Keeping Software Soft

A Practical Guide for Developers, Testers, and Managers

1st Edition

Jason Roberts

Keeping Software Soft

A Practical Guide for Developers, Testers, and Managers

Jason Roberts

This book is for sale at http://leanpub.com/KeepingSoftwareSoft

This version was published on 2018-08-31

ISBN 978-0-9924164-0-9

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

For my late mother who fed my imagination as a child and for my father for encouraging my love of science.

Thanks to all the early supporters of this book, including:

- *Mario Lopez*
- *Yolanda Maguire*
- *Gentry Riggen*
- *James Tweedie*

Contents

Introduction

Why Keep Software Soft?

Software should be easy to change.

Why should we care about keeping software soft? We don't have to care, but we accept that software that is not soft may be more aptly described as "rigidware".

Some argue that over time all software ends up as a big ball of goo: hard to understand, hard to learn, and hard to change.

We can respond to this argument in two ways: "why bother to even *try* and keep it soft then?"; or "lets try to slow the degradation as much as we can".

This still doesn't answer the question of *why* we should try and keep software soft; here are a few reasons:

- May result in lower total cost of ownership (TCO) for a longer-lived system
- May allow the business to be more agile (responding more quickly to changing marketplace conditions)
- May result in happier development teams (which in turn, may increase productivity and reduce staff turnover)

If these things aren't important to a business, if it has an unlimited supply of funds, time, and people; and the market in which it operates is slow moving and non-innovative, then this "rigidware" may well be acceptable.

Why Read This Book?

If you want to make developing software a happier, more enjoyable, and more productive experience this book is for you.

Keeping Software Soft provides practical guidance on how to keep software as soft as we can. You can read from beginning to end or just jump in at any point and use as a reference guide.

Part 1: The Code

It is amazing how much time is spent in management meetings talking about how to improve delivery speed and reduce costs. It sometimes seems that agile software development is seen as a silver-bullet that can fix all the ingrained problems an organisation has.

The strangest thing sometimes is how little energy is spent talking about improving the codebase itself.

Regardless of which software development life cycle process is being used, ultimately it is the code that must be transformed from its current state to a new state to support the goals of the organisation or users.

Part 1 of this book is focussed on the code itself, and how a codebase can be kept more pliable and easier to change.

The SOLID Principles of OO Design

THIS IS A SAMPLE PART OF THE SOLID CHAPTER

The SOLID acronym represents some important principles of object-oriented programming and design.

It stands for:

- **S** - Single Responsibility Principle
- **O** - Open Closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

This chapter examines each of these principles in term and relates them back to how they help to keep software soft.

The Single Responsibility Principle (SRP)

Any given class should do one well-defined thing, and **only** that one well-defined thing.

To put in another way, if a class has more than one responsibility, it will have to change whenever any one of it's responsibilities change. If there's more than one reason to change a class, its likely that the Single Responsibility Principle has been violated.

As an example, consider an `OrderManager` class that is responsible for: calculating the total order amount, debiting a credit card, and updating the order status in the database. Clearly this is a violation of SRP. Instead, a cleaner implementation could be to split out each responsibility into three separate classes: `OrderTotalsCalculator`, `CreditCardCharger`, and `OrderRepository`. Now each class only needs to change if it's responsibility changes.

Why SRP?

SRP is important to keeping software soft because:

- A class that conforms to SRP only needs to be changed when the thing it's responsible for changes
- Each class now has fewer lines of more highly-related code, so it should be easier for a developer to understand in its entirety

- More individual source code files (e.g. one for each responsible class) should result in fewer merge conflicts
- Testability is likely to be better and the related test code more highly focussed and cohesive

Without SRP there is greater fear (and potentially higher regression test costs) when changing any given part of the system.

Customer: "We want to change the way order totals are calculated." *Development Team*: "No problem, give us the new business rules, we'll write some tests then go and change the `OrderCalculator`, we don't have to worry about breaking the database because that's in a separate place".

Part 2: Testing

Part 2 of this book explores topics relating to the effective testing of software.

It covers topics (such as TDD) that are not wholly concerned with testing but that feature testing as a part of the overall activity.

The main focus is on tests that can be automated, whether at the code level or by automating the actual user interface.

A good suite of automated tests helps keep software soft by providing rapid feedback when things break. The longer a software defect ("bug") exists for, the more costly it becomes.

Tests can become a costly maintenance overhead, which is why it is important to take a holistic view of testing, a subject which is also covered in these chapters.

An Introduction to Unit Testing With xUnit.net

THIS IS A SAMPLE PART OF THE UNIT TESTING WITH XUNIT CHAPTER

Unit testing is the execution of discrete pieces of code to perform some action; and then checking (asserting) to see if they behaved as expected.

The purpose of having a suite of unit tests (or any type of test) is to increase the level of confidence that the system is behaving as expected.

The distinction of "**unit**" is important: a unit test should only test a small discreet piece of code (e.g. a class) and not rely on concrete dependencies such as other non-primitive objects or external things like databases and files.

Qualities of Good Unit Tests

In addition to only testing a discreet "unit", good unit tests posses a number of other qualities as outlined below.

Isolated and Independent

A given unit test should be able to be run at any time and in any order relative to other tests. When run it should not rely on the result/state of a previous test having been executed. Unit tests should also not rely on them being run on a specific server or network.

Repeatable

A given test should either pass or fail regardless of when it is run. The test should be able to be run an indefinite number of times, each time producing the same result (given that that the rest of the codebase stays the same).

Valuable

Any given unit test should add some value, i.e. if the test fails then the knowledge that "*something is broken over there*" should be of value to the development team. An example of a test that would add no value would be testing automatic property getters and setters; these are language features and should not be unit tested. Unit tests should only test custom code, not 3rd party or platform framework code.

Fast

Unit tests should execute quickly. They should be able to be run at any time by the developer and not hold them up for too long before they can continue coding. Unit tests that take too long to execute are likely to not be run as often as they should be which can result in defects existing for longer than they should do.

As a general rule, an individual unit test should execute in less than about half a second, though the specifics of the codebase and problem domain should be taken into account.

Trustworthy

The result of a test should be 100% reliable and the development team should not feel the need to doubt the results (e.g. by debugging the running test).

Readable

As with production code, test code should be as readable and of as high quality as production code.

Part 3: People and Process

People create code.

People follow a process when writing code - even if it is one of chaos.

Part three of this book discusses aspects relating to people, such as Software Craftsmanship and what makes programmers happy. Part three also looks at some processes related to software creation such as agile software development and pair programming.

At first glace some of these topics may not appear to be related to keeping software soft, but (for example) unhappy programmers may be more likely to write bad code and poor project management or time-management can create undue schedule pressure which leads to a temptation to write bad code.