

Unit Testing Checklist

BY JEAN-BAPTISTE RIEU

A **unit test** is a set of methods launched frequently to validate your code. It is usually a good idea to write code in this order:

1. Write a class API
2. Write a method to test the API
3. Implement the API
4. Launch the unit tests









Why write unit tests? They validate current and future implementations. They measure code quality. They force you to write testable, loosely

coupled code. They're cheaper than manual regression testing. They build confidence in your code. They help teamwork.

Why use a checklist? Unit testing can be harder than actual implementation. Unit testing forces you to really think things through. But unit tests should be simple, direct, and easy to read and maintain. You also need to know when to stop writing tests and start writing the implementation.

Use this checklist to be sure your tests are really useful and to the point. *Remember:* the checklist helps you avoid big mistakes, but you need to make sure of the following:

✓	ICON KEY: ? reason why ! attention ex example i further information
<input type="radio"/>	My test class is testing only one class.
	? You are testing a class API to be sure the public contract is respected.
<input type="radio"/>	My methods are testing only one method at a time.
	! Be sure not to test private methods! They are hidden implementation details, not API.
<input type="radio"/>	My variables and method names are explicit.
	ex For example, store an expected value in an expectedFoo variable instead of just foo. If you test many combinations, use composed variable names like inputValue_NotNull, inputValue_ZeroData, inputValue_PastDate, etc. (according to your application's coding convention).
<input type="radio"/>	My test cases are easy to read by humans.
	? Future maintainers should be able to read your tests before reading the implementation. This will help them understand a class API before tweaking or debugging it.
<input type="radio"/>	My tests respect the usual clean code standards.
	i There should be no flow control in a test method (switch, if, etc.). A good test is just a very straightforward sequence of setup/validate instructions. If necessary, use sub-methods to factorize and make your tests easier to read. In case of multiple scenarios, use multiple test methods (one for each case).
	ex For example, a test method should fit on screen without scrolling – 1 to 20 lines long. If the method is longer, consider writing multiple test methods for each case instead of jamming them together.
<input type="radio"/>	My tests are also testing expected exceptions.
	ex In java, use <code>@Test(expected=MyException.class)</code> .

<input type="radio"/>	My tests don't need access to a database.
	 <i>Or if a test does need database access, then it must be a mocked, "fire and forget" temporary database that you fill with test cases for every new test method (use the Setup/Teardown methods to prepare it).</i>
<input type="radio"/>	My tests don't need access to network resources.
	 <i>You can't rely on third parties like network or device presence to validate a method (use mocks).</i>
<input type="radio"/>	My tests control side effects, limit values (max, min) and null variables (even when they throw exceptions).
	 <i>You want to make sure these problem cases never occur, even when the test won't be used during maintenance</i>
<input type="radio"/>	My tests can be run at any time, at any place without needing configuration or human intervention.
<input type="radio"/>	My tests pass for the current implementation and are easy to evolve.
	 <i>Tests really exist to support code evolution. If they are too hard to maintain or too light to refine the code, then they are a useless burden. (Many developers avoid unit testing for this reason.)</i>
<input type="radio"/>	My tests are concrete.
	 <i>In Java: don't use Date() as input for a method you are testing, but build a concrete date out of Calendar (don't forget to force the timezone). Other example: use name = "Smith"; instead of name = "name"; or name = "test";</i>
<input type="radio"/>	My tests use a mock to simulate/stub complex class structure or methods.
	 <i>Remember to test only one class API at a time.</i>
	 <i>Never test third-party libraries through your own classes. Libraries should come with their own tests (this is actually a good way to choose a library).</i>
<input type="radio"/>	My tests are never @ignored or commented out. Never. Ever.
<input type="radio"/>	My tests help me validate my architecture.
	 <i>If you can't test a method or a class, your design is not agile enough.</i>



My tests can run on any supported platform, not just the targeted platform.



Don't expect a particular device or hardware configuration. Otherwise, your tests will make migration tougher and you will be incentivized to disable them.



My tests are lightning fast!



Slow tests shouldn't drag you down. Speed encourages you to launch your tests often. It also helps to reduce building time on Continuous Integration systems.



Use a test runner that allows you to launch one test at a time while you are writing it. Use "delay" or "sleep" with caution – i.e., only in some edge cases, like waiting for notifications or clock-based methods.

ABOUT THE AUTHOR



Jean-Baptiste Rieu is a champion of rigorous software engineering, from intelligent code refactoring to good UI/UX design. His specialties include Java and iOS development, Eclipse RCP, JFreeChart, OpenGL, data visualization, continuous integration, and Agile methods. He has found that unit testing and other code validation techniques not only improve code quality but also help developers understand their applications better. Learn more at <http://jbrieu.info/>.

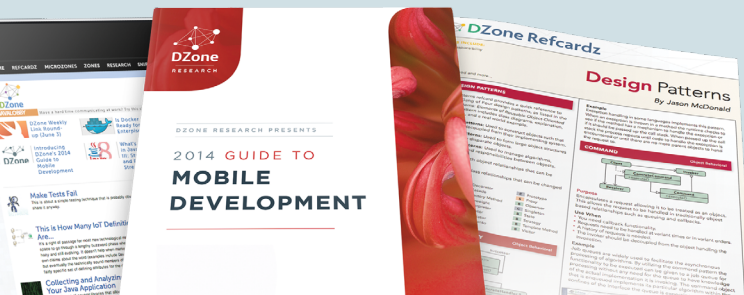
BROWSE OUR COLLECTION OF 250+ FREE RESOURCES, INCLUDING:

RESEARCH GUIDES: Unbiased insight from leading tech experts

REFCARDZ: Library of 200+ reference cards covering the latest tech topics

COMMUNITIES: Share links, author articles, and engage with other tech experts

JOIN NOW



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.

Copyright © 2014 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

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



9 781936 502776