THINGS EVERY PROGRAMMER SHOULD KNOW

EXTENDED

Compilation of essays from
programmer.97things.oreilly.com

Compiled by **Shirish Padalkar**

97 Things Every Programmer Should Know - Extended

Shirish Padalkar

This book is for sale at http://leanpub.com/97-Things-Every-Programmer-Should-Know-Extended

This version was published on 2014-09-23

Tweet This Book!

Please help Shirish Padalkar by spreading the word about this book on Twitter!

The suggested hashtag for this book is #97things.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#97things

To all the authors of essays in this book

Contents

Preface

Welcome to the **extended** version of 97 Things Every Programmer Should Know - Collective Wisdom from the Experts[1].

When I finished reading amazing 97 Things Every Programmer Should Know[2] book by Kevlin Henney[3], I loved it. Visiting the site made me realize that there are many more essays currently not included in the book.

This is a collection of those 68 additional essays from 97 Things Every Programmer Should Know[4] site.

The text in the book is taken from site **as is**. If you find any typographic error, please let us know and/or go ahead and update the original site.

Permissions

The licensing of each contribution follows a nonrestrictive, open source model. Every contribution is freely available online and licensed under a Creative Commons Attribution 3.0 License, which means that you can use the individual contributions in your own work, as long as you give credit to the original author:
http://creativecommons.org/licenses/by/3.0/us/[5]

About

I loved computers since I got introduced to computers in my school days. Started programming with QBasic, used WS4, Lotus-123, DOS and Windows 3.1.

Programming has been my passion. I work at **ThoughtWorks** and code for living. I love Java, Ruby, and I can read Python code. I had small time affairs with Haskell, LISP and Prolog as well.

Besides programming I like to find and report vulnerabilities in web applications.

I enjoy playing Tabla and love listening to Indian classical music.

Shirish Padalkar
https://twitter.com/_Garbage_

[1]http://shop.oreilly.com/product/9780596809492.do
[2]http://programmer.97things.oreilly.com
[3]http://programmer.97things.oreilly.com/wiki/index.php/Kevlin_Henney
[4]http://programmer.97things.oreilly.com
[5]http://creativecommons.org/licenses/by/3.0/us/

Acknowledgement

This is my first attempt to create a compilation of essays in book format. My colleagues at ThoughtWorks has really encouraged me to compile this book. Thank you guys, you are awesome.

Thank you Smita, wife of my colleague Kunal Dabir for beautiful cover artwork. It looks amazing.

Abstract Data Types

By Aslam Khan[6]

We can view the concept of *type* in many ways. Perhaps the easiest is that type provides a guarantee of operations on data, so that the expression `42 + "life"` is meaningless. Be warned, though, that the safety net is not always 100% secure. From a compiler's perspective, a type also supplies important optimization clues, so that data can be best aligned in memory to reduce waste, and improve efficiency of machine instructions.

Types offer more than just safety nets and optimization clues. A type is also an abstraction. When you think about the concept of type in terms of abstraction, then think about the operations that are "allowable" for an object, which then constitute its abstract data type. Think about these two types:

```
class Customer
  def totalAmountOwing ...
  def receivePayment ...
end

class Supplier
  def totalAmountOwing ...
  def makePayment ...
end
```

Remember that `class` is just a programming convenience to indicate an abstraction. Now consider when the following is executed.

```
y = x.totalAmountOwing
```

`x` is just a variable that references some data, an object. What is the type of that object? We don't know if it is `Customer` or `Supplier` since both types allow the operation `totalAmountOwing`. Consider when the following is executed.

[6]http://programmer.97things.oreilly.com/wiki/index.php/Aslam_Khan

```
y = x.totalAmountOwing
x.makePayment
```

Now, if successful, `x` definitely references an object of type `Supplier` since only the `Supplier` type supports operations of `totalAmountOwing` and `makePayment`. Viewing types as interoperable behaviors opens the door to polymorphism. If a type is about the valid set of operations for an object, then a program should not break if we substitute one object for another, so long as the substituted object is a subtype of the first object. In other words, honor the semantics of the behaviors and not just syntactical hierarchies. Abstract data types are organized around behaviors, not around the construction of the data.

The manner in which we determine the type influences our code. The interplay of language features and its compiler has led to static typing being misconstrued as a strictly compile-time exercise. Rather, think about static typing as the fixed constraints on the object imposed by the reference. It's an important distinction: The concrete type of the object can change while still conforming to the abstract data type.

We can also determine the type of an object dynamically, based on whether the object allows a particular operation or not. We can invoke the operation directly, so it is rejected at runtime if it is not supported, or the presence of the operation can be checked before with a query.

An abstraction and its set of allowable operations gives us modularity. This modularity gives us an opportunity to design with interfaces. As programmers, we can use these interfaces to reveal our intentions. When we design abstract data types to illustrate our intentions, then type becomes a natural form of documentation, that never goes stale. Understanding types helps us to write better code, so that others can understand our thinking at that point in time.

---

[7]http://creativecommons.org/licenses/by/3.0/us/

[8]http://programmer.97things.oreilly.com/wiki/index.php/Abstract_Data_Types

Acknowledge (and Learn from) Failures

By Steve Berczuk[9]

As a programmer you won't get everything right all of the time, and you won't always deliver what you said you would on time. Maybe you underestimated. Maybe you misunderstood requirements. Maybe that framework was not the right choice. Maybe you made a guess when you should have collected data. If you try something new, the odds are you'll fail from time to time. Without trying, you can't learn. And without learning, you can't be effective.

It's important to be honest with yourself and stakeholders, and take failure as an opportunity to improve. The sooner everyone knows the true state of things, the sooner you and your colleagues can take corrective action and help the customers get the software that they really wanted. This idea of frequent feedback and adjustment is at the heart of agile methods. It's also useful to apply in your own professional practice, regardless of your team's development approach.

Acknowledging that something isn't working takes courage. Many organizations encourage people to spin things in the most positive light rather than being honest. This is counterproductive. Telling people what they want to hear just defers the inevitable realization that they won't get what they expected. It also takes from them the opportunity to react to the information.

For example, maybe a feature is only worth implementing if it costs what the original estimate said, therefore changing scope would be to the customer's benefit. Acknowledging that it won't be done on time would give the stakeholder the power to make that decision. Failing to acknowledge the failure is itself a failure, and would put this power with the development team – which is the wrong place.

Most people would rather have something meet their expectations than get everything they asked for. Stakeholders may feel a sense of betrayal when given bad news. You can temper this by providing alternatives, but only if you believe that they are realistic.

Not being honest about your failures denies you a chance to learn and reflect on how you could have done better. There is an opportunity to improve your estimation or technical skills.

You can apply this idea not just to major things like daily stand-up meetings and iteration reviews, but also to small things like looking over some code you wrote yesterday and realizing that it was

[9]http://programmer.97things.oreilly.com/wiki/index.php/Steve_Berczuk

not as good as you thought, or admitting that you don't know the answer when someone asks you a question.

Allowing people to acknowledge failure takes an organization that doesn't punish failure and individuals who are willing to admit and learn from mistakes. While you can't always control your organization, you can change the way that you think about your work, and how you work with your colleagues.

Failures are inevitable. Acknowledging and learning from them provides value. Denying failure means that you wasted your time.

---

[10]http://creativecommons.org/licenses/by/3.0/

[11]http://programmer.97things.oreilly.com/wiki/index.php/Acknowledge_%28and_Learn_from%29_Failures

Anomalies Should not Be Ignored

By Keith Gardner[12]

Software that runs successfully for extended periods of time needs to be robust. Appropriate testing of long-running software requires that the programmer pay attention to anomalies of every kind and to employ a technique that I call *anomaly testing*. Here, anomalies are defined as unexpected program results or rare errors. This method is based on the belief that anomalies are due to causality rather than to gremlins. Thus, anomalies are indicators that should be sought out rather than ignored, as is typically the case.

Anomaly testing is the process of exposing the anomalies in the system though the following steps:

1. Augmenting your code with logging. Including counts, times, events, and errors.
2. Exercising/loading the software at sustainable levels for extended periods to recognize cadences and expose the anomalies.
3. Evaluating behaviors and anomalies and correcting the system to handle these situations.
4. Then repeat.

The bedrock for success is logging. Logging provides a window into the cadence, or typical run behavior, of your program. Every program has a cadence, whether you pay attention to it or not. Once you learn this cadence you can understand what is normal and what is not. This can be done through logging of important data and events.

Tallies are logged for work that is encountered and successfully processed, as well as for failed work, and other interesting dispositions. Tallies can be calculated by grepping through all of the log files or, more efficiently, they can be tracked and logged directly. Counts need to be tallied and balanced. Counts that don't add up are anomalies to be further investigated. Logged errors need to be investigated, not ignored. Paying attention to these anomalies and not dismissing them is the key to robustness. Anomalies are indicators of errors, misunderstandings, and weaknesses. Rare and intermittent anomalies also need to be isolated and pursued. Once an anomaly is understood the system needs to be corrected. As you learn your programs behavior you need to handle the errors in a graceful manner so they become handled conditions rather than errors.

In order to understand the cadence and expose the anomalies your program needs to be exercised. The goal is to run continuously over long periods of time, exercising the logic of the program.

[12]http://programmer.97things.oreilly.com/wiki/index.php/Keith_Gardner

I typically find a lot of idle system time overnight or over the weekend, especially on my own development systems. Thus, to exercise your program you can run it overnight, look at the results, and then make changes for the following night. Exercising as a whole, as in production, provides feedback as to how your program responds. The input stream should be close – if not identical – to the data and events you will encounter in production. There are several techniques to do this, including recording and then playing back data, manufacturing data, or feeding data into another component that then feeds into yours.

This load should also be able to be paced – that is, you need to start out slow and then be able to increase the load to push your system harder. By starting out slow you also get a feel for the cadence. The more robust your program is, the harder it can be pushed. Getting ahead of those rare anomalies builds an understanding of what is required to produce robust software.

---

[13]http://creativecommons.org/licenses/by/3.0/us/

[14]http://programmer.97things.oreilly.com/wiki/index.php/Anomalies_Should_not_Be_Ignored

Avoid Programmer Churn and Bottlenecks

By Jonathan Danylko[15]

Ever get the feeling that your project is stuck in the mud?

Projects (and programmers) always go through churn at one time or another. A programmer fixes something and then submits the fix. The testers beat it up and the test fails. It's sent back to the developer and the vicious cycle loops around again for a second, third, or even fourth time.

Bottlenecks cause issues as well. Bottlenecks happen when one or more developers are waiting for a task (or tasks) to finish before they can move forward with their own workload.

One great example would be a novice programmer who may not have the skill set or proficiency to complete their tasks on time or at all. If other developers are dependent on that one developer, the development team will be at a standstill.

So what can you do?

Being a programmer doesn't mean you're incapable of helping out with the project. You just need a different approach for how to make the project more successful.

- *Keep the communication lines flowing.* If something is clogging up the works, make the appropriate people aware of it.
- *Create a To-Do list for yourself of outstanding items and defects.* You may already be doing this through your defect tracking system like BugZilla, FogBugz, or even a visual board. Worst case scenario: Use Excel to track your defects and issues.
- *Be proactive.* Don't wait for someone to come to you. Get up and move, pick up the phone, or email that person to find out the answer.
- *If a task is assigned to you, make sure your unit tests pass inspection.* This is the primary reason for code churn. Just like in high school, if it's not done properly, you will be doing it over.
- *Adhere to your timebox.* This is another reason for code churn. Programmers sit for hours at a time trying to become the next Albert Einstein (I know, I've done it). Don't sit there and stare at the screen for hours on end. Set a time limit for how long it will take you to solve your problem. If it takes you more than your timebox (30 minutes/1 hour/2 hours/whatever), get another pair of eyes to look it over to find the problem.

[15] http://programmer.97things.oreilly.com/wiki/index.php/Jonathan_Danylko

- *Assist where you can.* If you have some available bandwidth and another programmer is having churn issues, see if you can help out with tasks they haven't started yet to release the burden and stress. You never know, you may need their help in the future.
- *Make an effort to prepare for the next steps in the project.* Take the 50,000-foot view and see what you can do to prepare for future tasks in the project.

If you are more proactive and provide a professional attitude towards the project, be careful: It may be contagious. Your colleagues may catch it.

---

[16]http://creativecommons.org/licenses/by/3.0/
[17]http://programmer.97things.oreilly.com/wiki/index.php/Avoid_Programmer_Churn_and_Bottlenecks

Balance Duplication, Disruption, and Paralysis

By Johannes Brodwall[18]

“I thought we had fixed the bug related to strange characters in names.”

“Well, it looks like we only applied the fix to names of organizations, not names of individuals.”

If you duplicate code, it’s not only effort that you duplicate. You also make the same decision about how to handle a particular aspect of the system in several places. If you learn that the decision was wrong, you might have to search long and hard to find all the places where you need to change. If you miss a place, your system will be inconsistent. Imagine if you remember to check for illegal character in some input fields and forgot to check in others.

A system with a duplicated decision is a system that will eventually become inconsistent.

This is the background for the common programmer credo “Don’t Repeat Yourself,” as the Pragmatic Programmers say, or “Once and only once,” which you will hear from Extreme Programmers. It is important not to duplicate your code. But should you duplicate the code of others?

The larger truth is that we have choice between three evils: duplication, disruption, and paralysis.

- We can duplicate our code, thereby duplicating effort and understanding, and being forced to hunt down bugs twice. If there’s only a few people in a team and you work on the same problem, eliminating duplication should almost always be way to go.
- We can share code and affect everyone who shares the code every time we change the code to better fit our needs. If this is a large number of people, this translates into lots of extra work. If you’re on a large project, you may have experienced code storms – days where you’re unable to get any work done as you’re busy chasing the consequences of other people’s changes.
- We can keep shared code unchanged, forgoing any improvement. Most code I – and, I expect, you – write is not initially fit for its purpose, so this means leaving bad code to cause more harm.

I expect there is no perfect answer to this dilemma. When the number of people involved is low, we might accept the noise of people changing code that’s used by others. As the number of people in a

[18]http://programmer.97things.oreilly.com/wiki/index.php/Johannes_Brodwall

project grows, this becomes increasingly painful for everyone involved. At some time, large projects start experiencing paralysis.

The scary thing about code paralysis is that you might not even notice it. As the impact of changes are being felt by all your co-workers, you start reducing how frequently you improve the code. Gradually, your problem drifts away from what the code supports well, and the interesting logic starts to bleed out of the shared code and into various nooks and crannies of your code, causing the very duplication we set out to cure. Although domain objects are obvious candidates for broad reuse, I find that their reusable parts usually limit themselves to data fields, which means that reused domain objects often end up being very anemic.

If we're not happy with the state of the code when paralysis sets in, it might be that there's really only one option left: To eschew the advice of the masters and duplicate the code.

---

[19]http://creativecommons.org/licenses/by/3.0/us/

[20]http://programmer.97things.oreilly.com/wiki/index.php/Balance_Duplication%2C_Disruption%2C_and_Paralysis