, and test if the class fulfills its requirements. And try to tear it apart.
* **Provide a random generator.** When the boundary cases are covered, a simple way to improve test coverage further is to generate random parameters so that the tests can be executed with different input every time. To achieve this, provide a simple utility class that generates random values of the base types like doubles, integers, strings, dates etc. The generator should produce values from the entire domain of each type. If the tests are fast, consider running them inside loops to cover as many possible input combinations as possible. The following example verifies that converting twice between little endian and big endian representations gives back the original value. As the test is fast, it is executed on one million different values each time.

void testByteSwapper()

{

for (int i = 0; i < 1000000; i++) {

double v0 = Random.getDouble();

double v1 = ByteSwapper.swap(v0);

double v2 = ByteSwapper.swap(v1);

assertEquals(v0, v2);

}

}

* **Know the cost of testing.** Not writing unit tests is costly, but writing unit tests is costly too. There is a trade-off between the two, and in terms of execution coverage the typical industry standard is at about 80%. The typical areas where it is hard to get full execution coverage is on error and exception handling dealing with external resources. Simulating a database breakdown in the middle of a transaction is quite possible, but might prove too costly compared to extensive code reviews which is the alternative approach.