97 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/

[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

Be Stupid and Lazy

By Mario Fusco[21]

It may sound amazing, but you could be a better programmer if you were both lazier and more stupid.

First, you must be stupid, because if you are smart, or if you believe you are smart, you will stop doing two of the most important activities that make a programmer a good one: Learning and being critical of your own work. If you stop learning, it will make it hard to discover new techniques and algorithms that could allow you to work faster and more effectively. If you stop being critical, you will have a hard time debugging and refactoring your own work. In the endless battle between the programmer and the compiler, the best strategy for the programmer is to give up as soon as possible and admit that it is the programmer rather than the compiler who is most likely at fault. Even worse, not being critical will also cause you to stop learning from your own mistakes. Learning from your own mistakes is probably the best way to learn something and improve the quality of your work.

But there is a more important reason why a good programmer must be stupid. To find the best solutions to problems, you must always start from a clean slate, keeping an open mind and employing lateral thinking to explore all the available possibilities. The opposite approach does not work out so well: Believing you have the right solution may prevent you from discovering a better one. The less you know, the more radical and innovative your ideas will be. In the end, being stupid – or not believing yourself to be so intelligent – also helps you to remain humble and open to the advice and suggestions of your colleagues.

Second, a good programmer must also be lazy because only a lazy programmer would want to write the kinds of tools that might ultimately replace much of what the programmer does. The lazy programmer can avoid writing monotonous, repetitive code, which in turns allows them to avoid redundancy, the enemy of software maintenance and flexible refactoring. A byproduct of your laziness is a whole set of tools and processes that will speed up production. So, being a lazy programmer is all about avoiding dull and repetitive work, replacing it with work that's interesting and, in the process, eliminating future drudgery. If you ever have to do something more than once, consider automating it.

Of course, this is only half the story. A lazy programmer also has to give up to laziness when it comes to learning how to stay lazy – that is, which software tools make work easier, which approaches avoid redundancy, and ensuring that work can be maintained and refactored easily. Indeed, programmers

[21]http://programmer.97things.oreilly.com/wiki/index.php/Mario_Fusco

who are good and lazy will look out for tools that help them to stay lazy instead of writing them from scratch, taking advantage of the efforts and experience of the open source community.

Perhaps paradoxically, the road toward effective stupidity and laziness can be difficult and laborious, but it deserves to be traveled in order to become a better programmer.

---

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

[23]http://programmer.97things.oreilly.com/wiki/index.php/Be_Stupid_and_Lazy

Become Effective with Reuse

by Vijay Narayanan[24]

When I started programming, I wrote most of the code while implementing new features or fixing defects. I had trouble structuring the problem the right way and I wasn't always sure how to organize code effectively. With experience, I started to write less code, while simultaneously searching for ways to reuse the work of other developers. This is when I encountered the next challenge: How do I pursue reuse in a systematic way? How do I decide when it is useful to invest in building reusable assets? I was fortunate to work with effective software developers who stood out in their ability to systematically reuse software assets. The adage that "good developers code, great ones reuse" is very true. These developers continuously learn, employ their domain knowledge, and get a holistic perspective of software applications.

Continuous Learning

The most effective developers constantly improve their knowledge of software architecture and design. They are admired for their technical skills and yet never miss an opportunity to learn something new, be it a new way to solve a problem, a better algorithm, or a more scalable design. Dissatisfied with incomplete understanding, they dig deeper into concepts that will make them more effective, productive, and earn the respect of peers and superiors.

Domain Relevance

They also have a knack for selecting appropriate reusable assets that solve specific problems. Instead of reusing frameworks arbitrarily, they pick and choose software assets that are relevant to the problems at hand and the problems they can foresee. Additionally, they can recognize opportunities where a new reusable asset can add value. These developers have a solid understanding of how one asset fits with another in their problem domain. This helps them pick not just one but a whole family of related components in the domain. For instance, their insights and background help them determine whether a business entity is going to need a certain abstraction or what aspect of your design needs additional variability. Too often, in pursuit of reuse, a developer can end up adding needless design complexity. This is typically reflected in the code via meaningless abstractions and tedious configuration. Without a solid understanding of the domain, it is all too easy to add flexibility, cost, and complexity in the wrong areas.

[24]http://programmer.97things.oreilly.com/wiki/index.php/Vijay_Narayanan

Holistic Understanding

Having a handle on only a few components is an easy place to start but, if this remains the extent of a developer's knowledge of a system, it will inhibit scalability of systematic reuse efforts. The saying "the whole is greater than the sum of parts" is very relevant here. Software assets need to fulfill their functional and non-functional obligations, integrate well with other assets, and enable realization of new and innovative business capabilities. Recognizing this, these developers increase their understanding of the overall architecture. The architecture view will help them see both the functional and nonfunctional aspects of applications. Their ability to spot code smells and needless repetition helps them continuously refactor existing code, simplify design, and increase reusability. Similarly, they are realistic about the limitations of reusable assets and don't attempt to overdo it. Finally, they are good mentors and guide junior developers and peers. This is useful when deciding on new reusable assets or leveraging existing ones.

---

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

[26]http://programmer.97things.oreilly.com/wiki/index.php/Become_Effective_with_Reuse

Better Efficiency with Mini-Activities, Multi-Processing, and Interrupted Flow

By Siv Fjellkårstad[27]

As a smart programmer you probably go to conferences, have discussions with other smart programmers, and read a lot. You soon form your own view, based largely on the received wisdom of others. I encourage you to also think for yourself. Used in new contexts you might get more out of old concepts, and even get value from techniques which are considered bad practice.

Everything in life consists of choices, where you aim to choose the best option, leaving aside options that are not as appropriate or important. Lack of time, hard priorities, and mental blocks against some tasks can also make it easy to neglect them. You won't be able to do everything. Instead, focus on how to make time for things that are important to you. If you are struggling with getting started on a task, you may crack it by extracting one mini-activity at a time. Each activity must be small enough that you are not likely to have a mental block against it, and it must take "no time at all". One to five minutes is often just right. Extract only one or two activities at a time, otherwise you can end up spending your time creating "perfect" mini-activities. If the task is "introduce tests to the code," the first mini-activity might be "create the directory where the test classes should live." If the task is "paint the house," the first mini-activity could be "set the tin of paint down by the door."

Flow is good when you perform prioritized tasks, but flow may also steal a lot of time. Anyone who has surfed the Web knows this. Why does this happen? It's certainly easy to get into flow when you're having fun, but it can be hard to break out of it to do something less exciting. How can you restrict flow? Set a 15-minute timer. Every time it rings you must complete a mini-activity. If you want, you can then sit down again with a clear conscience – for another 15 minutes.

Now you've decided what to prioritize, made activities achievable, and released time by breaking out of flow when you're doing something useless. You've now spent all 24 of the day's available hours. If you haven't done all you wanted to by now, you must multi-process. When is multi-processing suitable? When you have downtime. Downtime is time spent on activities that you have to go through, but where you have excess capacity – such as when you are waiting for the bus or eating breakfast. If you can do something else as well during this time, you'll get more time for other activities. Be aware that downtime may come and prepare things to fill it with. Hang a sheet of paper

[27]http://programmer.97things.oreilly.com/wiki/index.php/Siv_Fjellk%C3%A5rstad

with something you want to learn on the bathroom wall. Bring an article with you for reading while waiting for the bus. Think about a problem you must solve as you walk from the car to your work. Do you want to spend more time outside? Suggest a “walking meeting” for your colleagues, go for a walk while you eat your lunch, or close your eyes and lower your shoulders. Try to get off the bus a few stops before you reach home, walk from there, and think of the memo you have in your pocket.

---

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

[29]http://programmer.97things.oreilly.com/wiki/index.php/Better_Efficiency_with_Mini-Activities%2C_Multi-Processing%2C_and_Interrupted_Flow

Code Is Hard to Read

By Dave Anderson[30]

Each programmer has an idea in their head regarding *hard to read* and *easy to read*. Readability depends on many factors:

- **Implementation language**. Some syntax just *is* easier to read than others. XSLT anyone?
- **Code layout and formatting**. Personal preferences and pet hates, like "Where do I place the curly brace?" and indentation.
- **Naming conventions**. `userStatus` versus `_userstatus` versus `x`.
- **Viewer**. Choice of IDE, editor, or other tool used will contribute to readability.

The short message is that code is hard to read! The amount of math and abstract thinking required to read through even the most elegant of programs is taken for granted by many programmers. This is why there are reams of good advice and tools available to help us produce readable code. The points above should be very easy for any professional programmer to handle, but I left out the fifth point, which is:

- **Solution design**. The most common problem – "I see what's happening here… but it could be done better."

This addresses the "art" in programming. The mind thinks a certain way and some solutions will just sit better than others. Not because of any technical or optimization reason, it will just "read better." As a programmer you should strive to produce a solution that addresses all five of these points.

A good piece of advice is to have someone else glance at your solution, or to come back to it a day later and see if you "get it" first time. This advice is common but very powerful. If you find yourself wondering "What does that method/variable do again?", refactor! Another good litmus test is to have a developer working in a different language read your code. It's a great test of quality to read some code implemented in a different language to the one you're currently using and see if you "get it" straight away. Most scripting languages excel at this. If you are working on Java you can glance at well-written Ruby/bash/DSL and pick it up immediately.

[30]http://programmer.97things.oreilly.com/wiki/index.php/Dave_Anderson

As a rule of thumb, programmers must consider these five factors when coding. *Implementation language* and *Solution design* are the most challenging. You have to find the right language for the job – no one language is the golden hammer. Sometimes creating your own Domain-Specific Language (DSL) can vastly improve a solution. There are many factors for *Solution design*, but many good concepts and principles have been around for many years in computer science, regardless of language.

All code is hard to read. You must be professional and take the time to ensure your solution has flow and reads as well as you can make it. So do the research and find evidence to back up your assumptions, unit test by method, and question dependencies – a simple, readable implementation is head and shoulders above a clever-but-confusing, look-at-me implementation.

---

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

[32]http://programmer.97things.oreilly.com/wiki/index.php/Code_Is_Hard_to_Read

Consider the Hardware

By Jason P Sage[33]

It's a common opinion that slow software just needs faster hardware. This line of thinking is not necessarily wrong but, like misusing antibiotics, it can become a big problem over time. Most developers don't have any idea what is really going on "under the hood." There is often a direct conflict of interest between best programming practices and writing code that screams on the given hardware.

First, let's look at your CPU's prefetch cache as an example. Most prefetch caches work by constantly evaluating code that hasn't even executed yet. They help performance by "guessing" where your code will branch to before it even has happened. When the cache "guesses" correctly, it's amazingly fast. If it "guesses" wrong, on the other hand, all the preprocessing on this "wrong branch" is useless and a time-consuming cache invalidation occurs. Fortunately, it's easy to start making the prefetch cache work harder for you. If you code your branch logic so that the *most frequent result* is the condition that is tested for, you will help your CPU's prefetch cache be "correct" more often, leading to fewer CPU-expensive cache invalidations. This sometimes may read a little awkwardly, but systematically applying this technique over time will decrease your code's execution time.

Now, let's look at some of the conflicts between writing code for hardware and writing software using mainstream best practices.

Folks prefer to write many small functions in favor of larger ones to ease maintainability, but all those function calls come at a price! If you use this paradigm, your software may spend more time preparing and recovering from work than actually doing it! The much loathed `goto` or `jmp` command is the fastest method to get around followed closely by machine language indirect addressing jump tables. Functions are great for humans but from the CPU's point of view they're expensive.

What about inline functions? Don't inline functions trade program size for efficiency by copying function code inline versus jumping around? Yes they do! But even when you specify a function is to be inlined, can you be sure it was? Did you know some compilers turn regular functions into inline ones when they feel like it and vice versa? Understanding the machine code created by your compiler from your source code is extremely important if you wish to write code that will perform optimally for the platform at hand.

Many developers think abstracting code to the nth degree, and using inheritance, is just the pinnacle of great software design. Sometimes constructs that look great conceptually are terribly inefficient

[33] http://programmer.97things.oreilly.com/wiki/index.php/Jason_P_Sage

in practice. Take for example inherited virtual functions: They are pretty slick but, depending on the actual implementation, they can be very costly in CPU clock cycles.

What hardware are you developing for? What does your compiler do to your code as it turns it to machine code? Are you using a virtual machine? You'll rarely find a single programming methodology that will work perfectly on all hardware platforms, real or virtual.

Computer systems are getting faster, smaller and cheaper all the time, but this does not warrant writing software without regards to performance and storage. Efforts to save clock CPU cycles and storage can pay off as dividends in performance and efficiency.

Here's something else to ponder: New technologies are coming out all the time to make computers more *green* and ecosystem friendly. Efficient software may soon be measured in power consumption and may actually affect the environment!

Video game and embedded system developers know the hardware ramifications of their compiled code. Do you?

---

[34]http://creativecommons.org/licenses/by/3.0/us/
[35]http://programmer.97things.oreilly.com/wiki/index.php/Consider_the_Hardware

Continuous Refactoring

By Michael Hunger[36]

Code bases that are not cared for tend to rot. When a line of code is written it captures the information, knowledge, and skill you had at that moment. As you continue to learn and improve, acquiring new knowledge, many lines of code become less and less appropriate with the passage of time. Although your initial solution solved the problem, you discover better ways to do so.

It is clearly wrong to deny the code the chance to grow with knowledge and abilities.

While reading, maintaining, and writing code you begin to spot pathologies, often referred to as *code smells*. Do you notice any of the following?

- Duplication, near and far
- Inconsistent or uninformative names
- Long blocks of code
- Unintelligible boolean expressions
- Long sequences of conditionals
- Working in the intestines of other units (objects, modules)
- Objects exposing their internal state

When you have the opportunity, try deodorizing the smelly code. Don't rush. Just take small steps. In Martin Fowler's *Refactoring* the steps of the refactorings presented are outlined in great detail, so it's easy to follow. I would suggest doing the steps at least once manually to get a feeling for the preconditions and side effects of each refactoring. Thinking about what you're doing is absolutely necessary when refactoring. A small glitch can become a big deal as it may affect a larger part of the code base than anticipated.

Ask for help if your gut feeling does not guide you in the right direction. Pair with a co-worker for the refactoring session. Two pairs of eyes and sets of experience can have a significant effect – especially if one of these is unclouded by the initial implementation approach.

We often have tools we can call on to help us with automatic refactoring. Many IDEs offer an impressive range of refactorings for a variety of languages. They work on the syntactically sound parse tree of your source code, and can often refactor partially defective or unfinished source code. So there is little excuse for not refactoring.

[36]http://programmer.97things.oreilly.com/wiki/index.php/Michael_Hunger

If you have tests, make sure you keep them running while you are refactoring so that you can easily see if you broke something. If you do not have tests, this may be an opportunity to introduce them for just this reason, and more: The tests give your code an environment to be executed in and validate that the code actually does what is intended, i.e., passes the tests.

When refactoring you often encounter an epiphany at some point. This happens when suddenly all puzzle pieces fall into the place where they belong and the sum of your code is bigger than its parts. From that point it is quite easy to take a leap in the development of your system or its architecture.

Some people say that refactoring is waste in the Lean sense as it doesn't directly contribute to the business value for the customer. Improving the design of the code, however, is not meant for the machine. It is meant for the people who are going to read, understand, maintain, and extend the system. So every minute you invest in refactoring the code to make it more intelligible and comprehensible is time saved for the soul in future that has to deal with it. And the time saved translates to saved costs. When refactoring you learn a lot. I use it quite often as a learning tool when working with unfamiliar codebases. Improving the design also helps spotting bugs and inconsistencies by just seeing them clearly now. Deleting code – a common effect of refactoring – reduces the amount of code that has to be cared for in the future.

---

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

[38]http://programmer.97things.oreilly.com/wiki/index.php/Continuous_Refactoring

Continuously Align Software to Be Reusable

by Vijay Narayanan[39]

The oft cited reason for not being able to build reusable software is the lack of time in the development process. Agility and refactoring are your friends for reuse. Take a pragmatic approach to the reuse effort and you will increase the odds of success considerably. The strategy that I have used with building reusable software is to pursue continuous alignment. *What exactly is continuous alignment?*

The idea of continuous alignment is very simple: Place value on making software assets reusable continuously. Pursue this across every iteration, every release, and every project. You may not make many assets reusable on day one, and that is perfectly okay. The key thing is to align software assets closer and closer to a reusable state using relentless refactoring and code reviews. Do this often and over a period of time you will transform your codebase.

You start by aligning requirements with reusable assets and do so across development iterations. Your iteration has tangible features that are being implemented. They become much more effective if they are aligned with your overall vision. This isn't meant to make every feature reusable or every iteration produce reusable assets. You want to do just the opposite. Continuous alignment accepts that building reusable software is hard, takes time, and is iterative. You can try to fight that and attempt to produce perfectly reusable software first time. But this will not only add needless complexity, it will also needlessly increase schedule risk for projects. Instead, align assets towards reuse slowly, on demand, and in alignment with business needs.

A simple example will make this approach more concrete. Say you have a piece of code that accesses a legacy database to fetch customer email addresses and send email messages. The logic for accessing the legacy database is interspersed with the code that sends emails. Say there is a new business requirement to display customer email data on a web application. Your initial implementation can't reuse existing code to access customer data from the legacy system. The refactoring effort required will be too high and there isn't enough time to pursue that option. In a subsequent iteration you can refactor the email code to create two new components: One that fetches customer data and another that sends email messages. This refactored customer data component is now available for reuse with the web application. This change can be made in one, two, or many iterations. If you cannot get it done, you can include it to on your list of known outstanding refactorings along with existing tasks.

[39]http://programmer.97things.oreilly.com/wiki/index.php/Vijay_Narayanan

When the next project comes around and you get a requirement to access additional customer data from the web application, you can work on the outstanding refactoring.

This strategy can be used when refactoring existing code, wrapping legacy service capabilities, or building a new asset's features iteratively. The fundamental idea remains the same: Align project backlog and refactorings with reuse objectives. This won't always be possible and that is OK! Agile practices advocate exploration and alignment rather than prediction and certainty. Continuous alignment simply extends these ideas for implementing reusable assets.

---

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

[41]http://programmer.97things.oreilly.com/wiki/index.php/Continuously_Align_Software_to_Be_Reusable

Data Type Tips

By Jason P Sage[42]

The reserved words `int`, `shortint`, `short`, and `smallint` are a few names, taken from only two programming languages, that indicate a two-byte signed integer.

The names and storage mechanisms for various kinds of data have become as varied as the colors of leaves in New England in the fall. This really isn't so bad if you spend all your time programming in just one language. However, if you're like most developers, you are probably using a number of languages and technologies, which requires you to write code to convert data from one data type to another frequently.

Many developers for one reason or another resort to using *variant* data types, which can further complicate matters, require more CPU processing, and are usually abused. Variant data types definitely have their place but they are often abused. The fact is that a programmer should understand the strengths, weaknesses and implications of using any data type. One good example of where variants might be employed are functions specifically designed to accept and handle various types of data that might be passed into one or more variant parameters. One bad example of using variants would be to use them so frequently that language data type rules are effectively nullified.

You can ease data type complexity when writing conversions by using an apples to apples common reference point to describe data in much the same way that many countries with varied cultures and tongues have a common, standard language to speak. The benefit of designing your code around such an idea results in modular reusable code that makes sense and centralizes data conversion to one place.

The following data types are just commonplace subset of what is available and can store just about anything:

boolean	*true* or *false*	
single-byte char		
unicode char		
unsigned integer		8 bit

[42] http://programmer.97things.oreilly.com/wiki/index.php/Jason_P_Sage

unsigned integer	16 bit
unsigned integer	32 bit
unsigned integer	64 bit
signed integer	8 bit
signed integer	16 bit
signed integer	32 bit
signed integer	64 bit
float	32 bit
double	64 bit
string	undetermined length
string	fixed length
unicode string	undetermined length
unicode string	fixed length
unspecified binary object	undetermined length

The trick is to write code to convert your various data types to your "common tongue" and alternately write code to convert them back. If you do this for the various systems in your organization, you will have a data-type conversion code base that can move data to and from every system you did this for. This will speed data conversion tremendously.

This same technique works for moving data to and from disparate database software, accounting SDK interfaces, CRM systems, and more.

Now, converting and moving complex data types such as record structures, linked lists, and database tables obviously complicates things. Nonetheless, the same principles apply. Whenever you create a staging area whose layout is well defined, like the data types listed above, and write code to move data into a structure from a given source as well as the mechanism to move it back, you create valuable programming opportunities.

To summarize, it's important to consider what each data type offers and their implications in the language they are used in. Additionally, when considering systems integrations where disparate technologies are in use, it is wise to know how data types map between the systems to prevent dataloss.

Most organizations are very aware of the fetters that vendor lock-in creates. By devising a common tongue for all your systems to speak in, you manufacture a powerful tool to loosen those bonds.

The details may be in the data, but the data is stored in your data types.

---

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

[44]http://programmer.97things.oreilly.com/wiki/index.php/Data_Type_Tips

Declarative over Imperative

By Christian Horsdal[45]

Writing code is error-prone. Period. But writing imperative code is much more error-prone than writing declarative code. Why? Because imperative code tries to tell the machine what to do step by step, and usually there are lots and lots of steps. In contrast to this, declarative code states what the intent is, and leaves the step by step details to centralized implementations or even to the platform. Just as importantly, declarative code is easier to read, for the exact same reason: Declarative code expresses intent rather than the method for achieving the intent.

Consider the following imperative C# code for parsing an array of command line arguments:

```
public void ParseArguments(string[] args)
{
    if (args.Contains("--help"))
    {
        PrettyPrintHelpTextForArguments();
        return;
    }
    if (args.Length % 2 != 0)
        throw new ArgumentException("Number of arguments must be even");
    for (int i = 0; i < args.Length; i += 2)
    {
        switch (args[i])
        {
            case "--inputfile":
                inputfileName = args[i + 1];
                break;
            case "--outputfile":
                outputfileName = args[i + 1];
                break;
            case "--count":
                HandleIntArgument(args[i], args[i + 1], out count);
                break;
```

[45] http://programmer.97things.oreilly.com/wiki/index.php/Christian_Horsdal

```
            default:
                throw new ArgumentException("Unknown argument");
        }
    }
}
private void PrettyPrintHelpTextForArguments()
{
    Console.WriteLine("Help text explaining the program.");
    Console.WriteLine("\t--inputfile: Some helpful text");
    Console.WriteLine("\t--outputfile: Some helpful text");
    Console.WriteLine("\t--count: Some helpful text");
}
```

To add another recognized argument we have to add a `case` to the `switch`, and then remember to extend the help text printed in response to the `--help` argument. This means that the additional code needed to support another argument is spread out between two methods.

The declarative alternative demonstrates a simple yet powerful lookup-based declarative coding style. The declarative version is split in two: Firstly a declarative part, that declares the recognized arguments:

```
class Argument
{
    public Action<DeclarativeArgParser, string> processArgument;
    public string helpText;
}
private readonly SortedDictionary<string, Argument> arguments =
  new SortedDictionary<string, Argument>()
  {
      {"--inputfile",
       new Argument()
       {
           processArgument = (self, a) => self.inputfileName = a,
           helpText = "some helpful text"
       }
      },
      {"--outputfile",
       new Argument()
       {
           processArgument = (self, a) => self.outputfileName = a,
           helpText = "some helpful text"
       }
      },
      {"--count",
       new Argument()
       {
           processArgument =
              (self, a) => HandleIntArgument("count", a, out self.count),
           helpText = "some helpful text"
       }
      }
  };
```

Secondly an imperative part that parses the arguments:

```
public void ParseArguments(string[] args)
{
    if (args.Contains("--help"))
    {
        PrettyPrintHelpTextFieldsForArguments();
        return;
    }
    if (args.Length % 2 != 0)
        throw new ArgumentException("Number of arguments must be even");
    for (int i = 0; i < args.Length; i += 2)
    {
        var arg = arguments[args[i]];
        if (arg == null)
            throw new ArgumentException("Unknown argument");
        arg.processArgument(this, args[i + 1]);
    }
}
private void PrettyPrintHelpTextFieldsForArguments()
{
    Console.WriteLine("Help text explaining the program.");
    foreach (var arg in arguments)
        Console.WriteLine("\t{0}: {1}", arg.Key, arg.Value.helpText);
}
```

This code is similar to the imperative version above, except for the code inside the loop in `ParseArguments`, and the loop in `PrettyPrintHelpTextFieldsForArguments`.

To add another recognized argument a new key and `Argument` pair is simply added to the dictionary initializer in the declarative part. The type `Argument` contains exactly the two fields needed by the second part of the code. If both fields of `Argument` are initialized correctly everything else should just work. This means that the additional code needed to support another argument is localized to one place: The declarative part. The imperative part need not be touched at all. It just works regardless of the number of supported arguments.

At times there are further advantages to declarative style code: The clearer statement of intent sometimes enables underlying platform code to optimize the method of achieving the intended goal. Often this results in better performing, more scalable, or more secure software than would have been achieved using an imperative approach.

All in all prefer declarative code over imperative code.

---

[46]http://creativecommons.org/licenses/by/3.0/us/
[47]http://programmer.97things.oreilly.com/wiki/index.php/Declarative_over_Imperative

Decouple that UI

By George Brooke[48]

Why decouple the UI from the core application logic? Such layering gives us the ability to drive the application via an API, the interface to core logic without involving the UI. The API, if well designed, opens up the possibility of simple automated testing, bypassing the UI completely. Additionally, this decoupling leads to a superior design.

If we build a UI layer cleanly separated from the rest of our system, the interface beneath the UI can be an appropriate point in which to inject a record/replay mechanism. The record/replay can be implemented via a simple serial file. As we are typically simulating user input with the record/replay mechanism, there is not usually a need for very high performance, but if you want it, you can build it.

This separation of UI testing from functional testing is constrained by the richness of the interface between the UI and the core system. If the UI gets massively reorganized then so necessarily does any attached mechanism. From the point of view of tracking changes and effects, once the system is baselined it is probably a good idea to baseline any record/replay logs in the event of needing to identify some subsequent change in system behavior. None of this is particularly difficult to do providing that it is planned in to the project and, eventually, there is a momentum in terms of knowledgeable practitioners in this part of the black art of testing.

Downsides: There is always at least one… usually that the investment in recording and replaying what are typically suites of regression tests becomes a millstone for the project. The cost of change to the suite becomes so high that it influences what can economically be newly implemented. The design of reusable test code requires the same skills as those for designing reusable production code.

Upsides: Regression testing is not sensitive to cosmetic changes in the UI, massive confidence in new releases, and providing that all error triggers are retrofitted into the record/replay tests, once a bug is fixed it can never return! Acceptance tests can be captured and replayed as a smoke test giving a minimum assured level of capability at any time.

Finally, just because the xUnit family of tools is associated with unit testing, they do not have to be restricted to this level. They can be used to drive these system-wide activities,via the UI-API as described above, providing a uniform approach to all tests at all levels.

---

[48]http://programmer.97things.oreilly.com/wiki/index.php/George_Brooke

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

[50]http://programmer.97things.oreilly.com/wiki/index.php/Decouple_that_UI

Display Courage, Commitment, and Humility

By Ed Sykes[51]

Software engineers are always having good ideas. Often you'll see something that you think can be done better. But complaining is not a good way to make things better.

Once, in an informal developer meeting, I saw two different strategies of changing things for the better. James, a software engineer, believed that he could reduce the number of bugs in his code using Test-Driven Development. He complained that he wasn't allowed to try it. The reason he wasn't allowed was that the project manager, Roger, wouldn't allow it. Roger's reasons were that adopting it would slow down development, impacting the deliverables of the project, even if it eventually lead to higher quality code.

Another software engineer, Harry, piped up to tell a different story. His project was also managed by Roger, but Harry's project was using TDD.

How could Roger have had such a different opinion from one project to the other?

Harry explained how he had introduced Test-Driven Development. Harry knew from experimentation at home that he could write better software using TDD. He decided to introduce this to his work. Understanding that the deliverables on the project still had to be met, he worked extra hours so they wouldn't be jeopardised. Once TDD was up and running, the extra time spent writing tests was reclaimed through less debugging and fewer bug reports. Once he had TDD set up he then explained what he had done to Roger, who was happy to let it continue. Having seen that his deliverables were being met, Roger was happy that Harry was taking responsibility for improving code quality, without affecting the criteria that Roger's role was judged on, meeting agreed deliverables.

Harry had never spoken about this before because he hadn't known how it would turn out. He was also aware that bragging about the change he had introduced would affect the perceptions of Roger and Roger's bosses, the management team, would have of it. Harry praised Roger for allowing TDD to remain in the project, despite Roger having voiced doubts about it previously. Harry also offered to help anyone who wanted to use TDD to avoid mistakes that he had made along the way.

Harry showed the courage to try out something new. He committed to the idea by being prepared to try it at home first and then by being prepared to work longer to implement it. He showed humility

[51]http://programmer.97things.oreilly.com/wiki/index.php/Ed_Sykes

by talking about the idea only when the time was right, praising Roger's role and by being honest about mistakes he made.

As software engineers we sometimes need to display courage to make things better. Courage is nothing without commitment. If your idea goes wrong and causes problems you must be committed to fixing the problems or reverting your idea. Humility is crucial. It's important to admit and reflect upon the mistakes you make to yourself and to others because it helps learning.

When we feel frustrated and powerless the emotion can escape as a complaint. Focusing that emotion through the lens of hard work can turn that negativity into a persuasive positive force.

---

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

[53]http://programmer.97things.oreilly.com/wiki/index.php/Display_Courage%2C_Commitment%2C_and_Humility

Dive into Programming

By Wojciech Rynczuk[54]

You may be surprised by how many parallels between two apparently different activities like scuba diving and programming can be found. The nature of each activity is inherently complex, but is unfortunately often reduced to making bubbles or creating snippets of code. After successfully completing a scuba course and receiving a diving certificate, would-be divers start their underwater adventure. Most divers apply the knowledge and skills they acquired during the course. They rely on the exact measurements done by their diving computers and follow the rules which allow them to survive in the hazardous underwater environment. However, they are still newbies lacking experience. Hence, they break the rules and frequently underestimate threats or fail to recognize danger, putting their own lives – and very often the lives of others – at risk.

To create good and reliable designs, programming also requires a good theoretical background supported by practice. Although most programmers are taught how to follow a software process appropriately, all too often they undervalue or overlook the role of testing while designing and coding the application. Unit and integration tests should be considered inseparable parts of any software module. They prove the correctness of the unit and are sovereign when introducing further changes to the unit. Time spent preparing the tests will pay off in future. So, keep testing in mind from the very start of the project. The likelihood of failure will be significantly reduced and the chances of the success will increase.

The *buddy system* is often used in diving. From Wikipedia[55]:

> The “buddies” are expected to monitor each other, to stay close enough together to be able to help in an emergency, to behave safely and to follow the plan agreed by the group before the dive.

Programmers should also have buddies. Regardless of organizational process, one should have a reliable buddy, preferably an expert in the field who can offer a thorough and clear review of any work. It is essential that the output of every cycle of software production is evaluated because each of these steps is equally important. Designs, code, and tests all need considered peer review. One benefit of peer review is that both sides, the author and the reviewer, can take advantage of the review to

[54]http://programmer.97things.oreilly.com/wiki/index.php/Wojciech_Rynczuk
[55]http://en.wikipedia.org/wiki/Buddy_diving

learn from one another. To reap the benefits of the meeting both parties should be prepared for it. In the case of a code review, the sources ought to be previously verified, e.g., by static analysis tools.

Last, but not least, programming calls for precision and an in-depth understanding of the project's domain, which is ultimately as important as skill in coding. It leads to a better system architecture, design, and implementation and, therefore, a better product. Remember that diving is not just plunging into water and programming is not just cranking out code. Programming involves continuously improving one's skills, exploring every nook and cranny of engineering, understanding the process of software creation, and taking an active role in any part of it.

---

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

[57]http://programmer.97things.oreilly.com/wiki/index.php/Dive_into_Programming

Don’t Be a One Trick Pony

by Rajith Attapattu[58]

If you only know $LANG and work on operating system $OS then here are some suggestions on how to get out of the rut and expand your repertoire to increase your marketability.

- If your company is a $LANG-only shop, and anything else treated like the plague, then you could scratch your itch in the open source world. There are many projects to choose from and they use a variety of technologies, languages, and development tools which are sure to include what you are looking for. Most projects are meritocratic and don’t care about your creed, country, color, or corporate background. What matters is the quality of your contribution. You could start by submitting patches and work at your own pace to earn karma.
- Even though your company’s products are written only in $LANG for reasons that are beyond your control, it may not necessarily apply to the test code. If your product exposes itself over the network you could write your test clients in a language other than $LANG. Also most VMs support several scripting languages which allows processes to be driven and tested locally. Java can be tested with Scala, Groovy, JRuby, JPython, etc. and C# can be tested with F#, IronRuby, IronPython, etc.
- Even a mundane task can be turned into an interesting opportunity to learn. The Wide Finder project is an example of exploring how efficiently you can parse a log file using different languages. If you are collecting test results by hand and graphing them using Excel, how about writing a program in your desired language to collect, parse, and graph the results? This increases your productivity as it automates repetitive tasks and you learn something else in the process as well.
- You could leverage multiple partitions or VMs to run a freely available OS on your home PC to expand your Unix skills.
- If your employment contract prohibits you from contributing to open source, and you are stuck with doing grunt work with $LANG, have a look at project Euler. There’s a series of mathematical problems organized in to different skill levels. You can try your hand at solving those problems in any languages you are interested in learning.
- Traditionally books have been an excellent source for learning new stuff. If you are not the type to read books, there are a growing number of podcasts, videos, and interactive tutorials that explain technologies and languages.

[58]http://programmer.97things.oreilly.com/wiki/index.php/Rajith_Attapattu

- If you are stuck while learning the ropes of a particular language or technology, the chances are that somebody else has already run into the same problem, so google your question. It's also a good idea to join a mailing list for the language or technology you are interested in. You don't need to rely on your organization or your colleagues.
- Functional programming is not just for Lisp, Haskell, or Erlang programmers. If you learn the concepts you can apply them in Python, Ruby, or even in C++ to arrive at some elegant solutions.

It is your responsibility to improve your skills and marketability. Don't wait for your company or your manager to prod you to try your hand at learning new things. If you have a solid foundation in programming and technology, you can easily transfer these skills into the next language you learn or next technology you use.

---

[59]http://creativecommons.org/licenses/by/3.0/us/
[60]http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Be_a_One_Trick_Pony

Don’t Be too Sophisticated

By Ralph Winzinger[61]

How deep is your knowledge of your programming language of choice? Are you a real expert? Do you know every strange construct that the compiler won’t reject? Well, maybe you should keep it to yourself.

Most of the time a programmer does not write code just for himself. It’s very likely that he creates it within a project team or shares it in some other way. The point is that there are other people who will have to deal with the code. And since other people will deal with the code, some programmers want to make it just brilliant. The code will be formatted according to the chosen style guide and I’m sure that it will be well commented or documented. And it will use sophisticated constructs to solve the underlying problems in the most elegant way. And that’s sometimes exactly where the problems start.

If you look at development teams you will notice that most of their members are average-level programmers. They do a good job with the mainstream features of their programming languages, but they will see sophisticated code as pure magic. They can see that the code is working, but they don’t understand why. This leads to one problem we know all about: Although well formatted and commented, the code is still hard to maintain. You might get into trouble if there is a need for enhancement or to fix a bug when the magician is not within reach. There might even be a second problem: People tend to reject what they do not understand. They might refuse to use this brilliant solution, which makes it a little less brilliant after all. They might even blame the sophisticated solution if there are some strange problems. Thus, before creating a highly sophisticated solution, you should take a step back and ask yourself whether this kind of solution is really needed or not. If so, hide the details and publish a simple API.

So what is the thing the programmer should know? Try to speak the same language as the rest of your team. Try to avoid overly sophisticated constructs and keep your solutions comprehensible to all. This does not mean that your team should always stay on an average programming level without improvements. Just don’t improve the level by applying magic: Take your team with you. Give your knowledge to your team members when appropriate and do it slowly, step by step.

---

[61]http://programmer.97things.oreilly.com/wiki/index.php/Ralph_Winzinger

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

[63]http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Be_too_Sophisticated

Don’t Reinvent the Wheel

By Kai Tödter[64]

Every software developer wants to create new and exciting stuff, but very often the same things are reinvented over and over again. So, before starting to solve a specific problem, try to find out if others have already solved it. Here is a list of things you can try:

- Try to find the key words that characterize your problem and then search the web. For example, if your problem involves a specific error message, try to search for the most specific part of this message.
- Use social networks like Twitter and search for your key words. When searching in social networks you often get very recent results. You might be surprised that you often get faster solutions compared with an Internet search.
- Try to find a newsgroup or mailing list that relates to your problem space and post your problem. But don’t just try asking questions – also reply to others!
- Don’t be shy or afraid. There are no stupid questions! And that your problem might be trivial for other experts in the field only helps you to get better. So don’t hesitate to share that you have a problem.
- Always react pleasantly, even if you get some less than pleasant responses. Even if most replies you get from others are nice, there will probably still some people who want to make you feel bad (or just want to make them feel better). If you get nasty replies, focus on the subject rather than emotions.

But you will probably find out that most other developers react nicely and are often very helpful. Actually many of them are excited that others are having similar problems. If you finally solve your specific problem, make the solution available to others. You could blog or tweet about your problem and your solution. You may be surprised how many positive replies you get. Some people might even improve your solution based on their own experience. But make sure to give credit to all those who helped you. Everybody likes that – you would like it, too! Also, if you have found similar solutions during your search, mention them.

Over time, you may save a lot of time and get better solutions for your problem much faster. A nice side effect is that you automatically connect with people who work on similar topics. It also helps you to increase your network of people with the same professional interests.

[64]http://programmer.97things.oreilly.com/wiki/index.php/Kai_T%C3%B6dter

---

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

[66]http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Reinvent_the_Wheel

Don’t Use too Much Magic

By Mario Fusco[67]

In recent years *convention over configuration* has been an emerging software design paradigm. By taking advantage of it, developers only need specify the non-common part of their application, because the tools that follow this approach provide meaningful behavior for all the aspects covered by the tool itself. These default behaviors often fit the most typical needs. The defaults can be replaced by custom implementations where something more specific is needed. Many modern frameworks, such as Ruby on Rails and Spring, Hibernate, and Maven in the Java world, use this approach in order to simplify developers’ lives by decreasing both the number of decisions they need to take and the amount of configuration they need to set up.

Most of the tools that take this approach generally do so by simplifying the most common and frequent tasks in the fastest and easiest way possible, yet without losing too much flexibility. It seems almost like magic that you can do so much by writing so little. Under the hood these frameworks do lots of useful things, like populate your objects from an underlying database, bind their property values to your favorite presentation layer, or wire them together via dependency injection. Moreover, smart programmers tend to add their own magic to the that of those frameworks, increasing the number of conventions that need to be respected in order to keep things working.

In a nutshell, convention over configuration is easy to use and can allow savings of time and effort by letting you focus on the real problems of your business domain without being distracted by the technical details. But when you abuse it – or, even worse, when you don’t have a clear idea of what happens under the hood – there is a chance that you lose control of your application because it becomes hard to find out at which point the tools you are using don’t behave as expected, or even to say which configuration you are running since you didn’t declare it anywhere. Then small changes in the code cause big effects in apparently unrelated parts of the application. Transactions get opened or closed unexpectedly. And so on.

This is the point where things start going wrong and programmers must call on their problem-solving skills. Using the fantastic features made available by a framework is straightforward for any average developer so long as the magic works, in the same way that a pilot can easily fly a large airplane in fine weather with the automatic pilot doing its job. But can the pilot handle that airplane in middle of a thunderstorm or when the wheels don’t come out during the landing phase?

A good programmer is used to relying on the libraries he uses as much as a good pilot is used to rely on his automatic counterpart. But both of them know when it is time to give up their automatic tools

[67]http://programmer.97things.oreilly.com/wiki/index.php/Mario_Fusco

and dirty their hands. An experienced developer is able to use the frameworks properly, but also to debug them when they don’t behave as expected, to work around their defects, and to understand how they work and what their limits are. An even better developer knows when it is not the case to use them at all because they don’t fit a particular need or they introduce an unaffordable inflexibility or loss of performance.

---

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

[69]http://programmer.97things.oreilly.com/wiki/index.php/Don%27t_Use_too_Much_Magic

Done Means Value

By Raphael Marvie[70]

The definition of *done* for a piece or software varies from one development team to another. It can have any one of the following definitions: "implemented," "implemented and tested," "implemented, tested, and approved," "shippable," or even something else. In *The Art of Agile Development*, James Shore defines *Done Done* as "A story is only complete when on-site customers can use it as they intended." But don't we forget something in those definitions? *Can be used* is different from *actually used.*

Why do we write software in the first place? There are plenty of reasons, varying from one person to another, and ranging from pure pleasure to simply earning money. But ultimately, in the end, isn't our main goal to deliver value to the end user? In addition, delivering value also brings pleasure and earnings.

Every artifact we can set up and use is, therefore, only a means to deliver value to users rather than a goal. Every action we take is, therefore, only a step in our journey to deliver value to users rather than an end. Tests – especially green ones – are a means to gain confidence. Continuous integration – especially when well tuned – is a means to be ready at any time. Regular deliveries – as often as possible – are a means to reduce the time to value for our users. But not one of them is an end in itself. Each is only a means to improve our ability to deliver value to our users.

Looking at software development from a Lean perspective, anything that does not bring value is waste. Even if the code is beautifully written. Even if all the tests pass. Even if the client has accepted the functionality. Even if the code is deployed. Even if the server is up and running. As long as it is not used, as long as it does not bring value to the user, it is waste. Our job is not done. We still have to find out why the software is not used. We must have missed something. Why is the user not satisfied? Perhaps something 'outside' of our process, like the absence of training, prevents our users from gaining value?

There are a lot of *Something*-Driven Development approaches. Most of the *somethings* are 'technical' in nature: Domain, Test, Behavior, Model, etc. Why don't we use Satisfaction-Driven Development? The satisfaction of the user. The satisfaction that arises as a consequence of the software delivering value to the user. Everything done is focused on the delivery of this value. Maybe not changing the world, but improving at least by a little the life of the user. Satisfaction-Driven Development is compatible with all of the other *Something*-Driven Development approaches and can be employed simultaneously.

[70]http://programmer.97things.oreilly.com/wiki/index.php/Raphael_Marvie

We should keep in mind that the meaning for writing software is broader than technical. We should always keep in mind that our goal is to deliver value to the user. And then our job will be done, and done well.

---

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

[72] http://programmer.97things.oreilly.com/wiki/index.php/Done_Means_Value

Execution Speed versus Maintenance Effort

By Paul Colin Gloster[73]

Most of the time spent in the execution of a program is in a small proportion of the code. An approach based on simplicity is suitable for the majority of a codebase. It is maintainable without adversely slowing down the application. Following Amdahl's Law, to make the program fast we should concentrate on those few lines which are run most of the time. Determine bottlenecks by empirically profiling representative runs instead of merely relying on algorithmic complexity theory or a hunch.

The need for speed can encourage the use of an unobvious algorithm. Typical dividers are slower than multipliers, so it is faster to multiply by the reciprocal of the divisor than to divide by it. Given typical hardware with no choice to use better hardware, division should (if efficiency is important) be performed by multiplication, even though the algorithmic complexity of division and multiplication are identical.

Other bottlenecks provide an incentive for several alternative algorithms to be used in the same application for the same problem (sometimes even for the same inputs!). Unfortunately, a practical demand for speed punishes having strictly one algorithm per problem. An example is supporting uniprocessor and multiprocessor modes. When sequential, quicksort is preferable to merge sort, but for concurrency, more research effort has been devoted to producing excellent merge sort and radix sorts than quicksort. You should be aware that the best uniprocessor algorithm is not necessarily the best multiprocessor algorithm. You should also be aware that algorithm choice is not merely a question of one uniprocessor architecture versus one multiprocessor architecture. For example, a primitive (and hence cheaper) embedded uniprocessor may lack a branch predictor so a radix sort algorithm may not be advantageous. Different kinds of multiprocessors exist. In 2009, the best published algorithm for multiplying typical m×n matrices (by P. D'Alberto and A. Nicolau) was designed for a small quantity of desktop multicore machines, whereas other algorithms are viable for machine clusters.

Changing the quantity or architecture of processors is not the only motivation for diverse algorithms. Special features of different inputs may be exploitable for a faster algorithm. E.g., a practical method for multiplying general square matrices would be $O(n{>}2.376)$ but the special case of (tri)diagonal matrices admits an $O(n)$ method.

[73] http://programmer.97things.oreilly.com/wiki/index.php/Paul_Colin_Gloster

Divide-and-conquer algorithms, such as quicksort, start out well but suffer from excessive subprogram call overhead when their recursive invocations inevitably reach small subproblems. It is faster to apply a cut-off problem size, at which point recursion is stopped. A nonrecursive algorithm can finish off the remaining work.

Some applications need a problem solved more than once for the same instance. Exploit dynamic programming instead of recomputing. Dynamic programming is suitable for chained matrix multiplication and optimizing searching binary trees. Unlike a web browser's cache, it guarantees correctness.

Given vertex coloring, sometimes graph coloring should be directly performed, sometimes clique partitioning should be performed instead. Determining the vertex-chromatic index is NP-hard. Check whether the graph has many edges. If so, get the graph's complement. Find a minimum clique partitioning of the complement. The algorithmic complexity is unchanged but the speed is improved. Graph coloring is applicable to networking and numerical differentiation.

Littering a codebase with unintelligible tricks throughout would be bad, as would letting the application run too slowly. Find the right balance for you. Consult books and papers on algorithms. Measure the performance.

---

[74]http://CreativeCommons.org/licenses/by/3.0/us/

[75]http://programmer.97things.oreilly.com/wiki/index.php/Execution_Speed_versus_Maintenance_Effort

Expect the Unexpected

By Pete Goodliffe[76]

They say that some people see the glass half full, some see it half empty. But most programmers don't see the glass at all; they write code that simply does not consider unusual situations. They are neither optimists nor pessimists. They are not even realists. They're *ignore-ists*.

When writing your code don't consider only the thread of execution you expect to happen. At every step consider all of the *unusual* things that might occur, no matter how *unlikely* you think they'll be.

Errors

Any function you call may not work as you expect.

- If you are lucky, it will return an error code to signal this. If so, you should check that value; never ignore it.
- The function might throw an exception if it cannot honor its contract. Ensure that your code will cope with an exception bubbling up through it. Whether you catch the exception and handle it, or allow it to pass further up the call stack, ensure your code is correct. Correctness includes not leaking resources or leaving the program in an invalid state.
- Or the function might return no indication of failure, but silently not do what you expected. You ask a function to print a message: Will it always print it? Might it sometimes fail and consume the message?

Always consider errors that you can recover from, and write recovery code. Consider also the errors that you cannot recover from. Write your code to do the best thing possible – don't just ignore it.

[76] http://programmer.97things.oreilly.com/wiki/index.php/Pete_Goodliffe

Threading

The world has moved from single-threaded applications to more complex, often highly threaded, environments. Unusual interactions between pieces of code are staple here. It's hard to enumerate every possible interweaving of code paths, let alone reproduce one particular problematic interaction more than once.

To tame this level of unpredictability, make sure you understand basic concurrency principles, and how to decouple threads so they cannot interact in dangerous ways. Understand mechanisms to reliably and quickly pass messages between thread contexts without introducing race conditions or blocking the threads unnecessarily.

Shutdown

We plan how to construct a system: How to create all the objects, how to get all the plates to spin, and how to keep those objects running and those plates spinning. Less attention is given to the other end of the lifecycle: How to bring the code to a graceful halt without leaking resources, locking up, or crashing.

Shutting down your system and destroying all the objects is especially hard in a multi-threaded system. As your application shuts down and destroys its worker objects, make sure you can't leave one object attempting to use another that has already been disposed of. Don't enqueue threaded callbacks that target objects already discarded by other threads.

The Moral of the Story

The unexpected is not the unusual. You need to write your code in the light of this.

It's important to think about these issues early on in your code development. You can't tack this kind of correctness as an afterthought; the problems are insidious and run deeply into the grain of your code. Such demons are very hard to exorcise after the code has been fleshed out.

Writing good code is not about being an optimist or a pessimist. It's not about how much water is in the glass right now. It's about making a watertight glass so that there will be no spillages, no matter how much water the glass contains.

---

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

[78]http://programmer.97things.oreilly.com/wiki/index.php/Expect_the_Unexpected

First Write, Second Copy, Third Refactor

By Mario Fusco[79]

It is difficult to find the perfect balance between code complexity and its reusability. Both under- and overengineering are always around the corner, but there are some symptoms that could help you to recognize them. The first one is often revealed by excessive code duplication, while the second one is more subtle: Too many abstract classes, overly deep classes hierarchies, unused hook methods, and even interfaces implemented by only one class – when they are not used for some good reason, such as encapsulating external dependencies – can all be signs of overengineering.

It is said that late design can be difficult, error-prone, and time consuming, and the complete lack of it leads to messy and unreusable code. On the other hand, early engineering can introduce both under- and overengineering. Up-front engineering makes sense when all the details of the problem under investigation are well defined and stable, or when you think to have a good reason to enforce a given design. The first condition, however, happens quite rarely, while the second one has the disadvantage of confining your future possibilities to a predetermined solution, often preventing you from discovering a better one.

When you are working on a problem for the very first time, it is a difficult – and perhaps even useless – exercise to try to imagine which part of it could be generalized in order to allow better reuse and which not. Doing it too early, there is a good chance that you are jumping the gun by introducing unnecessary complexity where nobody will take advantage of it, yet at the same time failing to make it flexible and extensible at the points where it should be really useful. Moreover, there is the possibility that you won't need that algorithm anywhere else, so why waste your efforts to make reusable something that won't be reused?

So the first time you are implementing something new, write it in the most readable, plain, and effective way. It is definitely too early to put the general part of your algorithm in an abstract class and move its specialization to the concrete one or to employ any other generalization pattern you can find in your experience-filled programmer's toolbox. That is for a very simple reason: You do not yet have a clear idea of the boundaries that divide the general part from the specialized one.

The second time you face a problem that resembles the one you solved before, the temptation to refactor that first implementation in order to accommodate both these needs is even stronger. But

[79]http://programmer.97things.oreilly.com/wiki/index.php/Mario_Fusco

it may still be too early. It may be a better idea to resist that temptation and do the quickest, safest, and easiest thing it comes you in mind: Copy your first implementation, being sure to note the duplication in a *TO DO* comment, and rewrite the parts that need to be changed.

When you need that solution for the third time, even if to satisfy a slightly different requirement, the time is right to put your brain to work and look for a general solution that elegantly solves all your three problems. Now you are using that algorithm in three different places and for three different purposes, so you can easily isolate its core and make it usable for all three cases &mdashl and probably for many subsequent ones. And, of course, you can safely refactor the first two implementations because you have the unit tests that can prove that you are not breaking them, don't you?

---

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

[81]http://programmer.97things.oreilly.com/wiki/index.php/First_Write%2C_Second_Copy%2C_Third_Refactor

From Requirements to Tables to Code and Tests

By George Brooke[82]

It is generally reckoned that the process of getting from requirements to implementation is error prone and expensive. Many reasons are given for this: a lack of clarity on the part of the user; incomplete or inadequate requirements; misunderstandings on the part of the developer; catch-all *else* statements; and so on. We then test like crazy to show that what we have done at least satisfies the law of least surprise! I want to focus here on business logic. It is the execution of the business logic which delivers the value of an application.

Let's abstract business logic a little. We can imagine that for some circumstance – say, evaluation of a client or a risk, or simply the next step in a process – there are some criteria that, when satisfied, determine one or more actions to be performed. It would be convenient to capture the requirements in this form: a list of the criteria, *C1*, *C2*, …, *Cn* and a list of corresponding actions. Then for each combination of criteria we have a selection of actions, *Am*. If each of the criteria is simply a condition that evaluates to a Boolean, we know that there are possibly 2n possible criteria combinations to be addressed. Given such a requirements model we can *prove* completeness!

Assuming that the requirements are captured in this form, then we have an opportunity to (largely) automate the code generation process. In doing this we have massively reduced the gap between the business community and the developer. Some readers may recognise what I am describing: decision tables, which have been around for at least 40 years! Although there are processors around for decision tables, they are not necessary to get many of the benefits. For example, the enumeration of the criteria and the actions is a significant step in abstraction and understanding of the problem. It also provides the developer with the opportunity to analyse and detect logical nonsense in the requirements because of the formalism of the decision table representation. Feeding this back to the user, we can overcome some of the tension in the relationship.

For implementation, one choice would be to write (or generate) conventional *if-then-else* logic. We could also use the combination of criteria to index into an in-memory table representation of the decision table. This has the advantage of preserving the clear relationship between the decision table used for requirements specification and its implementation, still reducing the gap between specification and implementation even more. Of course, there is nothing that says the table need be in memory – we could store it in a relational database, and then access the appropriate table and

[82] http://programmer.97things.oreilly.com/wiki/index.php/George_Brooke

index directly to the set of actions. The general idea is to try to capture logic in a tabular form, and interpret it at runtime. You can also have runtime modification of table content.

And so to testing. If the code generation process from the table has been largely automated, there is little need to path test all the possible paths through the table. The code is the table; the table is the code. Certain tests need to be done, but these are more along the lines of configuration management and reality checking than path testing. The total test effort is significantly reduced because we have moved the work into unambiguous requirements specification.

Implementations of this type of system run from the humble programmer using the technique as a private coding method to full runtime environments with natural language translation to generate rule tables for interpretation.

---

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

[84]http://programmer.97things.oreilly.com/wiki/index.php/From_Requirements_to_Tables_to_Code_and_Tests

How to Access Patterns

By Klaus Marquardt[85]

Patterns document knowledge that many developers and projects share. If you know a fair number of them, you will be able to find better solutions faster.

To support such a strong claim, take a look at the typical contents of a pattern (each book varies in its presentation, but these elements will be present):

1. A name
2. A problem to be solved
3. A solution
4. Some rationale for the appropriateness of the solution
5. Some analysis of the applied solution

The strength of a pattern is not just the solution. It also provides insights on why this is a good solution. And it gives you information not only on the benefits, but also on possible liabilities. A pattern allows you and your peers to make well-informed decisions. It can make all the difference between "this is how it is done" (a stale attitude that prevents new ideas and change) and "this is how and why we do this here" (an attitude that is aware of a world outside of the acquired habits and open to evaluation and learning).

Patterns are useful; the hard part is using them. A problem will come up all of a sudden, and your peers will not wait for someone to claim "I'll find a pattern for this, give me a week to search." You need to have pattern knowledge before you actively consider using one. This is the hard part: Not only does it require lots of work in advance (and patterns often are not the most enjoyable pieces of literature), it also contradicts how most humans learn – by applying some solution and observing what happens. At the very least, we need some linkage from the stuff we read to some experience already present in our brain.

This is why I use and suggest a three-pass reading style for patterns.

1. The name and the first sentence of the solution.
2. The problem, the solution, the key consequences and a short example.

[85] http://programmer.97things.oreilly.com/wiki/index.php/Klaus_Marquardt

3. Everything, including implementation aspects and examples.

The first pass I'd do with every pattern I come across. It takes only a few seconds, but you get an impression of what this pattern is all about. Your impression may be off target, but that is OK for now. When some discussion pops up in your project, you will remember such a pattern and you'll be ready for the second pass. Isn't it a bit late by then? Not necessarily. Before a problem becomes urgent, most projects have some warning time. If you are aware of what is going on around your desk and task, you will be ahead just a bit – and this is sufficient.

The second pass is meant to give you all the ammunition you need in the heat of a design discussion. You will be able to judge proposals, and to propose some pattern yourself – or not to propose it (remember, applying a particular pattern is not unconditionally good). Both will take the team forward. Be careful with the pattern name in this phase. Depending on the team background, the name may not ring any bells. In such cases it is better to just explain the idea of the solution than to focus on its name.

Hold back the third pass for when you have decided on some pattern and you need to implement it. If you are like me, you probably don't want to read 30+ pages of some pattern before you are certain it's relevant to you. And you come to appreciate patterns whose authors knew how to apply the quality of brevity.

The first pass involves a paragraph at most and takes a minute. The second pass requires understanding about two pages and probably around 15 minutes. The third pass requires hours, but this is productive work time. With these reading styles you will be able to make the most of patterns.

---

[86]http://creativecommons.org/licenses/by/3.0/us/
[87]http://programmer.97things.oreilly.com/wiki/index.php/How_to_Access_Patterns

Implicit Dependencies Are also Dependencies

By Klaus Marquardt[88]

Once upon a time a project was developed in two countries. It was a large project with functionality spread across different computers. Each development site became responsible for the software running on one computer, had to fulfill its share of requirements and do its share of testing. The specification of the communication protocol became an early architectural cornerstone.

A few months later, each site declared victory: The software was finished! The integration team took over and plugged everything together. It seemed to work. A bit. Not much though: As soon as the most common scenarios were covered and the more interesting scenarios were tested, the interaction between the computers became unreliable.

Confronted with this finding, both teams held up the interface specification and claimed their software conformed to it. This was found to be true. Both sides declared victory, again. No code was changed, and they developed happily ever after.

The moral is that you have more dependencies than all your attempts for decoupling will let you assume you have.

Software components have dependencies, more so in large projects, even more when you strive to increase your code reuse. But that doesn't mean you have to like them: Dependencies make it hard to change code. Whenever you want to change code others depend on, you will encounter discussion and extra work, and resistance from other developers who would have to invest their time. The counterforces can become especially strong in environments with a lengthy development micro cycle, such as C++ projects or in embedded systems.

Many technical approaches have been adopted to reduce suffering from dependencies. On a detailed level, parameters are passed in a string format, keeping the interface technically unchanged, even though the interpretation of the string's contents changes. Some shift in meaning could be expressed in documentation only; technically the client's software update could happen asynchronously. At a larger scale, component communication replaces direct interface calls by a more anonymous bus where you do not need to contact your service yourself. It just needs to be out there somewhere.

These techniques actually make it harder to spot the underlying implicit dependencies. Let's rephrase the moral a bit: Obfuscated dependencies are still dependencies.

[88]http://programmer.97things.oreilly.com/wiki/index.php/Klaus_Marquardt

Source-level or binary independence does not relieve you or your team from dependency management. Changing an interface parameter's meaning is the same as changing the interface. You may have removed a technical step such as compilation, but you have not removed the need for redeployment. Plus, you've added opportunities for confusion that will boomerang during development, test, integration, and in the field – returning when you least expect it.

Looking at sound advice from software experts, you hear Fred Brooks talking you into conceptual integrity, Kent Beck urging *once and only once*, and the Pragmatic Programmers advising you to keep it DRY (Don't Repeat Yourself). While these concepts increase the clarity of your code and work against obfuscation, they also increase your technical dependencies – those that you want to keep low.

The moral is really about: Application dependencies are the dependencies that matter.

Regardless of all technical approaches, consider all parts of your software as dependent that you need to touch synchronously, in order to make the system run correctly. Architectural techniques to separate your concerns, all technical dependency management will not give you the whole picture. The implicit application dependencies are what you need to get right to make your software work.

---

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

[90]http://programmer.97things.oreilly.com/wiki/index.php/Implicit_Dependencies_Are_also_Dependencies

Improved Testability Leads to Better Design

By George Brooke[91]

Economic Testability is a simple concept yet one seen infrequently in practice. Essentially, it boils down to recognizing that since code-testing should be a requirement and that we have in place some nice, economical, standard tools for testing (such as xUnit, Fit, etc.), then the products that we build should always satisfy the new requirement of ease-of-test, in addition to any other requirements. Happily the ease-of-test requirement reinforces rather than contradicts best practice.

If you build your systems so that testing is made economic – while simultaneously of course preserving simplicity in the production model – the interfaces that you finally end up with are likely to be greatly improved over the one-environment system. Code which has to operate successfully and unchanged in two or more environments (the test and production environments) must pay more than lip service to clean interfaces and maximum encapsulation if the task of embedding within the multiple environments is not to become overwhelming. The discipline needed leads to better design and more modular construction. It really is a win-win situation.

Progressive testing is often stymied because certain necessary functions have not yet been implemented. For example, a common occurrence involves an object that needs to be tested, but makes use of a yet-to-be-built object. An incompleteness that means the test cannot be run. A way around this is to allow the provision of the yet-to-be-built object via a parametrized constructor: When testing we can provide a test double object without changing the internals at all; in the production code we can provide the real (tested) object to deliver the required functionality. The API of the test double and the production object are identical and the object under test is unaware of whether it is running in a test or a production environment.

We can go further of course – a lot further. This very soft style of building systems brings advantages all down the line when compared to the hard-wired logic commonly encountered in code. For example, if we need to introduce some kind of logging trail for complex bug diagnosis during development, we can provide the logging logic via the constructor parameters, using the plug-in technique already outlined. In production, using the *null object* pattern (which will provide a “do nothing” object with the same API as the logger), we can replace the logger with a harmless null alternative. This means of course that the code being monitored is unchanged, an important point for some types of bug. Should it be necessary that you dynamically enable logging in production

[91]http://programmer.97things.oreilly.com/wiki/index.php/George_Brooke

mode, this technique makes it very simple to enable or disable logging dynamically – or indeed anything else that you need. Then we have a production system that gracefully slips into logging mode when needed. It may be unfortunate that you should need to do this, but if you do, then sensible application of these techniques can be seriously reputation enhancing!

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

[93]http://programmer.97things.oreilly.com/wiki/index.php/Improved_Testability_Leads_to_Better_Design

In the End, It’s All Communication

By Thomas Lundström[94]

Programming is often looked upon as a solitary and uncommunicative craft. In truth, it is the exact opposite.

Programming. That’s you trying to communicate with the machine, telling it what to do. The machine will always do what you tell it to do, but not necessarily what you want it to do. There is huge potential for miscommunication.

Programming. That’s you trying to communicate with other members of your team. You and your peers need to decide how your system will work, fleshing out different modules of your system and keeping them in sync. Failure to do so will inevitably lead to your system malfunctioning. Therefore, make sure the communication between the members of the team as high bandwidth as possible, favoring face-to-face conversation over email. Where possible, avoid remote working and have the entire team seated together.

Programming. That’s you trying to communicate with other stakeholders of the project. It might be the IT department that will be in charge of the production environment. It might be the end users, providing you with information on how they actually use your system. It might be the decision maker who wants the system to carry out a specific part of their business process. Failure to communicate will inevitably make sure your system is unusable, unfit for production, or simply not the right tool for the job. Therefore:

- Even though you are a team, don’t shut the other stakeholders of the product out.
- Work with usability as early as possible.
- Practice putting the system into production.
- Develop a common language for communicating the properties of your product with the stakeholders (in Domain-Driven Design this is referred to as the *ubiquitous language*).

Programming. That’s you actually communicating with the programmers that read your code long after you’ve written it. If you fail to communicate the intent of your module, in time someone may misunderstand you. If that person writes code based upon the misunderstanding there will be a defect. Therefore:

[94]http://programmer.97things.oreilly.com/wiki/index.php/Thomas_Lundstr%C3%B6m

- Write your modules to have high cohesion and low coupling.
- Document your intent by adding unit tests.
- Make sure your code uses the ubiquitous language of your problem domain.

Thus, in order to be a successful programmer, you need to be a great communicator. Don’t be a mole, only poking your head above ground every few weeks with a new piece of functionality. Instead, get out there. Talk to users. Show your results as often as possible to the stakeholders. Employ daily stand-ups with your fellow team members.

Last but not least, seek out your peers and get involved in a user group. That way you’re constantly improving your communication skills.

---

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

[96]http://programmer.97things.oreilly.com/wiki/index.php/In_the_End%2C_It%27s_All_Communication

Integrate Early and Often

by Gerard Meszaros[97]

When you are working as part of a software development team, the software you write will invariably need to interact with software written by other team members. There may be a temptation for everyone to agree on the interfaces between your components and then go off and code independently for weeks or even months before integrating your code shortly before it is time to test the functionality. Resist this temptation! Integrate your code with the other parts of the system as early as possible – or maybe even earlier! Then integrate your changes to it as frequently as possible.

Why is early and frequent integration so important? Code that is written in isolation is full of assumptions about the other software with which it will be integrated. Remember the old adage, "Don't ASSUME anything because you'll make an ASS out of U and ME!" Each assumption is a potential issue that will only be discovered when the software is integrated. Leaving integration to the last minute means you'll have very little time to change how you do things if it turns out your assumptions are wrong. It's like leaving studying until the night before the final exam. Sure, you can cram, but you likely won't do a very good job.

You can integrate your code with components that don't exist yet by using a Test Double such as a Test Stub, a Fake Object, or a Mock Object. When the real object becomes available, integrate with it and add a few more tests with the real McCoy. Another benefit of frequent integration is that your change set is much smaller. This means that when you start integration of your changes the chances of having changed the same method or function as someone else is much smaller. This means you won't have to reapply your changes on top of someone else's just because they beat you to the check-in. And it reduces the likelihood of anyone's changes being lost.

High-performing development teams practice continuous integration. They break their work into small tasks – as small as a couple of hours – and integrate their code as soon as the task is done. How do they know it's done? They write automated unit tests before they write their code so they know what *done* looks like. When all the tests pass, they check in their changes (including the tests). Then, while they have a green bar (all tests passing) they refactor their code to make it as clean and simple as possible. When they are happy with the code, and all the tests pass, they check it in again. That's two integrations in one paragraph!

There are many tools available to support Continuous Integration (or CI, as it is also known). These tools automatically grab the latest version of the code after every check-in, rebuild the system to

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

make sure it compiles, and run all the automated tests to make sure it still works. If anything goes wrong, the whole team is informed so they can stop working on their individual task and fix the broken build. In practice, it doesn't get broken very often because everyone can run all the tests before they check in. Just another benefit of integrating early and often.

---

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

[99]http://programmer.97things.oreilly.com/wiki/index.php/Integrate_Early_and_Often

Interfaces Should Reveal Intention

By Einar Landre[100]

Kristen Nygaard, father of object-oriented programming and the Simula programming language, focused in his lectures on how to use objects to model the behavior of the real world and on how objects interacted to get a piece of work done. His favorite example was Cafe Objecta, where waiter objects served the appetites of hungry customer objects by allocating seating at table objects, providing menu objects, and receiving order objects.

In this type of model we will find a restaurant object with a public interface offering methods such as `reserveTable(numberOfSeats,customer,timePoint)` and `availableTables(numberOfSeats,timePoint)`, and waiter objects with methods such as `serveTable(table)` and `provideMenu(customer,table)` – object interfaces that reveal each object's intent and responsibility in terms of the domain at hand.

So, where are the *setters* and *getters* so often found dominating our object models? They are not here as they do not add value to the behavioral intention and expression of object responsibility.

Some might then argue that we need setters to support *dependency injection* (a.k.a. *inversion of control* design principle). Dependency injection has benefits as it reduces coupling and simplifies unit testing so that an object can be tested using a mock-up of a dependency. At the code level this means that for a restaurant object that contains table objects, code such as `Table table = new TableImpl(...);` can be replaced with `Table table;` and then initialized from the outside at runtime by calling `resturant.setTable(new TableImpl());`

The answer to that is that you do not necessarily need *setters* for that. Either you use the constructor or, even better, create an interface in an appropriate package called something like `ExternalInjections` with methods prefixed with `initializeAttributeName(AttributeType)`. Again the intention of the interface has been made clear by being public and separate. An interface designed to support the use of a specific design principle or the intent of frameworks such as Spring.

So what about the *getters*? I think you are better off just referring to queried attributes by their name, using methods named `price`, `name`, and `timePoint`. Methods that are pure queries returning values are, by definition, functions and read better if they are direct: `item.price()` reads better than `item.getPrice()` because it make the concepts of the domain stand out clearly following the principles found in natural language.

The conclusion on this is that setters and getters are alien constructs that do not reveal the intention and responsibility of a behavior-centric interface. Therefore you should try to avoid using them; there are better alternatives.

[100] http://programmer.97things.oreilly.com/wiki/index.php/Einar_Landre

---

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

[102]http://programmer.97things.oreilly.com/wiki/index.php/Interfaces_Should_Reveal_Intention

Isolate to Eliminate

By Stuart Herbert[103]

Whether you're looking at your own code before (or after!) you have shipped it, or you're picking up someone else's code after they have shipped it, tracking down and fixing bugs is a fundamental part of programming. If you know the code well, perhaps you can make an intuitive leap to jump immediately to where the bug is. But how do you go about tracking down a bug when intuition doesn't help?

The nature of all code is that larger systems are built from smaller underlying systems and components. They in turn are also built from smaller systems and components. The bug you are tracking down will have a cause in one of these, and will have symptoms that are visible in other systems. The remaining systems work fine, as far as the bug you're looking for is concerned, so you can use this knowledge to quickly and reliably find where the bug is.

Divide your larger systems down into smaller systems at logical points, such as different server stacks, APIs, major interfaces, classes, methods, and, if necessary, individual lines of code. Test both sides of the divide, with your tests focusing on the data that crosses the divide. If one side works as expected, the bug is not in there. You can eliminate that side from further testing. Continue testing the remaining systems and components, which you have now isolated, by dividing those into smaller systems and components. Keep going until you've reached the smallest testable system, component, unit, or fragment of code that exhibits the bug. Congratulations: You have isolated the fault.

Apart from being a strategy that allows you to work on code you've never seen before, this approach also has the advantage that it is evidence-based. It approach eliminates guesswork and forces developers' assumptions about how their code actually works to be challenged. The data never lies, but be aware that it can be misinterpreted!

The approach is inherently iterative. You'll often go back and forth between your code and your tests, making your code easier to test and your tests clearer, with more targeted test domains and results. Fix the tests that are relevant to the bug you are tracking down, but make a list of any other issues you find along the way so you can come back and address them at a later date. Stay on target, and park potential tangents and distractions for another time.

Being able to debug code is the single most important skill any programmer needs to master. Despite all the headlines you'll read on the Internet of programmers making it big, they are a tiny minority. Most programmers will spend the majority of their careers maintaining code that they have inherited

[103] http://programmer.97things.oreilly.com/wiki/index.php/Stuart_Herbert

from someone else. That's the reality of being a professional programmer, and strategies for taking code apart and solving problems in it will be the tools that you use the most.

---

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

[105]http://programmer.97things.oreilly.com/wiki/index.php/Isolate_to_Eliminate

Keep Your Architect Busy

By Klaus Marquardt[106]

Architect is probably the most prestigious technical job available in software development. Unsurprisingly, most developers have mixed feelings about the project's architect. Part of this mix is that you might want to become one yourself; another part is that competent and dominant people often appear arrogant and a threat to other people's self-confidence.

While you might be tempted to avoid any contact: be aware that this is your decision and not his. There are constructive ways to benefit from an architect's presence, both for your personal benefit and your project's progress. Ultimately, architect is a supportive role. Make the architect work for you. Ask him for problem resolution, remind him of his responsibilities in the ongoing project. This way he will not be occupied with some vague future project, or become ignorant of actual problems during implementation.

First of all, someone who does the decomposition of a system should also be able to take responsibility for its recomposition. This means that the architect who structured the system should also be the key integrator who makes sure that the developed pieces fit together and can be made work. Such a reciprocal definition of responsibilities will let every architect be careful and interested in the ongoing development. An architecture may even put his main focus on the final integration, a model I like to call integration-driven architecture.

Second, real projects can face architectural problems at any time. It is a myth that the "architectural" issues are all addressed at the beginning of a project. Each project learns many things during implementation that turn out to be relevant for the overall project success, some of them might flatly contradict what the architect stated months ago. (Note that this is neither the developer's nor the architect's fault.) Experienced architects explicitly leave some issues for resolution until such a time as more knowledge becomes available. Make your life easier: When you need a decision, invite the architect to solve the architectural issues.

Last, but not least, frequent contact with an architect is a great learning opportunity for you. The decisions you demand will likely benefit from your own ideas and proposals. You will receive feedback on your work, widen your horizon, and increase your career options. And, if your architect knows about the art of deciding no earlier than necessary, you can gain invaluable insights about what makes projects successful. And he will likely be thankful for your responsiveness – an assumed ivory tower is a place without much opportunity for feedback, in either direction.

[106]http://programmer.97things.oreilly.com/wiki/index.php/Klaus_Marquardt

---

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

[108]http://programmer.97things.oreilly.com/wiki/index.php/Keep_Your_Architect_Busy

Know When to Fail

By Geir Hedemark[109]

Almost all applications have some external resources they cannot do without. For instance, a server-side application that handles a membership list is probably not all that useful without access to wherever the list of members is stored.

When critical resources are not available on startup, the application should print an error message and stop immediately. If the application continues running, there will probably be a very long list of related errors in the log, which will in turn confuse any debugging attempts.

A while back, a consultancy in Norway won a contract for an application worth $500,000. When the customer became dissatisfied and threatened legal action, some more senior developers were told to have a look. They found that most of the customer frustration was caused by a mangled installation, and 80% of the application was missing any attempt at error handling. It was almost impossible to get the application operational without access to the source code because all exceptions were thrown away. After the two most pressing issues were fixed, the customer relationship was saved.

The usual response to complaints about hard-to-track errors is "we will fix them in the next version." Only they won't be fixed. The team will be too busy trying to sort out all of the misleading error reports from whoever tried to make the application work. The boss will notice, and hire more developers. These developers will try to install the application. There are now two plates to be kept spinning and the new developers will probably tell the boss exactly what they think. There is now even less time to formulate what the application needs of its environment. So it won't happen.

The first step towards getting error handling right is to stop the application when its environment is broken. Start agreeing on the environment spec in the first iteration, and make sure that failure to comply with the environment during startup leads to immediate and sudden failure of the application with a sensible error message. The application will now interact with whoever installs it in a nicer way, and there will be some kind of error handling structure to extend further down the line. This initial platform is crucial when the team tries to build some kind of quality around the behavior of the application later on.

The boss will probably not notice the absence of a wave of errors every time someone tries to install the application. The team will. They will be free to create more value instead of mopping up error reports.

Every team should know when to fail.

[109] http://programmer.97things.oreilly.com/wiki/index.php/Geir_Hedemark

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

[111]http://programmer.97things.oreilly.com/wiki/index.php/Know_When_to_Fail

Know Your Language

by Bob Archer[112]

Syntax and semantics are important, but they're just a starting point. There is much more to know if you want to use a language effectively: How to write short, understandable code that works and is likely to continue working and be able to read, understand, and incorporate third-party code.

Know the idioms that are specific to the language you're working in. For example, in C++ you can make an object uncopyable by making the copy constructor and assignment operator private. Also know the common ways of implementing cross-language idioms – the patterns in *Design Patterns* are a good start.

Know the history of the language. If your language was based on another language look at what changed. If your language has gone through multiple revisions look at what has changed (and why) between the revisions.

Know the future of the language. How is the language likely to change (or not)? Know which features are headed for deprecation and which features are likely to be added. Know how the language handles moving to a new, possibly incompatible version (e.g., Python 3.0). Know what the process is for changing the language (e.g., Python PEPs).

Know which features of the language have counterparts in other languages and which are unique to this language. Know which of the unique features are useful enough that they might make it into future languages.

Know what's wrong with the language. Read other people's critiques of the language; write your own critiques of the language. Understand why some of the flaws exist. Sometimes the reasons are historical. Sometimes the flaws are genuinely unavoidable or are the lesser of two evils. Sometimes the flaws are just mistakes. One interesting exercise is to take a flaw and work out how it could be corrected, and whether the knock-on effects would be worse than the flaw itself.

Know how you work around features that aren't in the language. For example, if the language does not have garbage collection, know what techniques are used to keep memory under control. Conversely, if the language does have garbage collection, learn the details of exactly how it works so that you're not surprised by odd performance and behavior.

Know the libraries that come with the language, and the common third-party libraries that are available (e.g., Boost and wxPython).

[112]http://programmer.97things.oreilly.com/wiki/index.php/Bob_Archer

Know what happens when something goes wrong. Know what errors the compiler gives for common mistakes. Know what happens when there's an error at runtime, and what you can do about it.

Know what mistakes are particularly prevalent in the language. Try and avoid them yourself and be aware that they might crop up in code from other programmers.

Know what the definitive guides to your language are. Some authors are more reliable than others. Some compilers stick more closely to the language specification than others do. If there is an official standard for your language, make sure you have a copy of it and use it. The standard may appear to be written in a foreign tongue, but with practice it's quite possible to understand it.

Finally, although knowledge is important, it is a means, not the end. As Goethe said "Knowing is not enough; we must apply." Take that knowledge, and use it to produce great software.

---

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

[114]http://programmer.97things.oreilly.com/wiki/index.php/Know_Your_Language

Learn the Platform

By Vatsal Avasthi[115]

In these times of *write once, run anywhere* languages, web frameworks, and virtualization, is the operating system gradually losing its importance for application developers? The answer is *no*. Applications that are easier to develop because of these abstractions may be harder to debug, fix, and optimize because of the very same abstractions.

The operating system is the platform on which you both develop and run your application. Knowing the platform makes you much more effective, whether designing, diagnosing, or optimizing your software. Your application gets developed on a platform, runs on a platform, and halts, hangs, and crashes on a platform, so learning the platform is a key skill in being a good programmer.

Learning the platform includes but does not limit you to knowing the tools present on the platform for development (IDE, build tools, etc.). It also means being familiar with other aspects of the operating system, not all of which may be directly related to development. Familiarity implies being able to answer the following questions:

- What is the underlying architecture of the operating system?
- How does the operating system manage memory?
- How are threads managed in the operating system? What is their life cycle?
- How does the file system work? What are the key file system abstractions? How are files organized?
- What utilities are available that tell you the state of the system and help you see what is happening under the hood?

If you are on Unix or one of its variants, such as Linux, one alternative approach to explore and learn more about the operating system is to install the server programs that come with the distributions and try to get these servers, daemons, and services configured and running. This may include daemons and servers providing remote shells (e.g., telnet, SSH), file transfer services (e.g., FTP), file and print services (e.g., Samba), web, mail, and so on.

Thinking that these tools and this knowledge is the preserve of system administrators, and therefore a no-no for programmers, may hurt you unexpectedly when your application is not the bottleneck or the root of the problem you are trying to solve. You need to get under the hood of the operating

[115]http://programmer.97things.oreilly.com/wiki/index.php/Vatsal_Avasthi

system and learn about common protocols like SMTP, HTTP, and FTP, client-server architecture, characteristics of daemons and services, and how to interrogate running processes and diagnose the state of the system. Most of the servers and services mentioned here enable interoperability with other systems in a standardized way, so they tell programmers how to interact with other systems and how a protocol can be used and incorporated in their own applications.

Today, applications are a complex tower of abstractions, so knowing your application is not enough. You also need to know what runs underneath it.

---

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

[117] http://programmer.97things.oreilly.com/wiki/index.php/Learn_the_Platform

Learn to Use a Real Editor

By Diomidis Spinellis[118]

I'm sorry to break the news to you, but Windows *Notepad* and, probably, the editor that comes with your IDE are toys. As a professional programmer invest the effort needed to use a real editor effectively. At the risk of starting a religious war let me point you toward *vim*, the modern supercharged incarnation of the Unix *vi* editor, and *Emacs*, the editor that some compare to an operating system.

Whatever your choice these are examples of tasks you should learn doing in your editing environment.

- Change something on lines matching (or not matching) a specific pattern. Or delete those lines. For instance, delete all empty lines.
- Convert a series of method calls into initialization data.
- Convert data from an HTML table into a series of SQL insert commands.
- Visit one file, copy various useful things into a few separate buffers, and then visit another to place them where needed. This is useful for copying separate interrelated parts of an APIs invocation sequence.
- Accumulate material from various places into a buffer for pasting in another place.
- Edit and re-execute a complex command you entered a few hours ago. Or repeat a complex sequence of commands in various parts of the file you're editing.
- Gather a sequence of named HTML section headings and convert them into a table of contents.
- Change assignments into method invocations.

For many of the above tasks, you need to master the powerful but slightly cryptic language of regular expressions. These are simply a way to express a recipe for a string your editor must match. Here is a cheat sheet with the most common special characters.

. Match any character

* Match the preceding expression any number of times

ˆ Match the beginning of the line

$ Match the end of a line

[118]http://programmer.97things.oreilly.com/wiki/index.php/Diomidis_Spinellis

[a-z_] Match the characters between the brackets (a lowercase letter and the underscore in this case)

[ˆ0-9] Match any but the characters between the brackets (any non-digit character in this example)

\< Match the beginning of a word

\> Match the end of a word

\ Match the following special character literally

The search-and-replace command of any editor worth its salt will allow you to bracket parts of a regular expression and reuse those parts in the replacement string. This by far the most powerful way to construct sophisticated editing commands.

You also want your editor to be running on all the systems you're using, to allow you to touch-type commands on the most common keyboard layouts you use (*vim* does a pretty good job on this front), to be scriptable using a rich language (*Emacs* is famous for this), to compile your project and guide you through its errors, to provide syntax highlighting, and, of course, to prepare a decent latte.

---

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

[120]http://programmer.97things.oreilly.com/wiki/index.php/Learn_to_Use_a_Real_Editor

Leave It in a Better State

By Patrick Kua[121]

Imagine yourself sitting down to make a change to the system. You open a particular file. You scroll down… suddenly you recoil in horror. Who wrote this horrible code? What poor programmer had their way with this? If you've been working as a professional programmer for a reasonable length of time, I'm sure you've been in this situation many times.

There is a more interesting question to ask. How did the process let the code end up in such a mess? After all, far too frequently, messy code is one of those things that emerges over time. There are, of course, many explanations, and many war stories that people share. Perhaps it was the new developer. Asked to make a change, they didn't have enough understanding of how the code was supposed to work, so they worked around it instead of investing time to truly understand what it did. Perhaps someone had to make a quick fix to meet a deadline and never returned to it afterwards – moving on, perhaps, to make the next mess. Perhaps a group of developers worked in the same area of code without ever sitting down together to establish a shared view of the design. Instead, they tiptoed around each other to avoid any arguments. Most development organizations don't help by adding additional pressure to churn out new features without emphasizing inward quality.

There are many reasons why mess gets created. But somehow developers use these same reasons to justify not cleaning up anything when they stumble across someone else's mess. Easy? Yes. Professional? No.

It's rare that a horrible codebase happens overnight (those are probably those demo systems thrown into production). Instead, our industry is addled by systems slowly brought to their knees by programmers who leave the code without any improvements, often making it just that little bit messier over time. Each small workaround adds another layer of unnecessary complexity, with the combined effect of quickly escalating a slightly complex system into the monstrous unmaintainable beasts we hear about all the time.

What can we do about it?

Businesses often refuse to set aside any time for programmers to clean up code. They often have a hard time understanding how nonessential complexity crept in because, rightly so, they view it as something that should not have happened.

The only real cure for a codebase suffering from this debilitating and incremental condition is to take a vow to "Leave It in a Better State." In the same way that small detrimental changes coalesce into a

[121]http://programmer.97things.oreilly.com/wiki/index.php/Patrick_Kua

big mess, small constructive changes converge to a much simpler system. It’s helpful to realize that small improvements do not need to consume large amounts of time. Small improvements might include simply renaming a variable to a much clearer name, adding a small automated test, or extracting a method to improve readability and the possibility for future reuse.

Adopt “Leave It in a Better State” as a way of working and life will be much easier for you in the long term.

---

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

[123]http://programmer.97things.oreilly.com/wiki/index.php/Leave_It_in_a_Better_State

Methods Matter

By Matthias Merdes[124]

A large part of today's software is written using object-oriented languages. Object-oriented software is composed of objects communicating via methods. So in a way it can be argued that not objects but methods are the basic building blocks of our code. While there is a lot of literature on design in the large – architecture and components – and on medium granularity – e.g., design patterns – surprisingly little can be found on designing individual methods. Design in the small, however, matters. A lot.

Ideally, any piece of source code should be readable like a good book: it should be interesting; it should convey its intention clearly; and last, but not least, it should be fun to read.

The simplest way to achieve readability is to use expressive names that properly describe the concepts they identify. In most cases this will mean relatively long names for methods and variables. Some argue that short names are easier and thus faster to type. Modern IDEs and editors are capable of simplifying the typing, renaming, and searching of names to make this objection less relevant. Instead of thinking about the typing overhead today, think about the time saved tomorrow when you and others have to reread the code. Be particularly careful when designing your method names because they are the verbs in the story you are trying to tell.

Keeping it simple (KISS) is a general rule of thumb in software design. One source of simplicity is brevity: Most of the time, shorter methods are simpler than long ones. While a low line count is a good starting point, the overall cognitive load can be further reduced by keeping the cyclomatic complexity low – ideally under 3 or 4, definitely under 10. Cyclomatic complexity is a numeric value that can easily be computed by many tools and is roughly equivalent to the number of execution paths through a method. A high cyclomatic complexity complicates unit testing and has been empirically shown to correlate with bugs.

Mixing concerns is often considered harmful. This is normally applied to classes as a whole, but applied at the method level it can further increase the readability of your code. Applied to methods this means that you should strive to map different aspects of your required behavior cleanly to different methods. Such a separation of technical and business concerns facilitates, and benefits from, the use of a well-defined domain vocabulary. Using such a vocabulary helps to reveal intention in your methods. This can ensure that a consistent and ubiquitous vocabulary is used when speaking to domain experts, but it also ensures that methods are focused and consistent, so that developers also benefit.

[124] http://programmer.97things.oreilly.com/wiki/index.php/Matthias_Merdes

When you have properly separated concerns try to do the same with levels of abstraction. This is achieved by staying at a single level of abstraction within each method. This way you don't jump back and forth between little technical details and the grander motives of your code narrative. The resulting methods will be easier to grasp and more pleasant to read.

All in all, careful design of short methods with low complexity that focus on a single fine-grained concern and stay at a single level of abstraction combined with proper expressive naming will definitely help to better convey the intention of your code to other developers – and to yourself. This will improve maintainability in the long run and, after all, most software development is maintenance.

Happy method design!

---

[125]http://creativecommons.org/licenses/by/3.0/us/
[126]http://programmer.97things.oreilly.com/wiki/index.php/Methods_Matter

The Programmer’s New Clothes

By Ryan Brush[127]

The conversation goes like this:

> *Mortal Developer*: This design seems complicated. Can we change the whack-a-mole feature? It adds extra steps to every use of the system. *Expert Domain Programmer*: No, some users might need whack-a-mole to interact with their legacy bug tracking system.

Now our poor mortal developer is stuck. After all, this is the expert we’re talking to here. If he says we need whack-a-mole, you better get whacking.

Our expert missed this: Complexity alone can be cause to reject a design. We need to start with an open mind, looking to understand how every aspect of a design pulls its weight. But you’re a smart programmer; if a system is difficult for you to understand, it might be too complicated. Now you have to point out and address that complexity. Sadly, there sometimes seems to be an “Emperor’s New Clothes” phenomenon in software. No one wants to admit something is hard to understand in an industry where intelligence is considered the greatest virtue.

From the other side, as we develop domain expertise we are prone to a trap. We live and breath the intricacies of the domain, becoming so fluent we can’t see the challenges of a novice. It’s like being a native speaker of a language: I don’t think about the huge number of special rules in English, so a particular rule doesn’t seem very onerous to me. But put them together and these rules present a huge obstacle to someone starting to learn it.

Yet there is hope. Unlike natural language, we can control the complexity of our designs. Imagine the biggest, most complicated specification you’ve seen. Imagine if the design team included a smart, experienced person without domain knowledge, but with veto power. Could we cut out much of the accidental complexity?

Some humble advice for anyone struggling with this right now:

- Remember Alan Kay’s insight: “Simple things should be simple, complex things should be possible.” If something complicated must be in the system, it still should not affect the simple things.

[127] http://programmer.97things.oreilly.com/wiki/index.php/Ryan_Brush

- View the project as a constant battle against complexity. A single complex module may seem unimportant, but it quickly compounds.
- Understand a project end-to-end. The project should be easily broken down into problems that are known to be solvable.
- A strong-willed engineer or executive can will a group down the wrong path. It's an engineer's duty to object and prevent spiraling complexity.
- Brian Kernighan said it well: "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"

All of this boils down to *Keep It Simple, Stupid*. It turns out keeping things simple can be pretty hard.

---

[128]http://creativecommons.org/licenses/by/3.0/us/
[129]http://programmer.97things.oreilly.com/wiki/index.php/The_Programmer%27s_New_Clothes

Programmers Are Mini-Project Managers

By Jonathan Danylko[130]

Every programmer has experienced this at one time or another where someone asked them to just "whip something up."

The project could initially appear as simple as a three-page web site, but when you actually sit down and talk with the customer, you realize they want to build an e-commerce system with resume-building capabilities, a forum, and a CMS (Content Management System) with all the bells and whistles.

Oh, and they are preparing to launch in a month.

Of course, we can't all take out a rolled-up newspaper and smack them on the nose saying "No!" But instead of freaking out, most professional programmers immediately ask the following questions:

- How much time is available to complete the project? (Time)
- How good is the code you write? (Quality)
- How much is available in the budget for this project? (Cost)
- What features are included before the launch? (Scope)

I know these questions are moving towards "project management" territory, but every programmer who has been in the industry for a long period of time understands that these factors creep into every single program they write, whether it's a fat client or a web application.

These four important factors determine whether a programming team (whether one member or twenty) will complete a project successfully or fail miserably in the customer's eyes:

- *Time*: How much time is available for coding, testing, and deployment? If there isn't enough time for coding, testing, or deployment, this may sacrifice quality because you are pressured through the tasks and may produce inefficient code.
- *Quality*: If you want maintainable code, you may lose time and the cost may increase over an approach that compromises quality. On the other hand, the technical debt of compromising quality is also likely to lose time later therefore increasing cost in the long-term.

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

- *Cost*: If you want it cheap, you may have a coding frenzy with a lot of unmaintainable code and gain some time, but the quality of the product will suffer.
- *Scope*: If you want to release a quality product, you may need to focus on the features that really matter to the user. Perform a temporary *feature toss* to release the product on time and save the other features for a future update.

The key to overcoming these "rectangle of tangles" is to ask the user some questions about the project. If you ask enough questions, you'll become more comfortable with what the user wants. The more comfortable you are with understanding what the user wants, the more comfortable it is to know what key components will be the hardest and easiest to create.

If you understand what the user is looking for, you will have an idea of how long it will take to finish a project (Time). Based on the time factor, you can gradually move towards a dollar estimate of the project (Cost) because you know what features need to be built (Scope). Finally, based on your skill set (since you are a professional programmer), you should be able to produce a really high-quality product.

Right?

---

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

[132]http://programmer.97things.oreilly.com/wiki/index.php/Programmers_Are_Mini-Project_Managers

Programmers Who Write Tests Get More Time to Program

By Johannes Brodwall[133]

I became a programmer so that I could spend time creating software by programming. I don't want to waste my time managing low-quality code.

Does your code work the first time you test it out? Mine certainly never does. So if I hand the code over to someone else, I'm sure to get a bug report back that will take up my valuable programming time. This much is obvious to most developers. However, it also means that I don't want to have to go thought a long manual test myself to verify my code.

The first thing I do when I'm starting on a programming task is ask myself: "How am I going to test that this actually works?" I know it has to be done, I know it will take up a lot of my time if I do a poor job of it, so I want to get it right.

A good way to start is by writing tests before you write the code. The tests will specify the required behavior of the code. If you write the tests with the question "How will I know when my task is complete?" the chances are that not only will your tests will be better for it, your design will also have improved.

I've come to codebases that were otherwise good, but that required me to write a lot of code to support my tests. In other words: The code was overly complex to use.

For example, a system that is built with an asynchronous chain of services connected via a message bus might require you to deploy your code before you can test it. I've redesigned such code in a couple of steps that progressively reduced the coupling, improved the cohesion, and simplified the testing of the code:

1. Allow the code to be run synchronously and in-process by a configuration change. The messaging infrastructure is no longer coupled to the business logic. The resulting design is both easier to follow and more flexible, in addition to being easier to test. However, testing still requires setting up a long chain of services.
2. Refactoring my services to calculate the result they send to the next service by using a transformation function makes the responsibilities of different parts of the code clearer. And I can test almost all the logic by just calling this transformation function and checking the return value.

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

When encountering a programming task, ask yourself: “How can I test this?” If the answer is “Not very easily” try to write a test anyway. The test will probably require a lot of setup and it may be hard to verify the results. These are design problems, not test problems. Use the information from the testing process to refactor your system to a better design.

May you achieve fewer bugs and spend all your days programming happily in a well-designed system!

---

[134]http://creativecommons.org/licenses/by/3.0/us/
[135]http://programmer.97things.oreilly.com/wiki/index.php/Programmers_Who_Write_Tests_Get_More_Time_to_Program

Push Your Limits

by Karoline Klever[136]

For many areas in life you need to know your limits in order to survive. Where it concerns personal safety, your limits define a boundary that should not be crossed. Where it concerns personal limitations, skills, and knowledge, however, knowing your limits serves an entirely different purpose. In programming, you want to know your limits so that you can pass them in order to become a better programmer.

Fortunately, not many programmers view their code as a ticket to their next paycheck. Those of us who are truly programmers at heart thrive on immersing ourselves in new code and new concepts, and will never cease to take an interest in and learn new technology. Whichever technology or programming language these programmers favor, they have one important thing in common: They know their limits, and they thrive on pushing them little by little every day.

We have all experienced that, if not armed with the knowledge we need or wished we had, attacking a bug or a problem head on takes a lot of time and effort. It can be too easy to pass the beast on to a colleague you know has the solution or can find one quickly. But how will that help you become a better problem solver? A programmer unaware of their own limits – or, worse, aware of them but not challenging them – is more likely to end up working in a never-ending loop of the same tasks every day. To insert a break into this loop, you need to acknowledge your limits by defining your programmatic weaknesses: Is your code readable by others? Are your tests sufficient? Focus on your weaknesses and you'll find that at the end of the struggle, you will have pushed that boundary an inch forward.

Needless to say, what others know should not be forgotten. Fellow programmers and colleagues offer a nearby source of knowledge and information which should be shared and taught to others. They are, however, ignorant of your limits. It is your responsibility to acknowledge exactly where your boundaries are hiding so that you can make better use of the knowledge they offer. One of the most challenging and, many would say, most effective ways to extend your own limits is to explain code and concepts to fellow programmers, for example by blogging or hosting a presentation. You will then force yourself to focus on one of your weaknesses, while at the same time deepening your own and others' knowledge by discussing and getting feedback from others.

At the end of the day, it is not about surviving. It is about surviving in the best way possible, by actively challenging yourself to excel as a programmer.

[136] http://programmer.97things.oreilly.com/wiki/index.php/Karoline_Klever

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

[138]http://programmer.97things.oreilly.com/wiki/index.php/Push_Your_Limits

QA Team Member as an Equal

By Ravindar Gujral[139]

Many organizations give a lot of weight to their developers but neglect their QA departments. There are many misconception about the role of QA. This happens more in bigger organizations where there is a greater emphasis on strict demarcation of responsibilities. This creates a culture of silos. It encourages developers to believe that they are not required to interact with QA. Most developers start thinking that they are better than QA and that all QA does is to nitpick at the beautiful software the developer has delivered.

Quality Assurance is more than defect recognition or Quality Control. A QA team member combines some unique skills, skills that embrace technical, process and, functional understanding of the system as well as a keen eye towards usability, especially critical for a UI-based system. Software QA (involving QC) is a craft, just like software development, and to think otherwise is to not understand software. Agile practices have helped reduce the misconceptions held regarding QA but there is a serious lack of understanding of the role of QA team members in most organizations, even the ones that adopt agile practices.

So let’s review just some of what a QA team member brings to a team, especially an agile team:

- They interact with the customer and have an in-depth understanding of features being implemented. A good QA team member understands what the system does functionally. They illuminate dark areas of software.
- They interact with the developers, understanding the technical implementation of a feature. They work with the developers to help write valid tests for features being implemented.
- They help understand patterns of implementations and help improve process and consistency of software developed. They help with automation of tests, offering developers the rapid feedback that is necessary as they build software.
- They help developers maintain the quality of the software features being delivered.

Above points are but a few things that a QA team member provides to a team. If you find that your organization sidelines QA, remind them the importance of QA.

Every organization should put as much time into hiring a QA team member as they put in hiring a developer. QA is a craft that takes years of practice. Your brightest and the smartest team members should consider becoming QA. Also every developer should learn to respect QA team members and treat them as equals.

[139]http://programmer.97things.oreilly.com/wiki/index.php/Ravindar_Gujral

---

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

[141] http://programmer.97things.oreilly.com/wiki/index.php/QA_Team_Member_as_an_Equal

Reap What You Sow

By Seb Rose[142]

Software professionals are a cynical bunch. We complain about management. We complain about customers. Most of all, we complain about other programmers. Whether it's code structure, test coverage, design decomposition, or architectural purity, there's always something that we think others do badly. How can this be?

Well, we learn a lot during formal education, but when we arrive at our first 'real' job we find that what we learnt at school isn't particularly relevant. We enter a new phase of learning, where we get conditioned to the environment at our new employers. There will be new standards, practices, processes, and technologies. Those of us who change jobs regularly get used to (and even look forward to) this learning curve at the beginning of a new assignment. As well as changing employer, there are also advances in the industry to consider – programming languages change, operating systems evolve, development processes mutate.

There are, however, plenty of software professionals who get comfortable with their knowledge and stop learning. They may resist change, even when it can be shown to be industry best practice, on the basis that "it isn't the way we do things here." Beyond the essentials, such as learning about the latest version of their IDE, they may not track changes in the industry at all. Given that you are reading this collection, I hope that it is safe for me to assume that you do not belong to that group. But are you doing anything to encourage others to continue their professional development?

Do you bring books to work and point your colleagues at interesting blogs? Do you forward links to relevant articles, mailing lists, and communities? Do you volunteer to present sessions on new techniques you come across and suggest process improvements to help deliver better value to your customers? In short, do you demonstrate that you are a professional, dedicated to the continuing development of our industry?

Your efforts may often be ignored, but don't give up. You will learn by trying, even if your seed lands on fallow ground. From time to time one (or more) of your colleagues will respond favorably to your activities and this is a cause for celebration. The angels may not rejoice, but (in a small way) you have made the world a better place!

So, put away that cynicism and disseminate your knowledge. Infect your colleagues with your passion. You do not have to be a passive victim of poorly educated peers. You are an active member of an evolving discipline. Go forth and sow the seeds of knowledge, and you may well reap the benefits.

[142]http://programmer.97things.oreilly.com/wiki/index.php/Seb_Rose

---

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

[144]http://programmer.97things.oreilly.com/wiki/index.php/Reap_What_You_Sow

Respect the Software Release Process

By Pete Goodliffe[145]

Presuming that you are writing software for the benefit of others as well as yourself, it has to get into the hands of your "users" somehow. Whether you end up rolling a software installer shipped on a CD or deploying the software on a live web server, this is the important process of creating a *software release.*

The software release process is a critical part of your software development regimen, just as important as design, coding, debugging, and testing. To be effective your release process must be: simple, repeatable, and reliable.

Get it wrong, and you will be storing up some potentially nasty problems for your future self. When you construct a release you must:

- Ensure that you can get the exact same code that built it back again from your source control system. (You do use source control, don't you?) This is the only concrete way to prove which bugs were and were not fixed in that release. Then when you have to fix a critical bug in version 1.02 of a product that's five years old, you can do so.
- Record exactly how it was built (including the compiler optimization settings, target CPU configuration, etc.). These features may have subtly affects how well your code runs, and whether certain bugs manifest.
- Capture the build log for future reference.

The bare outline of a good release process is:

- Agree that it's time to spin a new release. A formal release is treated differently to a developer's test build, and should **never** come from an existing working directory.
- Agree what the "name" of the release is (e.g., "5.06 Beta1" or "1.2 Release Candidate").
- Determine exactly what code will constitute this release. In most formal release processes, you will already be working on a *release branch* in your source control system, so it's the state of that branch right now.
- Tag the code in source control to record what is going into the release. The tag name must reflect the release name.

[145] http://programmer.97things.oreilly.com/wiki/index.php/Pete_Goodliffe

- Check out a virgin copy of the entire codebase at that tag. **Never** use an existing checkout. You may have uncommitted local changes that change the build. Always tag *then* checkout the tag. This will avoid many potential problems.
- Build the software. This step **must not** involve hand-editing any files at all, otherwise you do not have a versioned record of exactly the code you built.
- Ideally, the build should be automated: a single button press or a single script invocation. Checking the mechanics of the build into source control with the code records unambiguously how the code was constructed. Automation reduces the potential for human error in the release process.
- Package the code (create an installer image, CD ISO images, etc.). This step should also be automated for the same reason.
- Always test the newly constructed release. Yes, you tested the code already to ensure it was time to release, but now you should test this "release" version to ensure it is of suitable release quality.
- Construct a set of "Release notes" describing how the release differs from the previous release: the new features and the bugs that have been fixed.
- Store the generated artifacts and the build log for future reference.
- Deploy the release. Perhaps this involves putting the installer on your website and sending out memos or press releases to people who need to know. Update release servers as appropriate.

This is a large topic tied intimately with configuration management, testing procedures, software product management, and the like. If you have any part in releasing a software product you really must understand and respect the sanctity of the software release process.

---

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

[147]http://programmer.97things.oreilly.com/wiki/index.php/Respect_the_Software_Release_Process

Restrict Mutability of State

By Kevlin Henney[148]

"When it is not necessary to change, it is necessary not to change." – Lucius Cary

What appears at first to be a trivial observation turns out to be a subtly important one: A large number of software defects arise from the (incorrect) modification of state. It follows from this that if there is less opportunity for code to change state, there will be fewer defects that arise from state change!

Perhaps the most obvious example of restricting mutability is its most complete realization: immutability. A moratorium on state change is an idea carried to its logical conclusion in pure functional programming languages such as Haskell. But even the modest application of immutability in other programming models has a simplifying effect. If an object is immutable it can be shared freely across different parts of a program without concern for aliasing or synchronization problems. An object that does not change state is inherently thread-safe – there is no need to synchronize state change if there is no state change. An immutable object does not need locking or any other palliative workaround to achieve safety.

Depending on the language and the idiom, immutability can be expressed in the definition of a type or through the declaration of a variable. For example, Java's `String` class represents objects that are essentially immutable – if you want another string value, you use another string object. Immutability is particularly suitable for value objects in languages that favor predominantly reference-based semantics. In contrast, the `const` qualifier found in C and C++, and more strictly the `immutable` qualifier in D, constrain mutability through declaration. `const` qualification restricts mutability in terms of access rights, typically expressing the notion of read-only access rather than necessarily immutability.

Perhaps a little counterintuitively, copying offers an alternative technique for restricting mutability. In languages offering a transparent syntax for passing by copy, such as C#'s `struct` objects and C++'s default argument passing mode, copying value objects can greatly improve encapsulation and reduce opportunities for unnecessary and unintended state change. Passing or returning a copy of a value object ensures that the caller and callee cannot interfere with one another's view of a value.

[148] http://programmer.97things.oreilly.com/wiki/index.php/Kevlin_Henney

But beware that this technique is somewhat error prone if the passing syntax is not transparent. If programmers have to make special efforts to remember to make the copy, such as explicitly call a `clone` method, they are also being given the opportunity to forget to do it. It becomes a complication that is easy to overlook rather than a simplification.

In general, make state and any modification to it as local as possible. For local variables, declare as late as possible, when a variable can be sensibly initialized. Try to avoid broadcasting mutability through public data, global and class `static` variables (which are essentially globals with scope etiquette), and modifier methods. Resist the temptation to mirror every *getter* with a *setter*.

Restricting mutability of state is not some kind of silver bullet you can use to shoot down all defects. But the resulting code simplification and improvement in encapsulation make it less likely that you will introduce defects, and more likely that you can change code with confidence rather than trepidation.

---

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

[150]http://programmer.97things.oreilly.com/wiki/index.php/Restrict_Mutability_of_State

Reuse Implies Coupling

By Klaus Marquardt[151]

Most of the Big Topics (capital *B*, capital *T*) in the discussion of software engineering and practices are about improving productivity and avoiding mistakes. Reuse has the potential to address both aspects. It can improve your productivity since you needn't write code that you reuse from elsewhere. And after code has been employed (reused) many times, it can safely be considered tested and proven more thoroughly than your average piece of code.

It is no surprise that reuse, and all the debate on how to achieve it, has been around for decades. Object-oriented programming, components, SOA, parts of open source development, and model-driven architecture all include a fair amount of support for reuse, at least in their claims.

At the same time, software reuse has hardly lived up to its promises. I think this has multiple causes:

1. It is hard to write reusable code.
2. It is hard to reuse code.
3. Reuse implies coupling.

The first cause has to do with interface design, negotiations with many customers, and marketing.

The second has two aspects: It takes mental effort to want to reuse; it takes technical effort to reuse. Reuse works fine with operating systems, libraries, and middleware, whether commercial or open source. We often fail, however, to reuse software from our colleagues or from unrelated projects within our company. Not only is it way more sexy to design something yourself, reuse would also make you depend on somebody else.

Which brings us to the third cause. Dependency means that you are no longer the smith of your own luck. You will be fine as long as the code you reuse is considered a commodity, something you really really don't want to do yourself. The closer you come to the heart and style of your application, the easier you find good reasons why to not reuse – depending on something you don't own, know, maintain, or schedule. And given you decided to reuse some code from elsewhere against some odds, the owner of that code might not appreciate and support your initiative. Providing code for reuse necessitates a more careful design, more deliberate change control, and additional effort to support your users, leaving less time for other duties.

[151]http://programmer.97things.oreilly.com/wiki/index.php/Klaus_Marquardt

The coupling issues are amplified when you implement reuse across your company. All of a sudden, the provider and all of the reusers are coupled to each other. And worse, even the reusers are indirectly coupled to one another. Each feature or change initiates debates about its relevance and priority, its schedule, and which interfaces will be affected. All the reusing projects struggle to have their expectation covered first. Software reuse requires a tremendous amount of management, control, and overhead to enable and sustain its success.

There is a silent way to start software reuse. Different projects may exchange code and knowledge, and allow to use each others code at the users own risk. This approach starts with minimal provider effort – and a license for forking, causing deviations of the reused code for different projects. While the projects evolve their own variant, this ceases to be reuse rather soon. It becomes reuse when, every now and then, all the variants are reintegrated into an evolving baseline that is then used and becomes more stable over the months and years. With this mindset of sustainability over accountability, your company might be able to achieve the prerequisite for successful reuse: Software that is considered a commodity.

---

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

[153]http://programmer.97things.oreilly.com/wiki/index.php/Reuse_Implies_Coupling

Scoping Methods

By Michael Hunger[154]

It has long been recommended that we should scope our variables as narrowly as possible. Why is that so?

- Readability is greatly improved if the scope of a named variable is so small that you can see its declaration only a few lines above its usage.
- Variables leaving scope are quickly reclaimed from the stack or collected by the garbage collector.
- Invalid reuse of locally scoped variables is impossible.
- Singular assignment on declaration encourages a functional style and reduces the mental overhead of keeping track of multiple assignments in different contexts
- Local variables are not shared state and are therefore automatically thread safe.

But what about scoping our methods?

We try to decompose methods into smaller units of computation, each of which is easily understandable. This goes hand in hand with the principle that a method should deal only with a *single level of abstraction*. If the result, however, is that your class then houses too many small methods to be easily understandable, it's time to rescope its methods. Although similar in some ways, that's not quite the same as decomposing classes with low cohesion into different smaller classes. The class may be quite cohesive, it's just that is spans too many levels of abstraction.

Although there are layout rules that make locality and access more significant (like putting a private method just below the first method that uses it), scoping is a cleaner way of separating cohesive parts of a class.

How do you scope methods?

Create objects that correspond to the public methods, and move the related private methods over to the method objects. If you had parameter lists for the private methods – especially long ones – you can promote these some of these parameters to instance variables of the method object.

You then have your original class declare its dependencies on these fragments and orchestrate their invocation. This keeps your original class in a coordinating role, freed from the detail of private

[154]http://programmer.97things.oreilly.com/wiki/index.php/Michael_Hunger

methods. The lifetime of your method objects depends on their intended use. Mostly I create them within the scope just before being called and let them die immediately after. You may also choose to give them more significant status and have them passed in from outside the object or created by a factory.

These method objects give the newly created method scope a name and a location. They stay very narrowly focused and at a consistent level of abstraction. Often they become home for more functionality working on the state they took with them, e.g., the promoted parameters or the instance variables used by just these methods.

If you have private methods that are often reused within different other methods it's perhaps time to accept their importance and promote them to public methods in a separate method object.

---

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

[156]http://programmer.97things.oreilly.com/wiki/index.php/Scoping_Methods

Simple Is not Simplistic

By Giovanni Asproni[157]

“Very often, people confuse simple with simplistic. The nuance is lost on most.” Clement Mok

From the *New Oxford Dictionary Of English*:

- **simple** easily understood or done; presenting no difficulty
- **simplistic** treating complex issues and problems as if they were much simpler than they really are

In principle we all appreciate that simple software is more maintainable, has fewer bugs, has a longer lifetime, etc. We like to think that we always try to implement the most appropriate solutions, aspiring to this condition of simplicity. In practice, however, we also know that many developers often end up with unmaintainable code very quickly.

In my experience, the most common reason for that is due to lack of understanding of what the real problem than needs to be solved is. In fact, before implementing a new piece of functionality, there are several equally important things to do:

1. Understand the requirements: Is what the users are asking for what they *really* need?
2. Think about how to fit the functionality into the system *cleanly*: What parts of the current system, if any, need to change to best accommodate it?
3. Think about what and how to test: How can I demonstrate that the functionality is implemented correctly? How can I make it so that the tests are simple to write and simple to run?
4. Given all the above, think about the time necessary to implement it: Time is always a major concern in software projects.

[157]http://programmer.97things.oreilly.com/wiki/index.php/Giovanni_Asproni

Unfortunately, when working on a "simple" solution, many developers do the following: gloss over point (1), assuming that the users actually know what they need; consider point (2), but forgetting the part about *cleanly*; skip point (3) altogether; finally, reduce the time at point (4) as much as possible by cutting corners. Far from being simple, that is actually a simplistic solution.

The net result is an increase in a system's internal complexity when this short-cut approach is used repeatedly to implement and add to the system's functionality. The maintainability and extensibility are affected negatively. Defects, however, are affected positively. And users will most likely be unhappy because the functionality is unlikely to match their expectations or their needs.

Doing the right thing – i.e., attending to all the above points considerately – in the short term requires more immediate work, and feels harder and more time consuming. In the medium to long term, however, the system will be easier and less expensive to maintain and evolve. The users (and the developers) will also be much happier.

Of course, there may be times when a solution is required very quickly and a clean implementation in a short time is impossible. However, hacking a solution should be a deliberate choice. The costs – the impact of the accumulated technical debt – have to be weighed carefully against any gains.

As Edward De Bono wrote, "simplicity before understanding is simplistic; simplicity after understanding is simple."

---

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

[159]http://programmer.97things.oreilly.com/wiki/index.php/Simple_Is_not_Simplistic

Small!

by Uncle Bob[160]

Look at this code:

```java
private void executeTestPages() throws Exception {
  Map<String, LinkedList<WikiPage>> suiteMap = makeSuiteMap(page, root, getSuit\
eFilter());
  for (String testSystemName : suiteMap.keySet()) {
    if (response.isHtmlFormat()) {
      suiteFormatter.announceTestSystem(testSystemName);
      addToResponse(suiteFormatter.getTestSystemHeader(testSystemName));
    }
    List<WikiPage> pagesInTestSystem = suiteMap.get(testSystemName);
    startTestSystemAndExecutePages(testSystemName, pagesInTestSystem);
  }
}
```

Your first reaction is probably not positive. Not because the code is that complicated or daunting, but just because you don't necessarily feel like untangling the intent hidden in 11 lines of code with 3 levels of indent. You know you can do it, but it feels like work.

Now look at this function:

```java
private void executeTestPages() throws Exception {
  Map<String, LinkedList<WikiPage>> pagesByTestSystem;
  pagesByTestSystem = makeMapOfPagesByTestSystem(page, root, getSuiteFilter());
  for (String testSystemName : pagesByTestSystem.keySet())
    executePagesInTestSystem(testSystemName, pagesByTestSystem);
}
```

Notice that it doesn't feel so much like work. You can look at it, and grasp the intent without much effort. Not much has changed, I've just cleaned up the code a little. And yet that small change, so easy to do with modern refactoring browsers and a suite of tests, makes the function much easier to read.

The extracted functions are pretty easy to read too:

[160]http://programmer.97things.oreilly.com/wiki/index.php/Uncle_Bob

```
 private void executePagesInTestSystem(String testSystemName,
                                       Map<String, LinkedList<WikiPage>> pagesBy\
TestSystem) throws Exception {
   List<WikiPage> pagesInTestSystem = pagesByTestSystem.get(testSystemName);
   announceTestSystem(testSystemName);
   startTestSystemAndExecutePages(testSystemName, pagesInTestSystem);
 }


 private void announceTestSystem(String testSystemName) throws Exception {
   if (response.isHtmlFormat()) {
     suiteFormatter.announceTestSystem(testSystemName);
     addToResponse(suiteFormatter.getTestSystemHeader(testSystemName));
   }
 }
```

The point is that functions should be *small*. How small? Just a few lines of code with one or two levels of indent.

“You can’t be serious!” I hear you say. But serious I am. It is far better to have many small functions than a few large ones.

“But doesn’t the proliferation of functions make the code more confusing?”

It certainly does if the proliferated functions are scattered hither and yon with no sense of organization. However, when those small functions gathered together into a well ordered and organized module, then they aren’t confusing at all. Large and deeply indented functions are much more confusing that a well organized set of simple little functions.

Think of it this way. When you were young you had a “system” for knowing where all your things were. They were on the floor of your room, or under the bed, or in a pile in your closet. Your mother would yell at you from time to time to clean up your room, but you did your best to thwart her intent because your system worked just fine for you. You knew that tomorrow’s socks were right on the floor where you left them last night. The same for your underwear. You knew that your favorite toy was under your bed somewhere. You had a system.

But finally your mother got so frustrated that she forced you to help her (meaning watch her) clean up your room. You watched as she hung clothes up in the closet, and put toys on shelves or in drawers. You watched as she organized your things and put them away. And (sometime in your 30s) you realized that she had a point.

Yes, it’s generally better to have a place for everything and put everything in it’s place. And that’s just what dividing your code into many small functions is. Large functions are just like all the clothes under your bed. Splitting them up into many little functions is like putting all your clothes on hangers and sorting them nicely in your closet. Large functions are a child’s way to organize. Small functions are an adult’s (or should I say a professional’s) way to organize.

---

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

[162]http://programmer.97things.oreilly.com/wiki/index.php/Small%21

Soft Skills Matter

By Bruce Rennie[163]

A good friend of mine – a developer I respect a great deal – and I have an ongoing, friendly argument. The essence of the argument boils down to which set of skills is more important for a developer: hard, technical skills or soft, people skills?

Developers can hardly be blamed for focusing on hard skills. It seems like every day a new language, framework, API, or toolkit is released. Developers can, and do, invest considerable time in simply keeping up. And, let's be honest, we work in an industry that makes a virtue out of technical prowess. Rock star programmers, anyone?

Why should developers care about soft skills? For most of us, there is one very good reason: We don't work alone. The overwhelming majority of us will spend our careers working in teams. Our fate is not solely in our own hands. If we want to succeed, we need the help of others. So, what are some of the skills that can help us in a team situation?

- The ability to communicate ideas and designs quickly and clearly.
- The ability to listen to the ideas of others.
- Enough confidence to lead.
- Enough self-esteem to follow.
- The ability to teach.
- The willingness to learn.
- A desire to promote consensus combined with the courage to accept conflict in pursuit of that consensus.
- Willingness to accept responsibility.
- Above all, respect for your teammates.

Respect, while not normally recognized as a skill, is essential and covers a lot of ground. It can be exhibited in something as difficult as politely delivering (or receiving) constructive criticism. On the other end of the spectrum it can be something as simple as bathing regularly. Believe me, it all matters to your teammates.

Of course, the answer to our friendly argument is that you need both sets of skills. However, I've never personally witnessed a project go south due to a lack of technical expertise and, while I know

[163]http://programmer.97things.oreilly.com/wiki/index.php/Brucer

it does happen, I generally have faith in my colleagues' ability to absorb new technologies. On the other hand, I have seen projects fail, almost before they left the gate, simply because the team couldn't work together. And I have seen teams of developers who might otherwise be described as 'average' come together like finely tuned engines and produce something amazing. I know which I prefer.

---

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

[165]http://programmer.97things.oreilly.com/wiki/index.php/Soft_Skills_Matter

Speed Kills

by Uncle Bob[166]

You are a programmer. That means you are under tremendous pressure to go fast. There are deadlines to meet. There are bugs to fix before the big demo. There are production schedules to keep. And your job depends on how fast you go and how reliably you keep your schedules. And that means you have to cut corners, compromise, and be quick and dirty.

Baloney.

You heard me. Baloney! There's no such thing as quick and dirty in software. Dirty means slow. Dirty means death.

Bad code slows everyone down. You've felt it. I've felt it. We've all been slowed down by bad code. We've all been slowed down by the bad code we wrote a month ago, two weeks ago, even yesterday. There is one sure thing in software. If you write bad code, you are going to go slow. If the code is bad enough, you may just grind to a halt.

The only way to go fast is to go well.

This is just basic good sense. I could quote maxim after maxim. Anything worth doing is worth doing well. A place for everything and everything in its place. Slow and steady wins the race. And so on, and so forth. Centuries of wisdom tell us that rushing is a bad idea. And yet when that deadline looms….

Frankly, the ability to be deliberate is the mark of a professional. Professionals do not rush. Professionals understand the value of cleanliness and discipline. Professionals do not write bad code – ever.

Team after team has succumbed to the lure of rushing through their code. Team after team has booked long hours of overtime in an attempt to get to market. And team after team has destroyed its projects in the attempt. Teams that start out moving quickly and working miracles often slow down to a near crawl within a few months. Their code has become so tangled, so twisted, and so interdependent, that nobody can make a change without breaking eight other modules. Nobody can touch a module without having to touch twenty others. And every change introduces new unforeseen side-effects and bugs. Estimates stretch from days to weeks to months. The team slows to a grinding plod. Managers are frantic and developers start looking for new jobs.

[166]http://programmer.97things.oreilly.com/wiki/index.php/Uncle_Bob

And what can managers do about it? They can scream and yell about going faster. They can make the deadlines loom even closer. They can browbeat and cajole. But in the end, nothing works except hiring more programmers. And even that doesn't work for long, because the new guys simply add even more mess on top of the old mess. In a short time the team has slowed again, continuing its inexorable march towards the asymptote of zero productivity.

And whose fault is this disaster? The programmers. You. Me. We rushed. We made messes. We did not stay clean. We did not act professionally.

If you want to be a professional, if you want to be a craftsman, *then you must not rush*. You must keep your code clean. So clean it barely needs comments.

---

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

[168]http://programmer.97things.oreilly.com/wiki/index.php/Speed_Kills

Structure over Function

by Peter Sommerlad[169]

When learning to program, once you have mastered the syntax of the programming language, the function of a piece of code is the most important thing to get right. It feels great to pass the hurdle of getting a program to actually *run* and do what you intended. Unfortunately, that initial satisfaction with your programming skills can be misleading. Programs that (seem to) work are necessary but not sufficient for your life as a good programmer.

So what is missing?

You and others will have to read, understand, use, and modify your program code. Changes to code are inevitable. You therefore need to ensure that your code can evolve while its functionality survives.

One of the most significant barriers to evolution is the code's structure. Significant structure exists at many levels: the number and order of statements in a function, loop and conditional blocks; the nesting within control structures; the functions within a module or class; the relationships between one piece of code and others; the partitioning and dependencies between classes, modules and subsystems – its overall design and architecture.

Hindrance to change from poor structure comes in many different species. The simplest taxonomy of code structure is in terms of high cohesion (i.e., the Single Responsibility Principle) and low coupling. Both are easy to measure and understand, and yet very hard to achieve. Violation of these principles is contagious, so without an antidote, code that depends on other, poorly structured code tends to degenerate as well. Take low cohesion and high coupling, or other metrics showing structural disorder, as symptoms to be diagnosed and remedied.

Refactoring is the treatment to improve code structure. Automated refactoring, test automation, and version control provide safety, so that treatment does no harm or can be easily undone.

While the need to refactor is almost inevitable – because we learn about better solutions while we program – there are also preventive measures that make refactoring less expensive and burdensome.

Where poor structure can occur at all levels, good structure builds from the ground up. Keep your program code clean and understandable from the statement level to the coarsest partitioning. Consider your code as *not* "working" unless it is actually working in a way that you and other programmers can easily understand and evolve. A user might not initially recognize bad structure,

[169]http://programmer.97things.oreilly.com/wiki/index.php/Peter_Sommerlad

so long as functionality is available. However, evolving the system might not happen at the pace the user expects or desires once structural decay in the code sets in.

Keep your code well organized. Program deliberately. Refactor mercilessly towards DRY code. Use patterns and simplicity to guide your efforts to improve the code's structure. Understand and evolve a system's architecture towards doing more with less code. Avoid too much up-front genericity and refactor away from laborious specific solutions repeating code.

As a professional programmer you will know that function is important, but you need to focus on structural improvement when programming, if your system is to grow beyond the scale of "Hello, World!"

---

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

[171]http://programmer.97things.oreilly.com/wiki/index.php/Structure_over_Function

Talk about the Trade-offs

by Michael Harmer[172]

So you've got your specification, story, card, bug report, change request, etc. You've got a copy of the latest code and you are about to start designing. STOP!

Don't do a thing until you understand the attributes that the completed code is supposed to exhibit. Are there runtime attributes the code should have (performance, size)? Perhaps there are constraints on the production of the code (time to complete, total effort, total cost, language)? Maybe long-term attributes of the code are important (maintainability, language)?

There are a surprisingly large number of ways in which the various attributes of code (and architectures, UIs, etc.) are traded off against one another. There is an equally large discrepancy between the default approach taken by programmers and their managers. Some people will think that everything must be optimized for speed. Others will spend forever ensuring that variable declarations are lined up in source files. While others will obsess about W3C standards compliance and usability.

If you use your default approach or make an assumption, there is a very good chance that you'll end up doing the wrong thing. Ask what trade-offs there are. If everyone looks blank then here's you chance to make a real difference. Carefully consider and then suggest what trade-offs there are between different attributes. It will help ensure that everyone pulls in the same direction, will flush out conflicts by allowing you and the team to discuss problems with reference to a concrete list, and may well help to save the project.

The list of attributes I use (in no particular order) are:

- *Correctness*: Does the code do its job?
- *Modifiability*: How easy is the code to modify?
- *Performance*: How fast does the code run? How much memory, disk space, CPU, network usage, etc. will it use?
- *Speed of production*: How quickly will the code be constructed?
- *Reusability*: To what degree will the code be architected to allow later projects to reuse code?
- *Approachability*: How difficult is it for people who are proficient in the languages and tools used to be able to take on future maintenance tasks?

[172]http://programmer.97things.oreilly.com/wiki/index.php/Michael_Harmer

- *Process strictness*: How important is it that the nominated development process and coding procedure is followed? In other words, is anyone going to be sacked if they don't follow the identified processes?
- *Standards compliance*: How important is it to comply with the various relevant standards?

You'll note that these attributes aren't independent. Importantly, everything needs to be balanced – it's generally unwise to maximize a single attribute at the expense of others.

Remember, whether you know it or not, you will be making trade-offs between the attributes of your code. It is best if the trade-offs are carefully considered and well communicated.

---

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

[174]http://programmer.97things.oreilly.com/wiki/index.php/Talk_about_the_Trade-offs

There Is Always Something More to Learn

By Klaus Marquardt[175]

When interviewing potential software engineers, most people care for technical competence, for some degree (as a proof of the ability to finish a project), and for a social fit with the current team. Further aspects worth looking for include:

- amount and kind of project experience;
- experience in different roles and project phases;
- ability and will to learn.

The initiation and the finalization of a project are more tightly linked than you can possibly learn in class. It is extremely helpful to have project team members who can sense the outcome of certain early decisions. There seems to be no more significant factor to project success than the presence of programmers who have previously worked on successful projects. Project success is created in each single phase of the project, and in each participating role. Seasoned project engineers not only know what to do about the project, they also know how their behaviour and decisions influences other people. Knowing a role also from the outside helps to fill it more successfully and sustainable.

There is a second aspect to this experience: learning about your skills and desires. Before you have not been involved in testing, or project management, it is hard to tell whether this is something you like doing and whether you are good at it. When you have the opportunity, participate in each phase of some project, from early conception to the last episode of maintenance. The role you like best and the one you are best at are likely the same - but if not there are good reasons to choose one you are good at. The project benefits from that decision, you likely are more successful and quicker at work, and you have the opportunity to grow beyond your role.

The ability to determine a project is doomed while it is still in acquisition or in definition can be extremely helpful. But this is only the start. More helpful is the ability to make a project fail early. Even more valuable is to know what you can do about indications of doom, and how to turn it into success. Furthermore, before the software is ready, the project is not finished. Again, it is easy to know that a project may fail almost touching the finishing line. The hard part is knowing the

[175]http://programmer.97things.oreilly.com/wiki/index.php/Klaus_Marquardt

difference between the finishing line, and almost the finishing line. And what to do to bridge this gap.

So here is the career-making advice: attend successful projects. Attend lots of projects, and learn.

Join projects in whatever phase, and strive to make them successful. Take care that while you see more projects, that you actively join each phase at least once. If you happen to join a team or company that fails to complete projects: see what you can learn, gather experience, and leave. There is a lot of learning in failure - but you need your opportunity to join a successful project, and to join each phase of a successful project.

> She knows there's no success like failure And that failure's no success at all. Bob Dylan, "Love Minus Zero/No Limit"

---

[176]http://creativecommons.org/licenses/by/3.0/
[177]http://programmer.97things.oreilly.com/wiki/index.php/There_Is_Always_Something_More_to_Learn

There Is No Right or Wrong

By Mike Nereson[178]

It can be hard to consistently make the right decision. Throughout each and every day we as programmers are faced with decisions: design, implementation, style, names, behavior, relationships, abstraction, …. The list goes on and is unique for each of us.

As software developers, our goal is to produce working code that contributes to a software system or application. How we get to that working system will take one of many possible paths. On this journey to production-ready code, any number of the decisions that you will make along the way can be made differently to send you down a different path. So how is it that we can say that there is a right path and a wrong path to write some method, or that there is a right and a wrong framework, or even a right and a wrong language?

The answer, for almost anything in software development, is that there is no right or wrong path. There is no wrong variable name or naming convention. There is no wrong build or dependency management tool. And there is certainly no wrong language. **There are, however, better ways and worse ways.** That is to say, there are some decisions that you make along the way that will make your software better, and some decisions that will make it worse. But better and worse in what way?

Better and worse are subjective terms. They are biased opinions that an individual perceives based on past experience, emotion, beliefs, and sometimes ignorance. Objectively, however, *better* software is generally

- Easier to modify, whether changing existing code, removing deprecated code, or adding new code.
- More reliable with less down time and a higher degree of predictability.
- Easier to read and to understand.
- Able to provide better performance while using fewer resources.

These are facets of your software that you affect with every decision that you make. These goals are what you strive to achieve with each and every decision that you make.

So the next time you are faced with making a decision, try your best to understand all of your options. Try to determine how each possible solution will affect these goals and your own unique software goals. Then, don't try to choose the *right* option, but choose the *better* option. Similarly, when team members, both past and present, make a decision, try not to judge their decision as right or wrong.

[178] http://programmer.97things.oreilly.com/wiki/index.php/Mike_Nereson

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

[180]http://programmer.97things.oreilly.com/wiki/index.php/There_Is_No_Right_or_Wrong

There Is No Such Thing as Self-Documenting Code

By Carroll Robinson[181]

Source code that does not contain comments is sometimes said to be "self-documenting." In reality, there is no such thing. All programs require comments.

In an ideal development environment, coding standards and code reviews create a culture that enforces good commenting practices (and usually high-quality code as well). For less-than-ideal environments, where the programmer sets his or her own commenting standards, the following guidelines are suggested.

Comment to supplement the source code.

Source code is written not only to be processed by the compiler, but also to communicate to other programmers. Comments should add value by supplementing the reader's understanding of the code. Avoid cluttering the code with redundant information, as in the following example, where the comment communicates nothing beyond what the programming statement specifies:

```
// Add offset to index
index += offset;
```

Change the comment so that it explains how the instruction helps to accomplish the larger task at hand. Or, change the names of the variables to make the operation more clear, so that a comment is not needed.

Comment to explain the unusual.

Sometimes, an unusual coding construct is needed to implement an algorithm or to optimize for time or space constraints. In these cases, comments can be quite valuable. For example, if a function processes an array from highest index to lowest, when all of the other functions process the same array in the opposite order, then provide a comment to explain why the function requires a different approach.

[181] http://programmer.97things.oreilly.com/wiki/index.php/Carroll_Robinson

Comment to document hardware/software interactions.

The closer an application is to controlling underlying hardware, the more comments generally are needed to make the program understandable. Source code for device drivers or embedded applications benefits from comments that describe hardware interactions, such as special timing or sequencing requirements. Some hardware-related source code may contain memory-mapped structures with cryptic bit-field names that are defined by the integrated circuit manufacturer; comments help to clarify these identifiers. Sometimes, a section of code is written to work around a hardware problem. A comment indicating the problem and referencing relevant hardware errata is useful, especially when the hardware problem is fixed and the maintenance programmers want to figure out which instructions can be simplified or removed.

Comment with maintenance in mind.

Balance the added value of each comment with its maintenance cost. Remember that every comment becomes part of the source code that must be maintained. When comments do not add to the reader's understanding, they are a useless burden. When comments contradict the source code, they create confusion for future programmers. Is the code correct and the comment wrong? Or, does the comment reflect the intent of the programmer, and the code contain a bug that was not exposed during testing?

In summary, self-documenting source code is a worthy, yet unattainable goal. Strive for it by coding with clarity, but recognize that you can never fully achieve it. Add comments to enhance the understanding of your code, while not commenting what is obvious from reading the code itself. Just be mindful that what is obvious to you, when you are deep in the details of writing the program, may not be obvious to someone else. Provide comments to indicate the things that another programmer would want to know when reading your code for the first time. The programmers who inherit your code will appreciate it.

---

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

[183]http://programmer.97things.oreilly.com/wiki/index.php/There_Is_No_Such_Thing_as_Self-Documenting_Code

The Three Laws of Test-Driven Development

by Uncle Bob[184]

The jury is in. The controversy is over. The debate has ended. The conclusion is: *TDD works*. Sorry.

Test-Driven Development (TDD) is a programming discipline whereby programmers drive the design and implementation of their code by using unit tests. There are three simple laws:

1. You can't write any production code until you have first written a failing unit test.
2. You can't write more of a unit test than is sufficient to fail, and not compiling is failing.
3. You can't write more production code than is sufficient to pass the currently failing unit test.

If you follow these three laws you will be locked into a cycle that is, perhaps, 30 seconds long.

Experienced programmers' first impression of these laws is that they are just *stupid*. But if we follow these laws, we soon discover that there are a number of benefits:

- **Debugging**. What would programming be like if you were never more than a few minutes away from running everything and seeing it work? Imagine working on a project where you *never* have several modules torn to shreds hoping you can get them all put back together next Tuesday. Imagine your debug time shrinking to the extent that you lose the muscle memory for your debugging hot keys.
- **Courage**. Have you ever seen code in a module that was so ugly that your first reaction was "Wow, I should clean this." Of course your next reaction was: "I'm not touching it!" Why? Because you were *afraid* you'd break it. How much code could be cleaned if we could conquer our fear of breaking it? If you have a suite of tests that you trust, then you are not afraid to make changes. *You are not afraid to make changes!* You see a messy function, you clean it. All the tests pass! The tests give you the courage to clean the code! Nothing makes a system more flexible than a suite of tests – nothing. If you have a beautiful design and architecture, but have no tests, you are still afraid to change the code. But if you have a suite of high-coverage tests then, even if your design and architecture are terrible, you are not afraid to clean up the design and architecture.

[184]http://programmer.97things.oreilly.com/wiki/index.php/Uncle_Bob

- **Documentation**. Have you ever integrated a third party package? They send you a nice manual written by a tech writer. At the back of that manual is an ugly section where all the code examples are shown. Where's the first place you go? You go to the code examples. You go to the code examples because that's where the *truth* is. Unit tests are just like those code examples. If you want to know how to call a method, there are tests that call that method every way it can be called. These tests are small, focused *documents* that describe how to use the production code. They are *design documents* that are written in a language that the programmers understand; are unambiguous; are so formal that they *execute*; and cannot get out of sync with the production code.
- **Design**. If you follow the three laws every module will be *testable* by definition. And another word for *testable* is *decoupled.* In order to write your tests first, you have to decouple the units you are testing from the rest of the system. There's simply no other way to do it. This means your designs are more flexible, more maintainable, and just cleaner.
- **Professionalism**. Given that these benefits are real, the bottom line is that it would be *unprofessional* not to adopt the practice that yields them.

---

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

[186]http://programmer.97things.oreilly.com/wiki/index.php/The_Three_Laws_of_Test-Driven_Development

Understand Principles behind Practices

By Steve Berczuk[187]

Development methods and techniques embody principles and practices. Principles describe the underlying ideas and values of the method; practices are what you do to realize them.

Following practices without deep understanding can allow you to try something new quickly. By forcing yourself to work differently you can change your practices with ease and speed. Being disciplined about changing how you work is essential in overcoming the inertia of your old ways. Practices often come first.

Over time you will discover situations where a practice seems to be getting in your way. That is the time to consider varying the practices from the canon. You need to be careful to distinguish between cases where the practice is truly not working in your situation, and cases where it feels awkward just because it is different. Don't optimize before you understand why the current way is not working for you. Understanding the underlying principles allows you to make decisions about how to apply a practice. For example, if you thought that the reason for pair programming was to save money on computers, your approach would be quite different than if you looked at pairing for the benefit of real-time code reviews.

Following a practice without understanding can lead to trouble too. For example, Test-Driven Development can simplify code, enable change, and make development less expensive. But writing overly complicated or inappropriate tests can increase the complexity of the code, increasing the cost of change.

Being excessively dogmatic about how things are done can also erode innovation. In addition to understanding the principles behind a practice, question whether the principles and practices make sense in your context, but be careful: trying to customize a process without understanding how the principles and practices relate to each other can set you up for failure. The clichÃ©d example is "doing XP" by skipping documentation and doing none of the other practices.

When trying something new:

- Understand what you're trying to accomplish. If you don't have a goal in mind when trying a new process, you won't be able to evaluate your progress meaningfully.

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

- Start by following best practices as close to “the book” as possible. Resist the temptation to customize early; you risk losing the benefits of a new way of working, and of reverting to your old ways under a new name.
- Once you have had some experience, evaluate whether your execution of the practices are in line with their principles and, if they are, adapt the practices to work better in your environment.

---

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

[189]http://programmer.97things.oreilly.com/wiki/index.php/Understand_Principles_behind_Practices

Use Aggregate Objects to Reduce Coupling

By Einar Landre[190]

Despite that fact that architects and developers know that tight coupling leads to increased complexity, we experience large object models growing out of control on an almost daily basis. In practice, this leads to performance problems and lack of transactional integrity. There are many reasons why this happens: the inherent complexity of the real world, which has few clear boundaries; insufficient programming language support for dynamic multi-object grouping; and weak design practices.

To illustrate the problem, consider a simple order management system built from three object classes: `Order`, `OrderLine`, and `Item`. The object model can be traversed as `order.orderLine.item`. Assume now that the item price must be updated. What should happen to confirmed and delivered orders? Should their price also be changed? In most cases, certainly not. To secure that rule, the item price at the time of purchase can be copied into `orderLine` together with the quantity. Another example is to think about what happens when an order is canceled, and its corresponding object deleted. Should attached `orderLines` be deleted? Most likely, yes. Should the referenced items be deleted as part of an order? Most likely not.

Dealing with this type of problem at large leads us to a set of design heuristics first published by Eric Evans in his book Domain-Driven Design[191] under the heading *aggregates*. An aggregate is a set of objects defined to belong together and, therefore, should be handled as one unit when it comes to updates. The pattern also defines how an object model is allowed to be connected, with the following three rules considered to be the most important:

1. External objects are only allowed to hold references to the *aggregate root*.
2. Aggregate members are only allowed to be accessed through the *root*.
3. Member objects exist only in context of the *root*.

Applying these rules on our simple order management system the following can be stated: `Order` is the *root* entity of the order aggregate while `orderLine` is a member. `Item` is the *root* of

[190]http://programmer.97things.oreilly.com/wiki/index.php/Einar_Landre
[191]http://domaindrivendesign.org/books/index.html

another aggregate. Deleting an order implies that all of its `orderLines` are deleted within the same transaction. Items are not affected by changes to orders. Orders are not affected by changes to items.

Identifying aggregates can be a difficult design task, with refactoring and domain expertise a must. On the other hand, the aggregate pattern is a powerful tool to reduce coupling, taking out enemy number one in our struggle for good design.

---

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

[193]http://programmer.97things.oreilly.com/wiki/index.php/Use_Aggregate_Objects_to_Reduce_Coupling

Use the Same Tools in a Team

By Kai Tödter[194]

We all love our tools. And many of us think that we use the *best* tools and that our tools are more effective and better than the tools our co-workers are using. But, when you work in a team, the team is often more effective if there is some consistency across the team:

- Exactly the same tool chain. In other words, exactly the same version of the same IDE, same compiler, etc.
- Exactly the same coding conventions and coding style, which is also backed by the tools we use.
- Exactly the same process of contributing code. For example, cleaning up the code using the team's coding conventions and styling before checking code in.

But why should you use the same tool set? Today's development tools are powerful and complex. Using the same tool set brings many benefits to the team: If everyone uses the same tools, everyone can coach and help other team mates master these tools. The team members are able to focus more on the problem domain instead of struggling with different tools alone. If someone discovers a hidden gem in one of the tools, such as a handy shortcut or reporting feature, the whole team can benefit from the find. Similarly if a team member finds a workaround for a known tool bug.

When trying to introduce a tool to the team, try make a convincing and objective case to the team why this tool would be a good addition to the tool chain for the project. Be open to alternative suggestions. At the end of the day, every team member should agree on the tool chain and have the feeling that this is more like a team decision rather than an imposed decision. After a while, collaboration of team members will automatically improve and they might see an increase of overall productivity.

But often there are team members that still keep using their own tools rather than the tools the team agreed on. It is very important for the team effort to get these team members on board. There are many ways to accomplish this: Let them explain why their tool is better and why they can't use the team tool. If they convince the team, then switch to their tool. If they don't, try to find a migration path. For example, if they want to use their specific tool, let them try to configure that tool to accept the same input and produce exactly the same output compared with the team tool. But at the same

[194]http://programmer.97things.oreilly.com/wiki/index.php/Kai_T%C3%B6dter

time, you and other team members should try to convince those using different tools to use the team-agreed tools.

Make the team tools easily accessible and installable. It helps a lot if you provide a script or another mechanism that automates the installation of the complete tool environment for a project. Once you have this mechanism in place, it is very easy for a new team member to start being productive. Also, keep the tooling up to date. Provide a simple update mechanism for all team members to make sure that everybody is always using the latest and greatest tool chain for the project.

---

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

[196]http://programmer.97things.oreilly.com/wiki/index.php/Use_the_Same_Tools_in_a_Team

Using Design Patterns to Build Reusable Software

by Vijay Narayanan[197]

Design patterns are proven solutions to design problems within a given context. They help developers capture meaningful abstractions, add design flexibility, and encapsulate common behavior within their domain. Here are a few design patterns that are helpful in the context of building reusable assets:

Strategy – Polymorphically bind implementations to execute a particular algorithm. This is useful when encapsulating product variations, i.e., multiple flavors of a feature. For instance, searching a product catalog feature could use exact text matching for product A while using fuzzy matching for product B. This could also be used to support multiple flavors within the same product, choosing a concrete implementation based on runtime criteria, such as type of user or the nature of input parameter or volume of potential results.

Adapter – Used for translating one interface into another to integrate incompatible interfaces. Introduce an adapter when reusing a legacy asset. It is also useful when there are multiple versions of a reusable asset that you need to support in production. A single implementation can be reused to support both the new and the old version. An adapter can be used for supporting backward compatibility.

Content-Based Router – Used for routing messages that are being sent via a reliable messaging channel. The router can parse a portion of the message and invoke appropriate message handlers using metadata contained in the original message. Content-based routers can facilitate reuse of message handlers across transports (such as JMS and HTTP). They can also invoke rules depending on message characteristics.

Abstract Factory – Used to create coherent families of objects. You also don't want your calling code to know the nuances and complexity of constructing domain objects. This pattern is very useful when creating domain objects that require business rules. You want the complex creation in a single place in your codebase. I typically use this pattern when supporting bundling of product features. If you want to reuse a set of product features across bundles you will have several objects that all need to be carefully chosen at runtime for consistent behavior.

[197]http://programmer.97things.oreilly.com/wiki/index.php/Vijay_Narayanan

Decorator – Used with a core capability that you want to augment at runtime. Decorators are pluggable components that can be attached or detached at runtime. The upside is that the decorators and the component being decorated are both reusable, but you need to watch out for object proliferation. I use this pattern when the same data needs to be formatted or rendered differently or when I have a generic set of data fields that needs to be filtered based on some criteria.

Command – Used to encapsulate information that is used to call a piece of code repeatedly. Can also be used to support undo/redo functionality. The commands can be reusable across different product interfaces and act as entry points to invoke backend logic. For instance, I have used this pattern to support invocation of services from a self-service portal and interactive voice response channels.

These are just a few examples of patterns that help with reuse. When you get a problem, pause and reflect to see if a pattern is applicable.

---

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

[199]http://programmer.97things.oreilly.com/wiki/index.php/Using_Design_Patterns_to_Build_Reusable_Software

Who Will Test the Tests Themselves?

By Filip van Laenen[200]

The Roman poet Juvenal posed the question in one of his satires: *Quis custodiet ipsos custodes?* (Who will guard the guards themselves?) When we're writing tests, we should ask the question to our selves too: Who (or what) will test the tests we're writing? In practice, the third law of Test-Driven Development (TDD) – you can't write more production code than is sufficient to pass the currently failing unit test – isn't as easy to follow as it may seem.

Let's consider a simple case: finding the largest element in an array of integers. We can start with a simple unit test, like this:

```
def test_return_single_element
    assert_equal(1, max([1]))
end
```

Then we write a method doing just that:

```
def max(array)
    return array.first
end
```

The next unit test could be that in an array with two integers, say `[1, 2]`, the method should return the larger one, in this case `2`. At this point many programmers will go and implement the complete method, maybe like this:

[200] http://programmer.97things.oreilly.com/wiki/index.php/Filip_van_Laenen

```
def max(array)
    result = array.first
    array.each do | element |
               if (element > result)
                  result = element
               end
    end
    return result
end
```

In fact, if we run a tool that measures the code coverage, it will indicate test coverage is 100%. But does this mean that we're done?

If we don't consider the cases of `null` arguments or empty arrays for a moment, our method is complete and correct. But if we run a mutation tester against our source code using these two unit tests only, we will find out something is wrong. Indeed, if we remove the condition of the `if` statement (by setting it to `true`, for instance), the two unit tests will still run fine. What happened?

Well, we broke the third law of TDD. We shouldn't have implemented the complete method yet, but first changed the body of the method to `return array.last`, then written a third unit test using `[2, 1]` as test data, and only then programmed the whole method. We were, however, too eager to start programming, and probably already had the third unit test running in our head. That's also why we were so surprised that all the unit tests were still running fine, even though the implementation obviously was incomplete.

What can we do to avoid situations like this? As is so often the case in our profession, the computer can help. There exist special tools, such as the already mentioned mutation testers, that can go through our source code, make small changes, and then check back whether all our unit tests are still running fine. If we meticulously followed the TDD laws, then for every change the mutation tester makes in our source code we should find that at least one unit test fails. If it doesn't, we've done something wrong – or, rather, too much.

Use mutation testers with caution, though. If used blindly and excessively, mutation testing can quickly become very time consuming, thereby losing its value. Use it primarily on the most important parts of your code, and remove false positives through continuous configuration. But make sure you don't remove the interesting mutations it generates, in particular those you don't understand, the ones you believe would never break any of your unit tests. They are the interesting ones: They will reveal where you've done more than one thing at a time, and teach you how to slow down and start writing better unit tests.

---

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

Retrieved from http://programmer.97things.oreilly.com/wiki/index.php/Who_Will_Test_the_Tests_-Themselves?[202]

[202]http://programmer.97things.oreilly.com/wiki/index.php/Who_Will_Test_the_Tests_Themselves%3F

Work with a Star and Get Rid of the Truck Factor

By Cecilia Sjolin[203] and Ida Hveding Huse[204]

Introducing pair programming into an agile software project helps to increase the productivity of the team. By continuously rotating pairs and tasks, knowledge of the whole project is spread across all team members in a fast and efficient way. Therefore, the team gets rid of the *truck factor* where important knowledge is possessed by only by a few key people.

Pair programming may sound scary in the beginning, because each programmer potentially exposes their vulnerabilities to someone while working closely with them. However, the discussions pair programming brings out help solve problems faster and better. Pair programming also leads to ongoing code review. Pair programming is an excellent technique for training up new team members. The new team member is not left alone and can start developing (and be productive) right away. For experienced programmers, one challenge with pair programming might be patience with younger and less-experienced programmers. Pair programming, however, forces all team members to challenge themselves in a safe setting and give team members the chance to both broaden and deepen their skills.

One of our experiences, however, is that team members can become overcommitted to fulfilling the task or user story they have started. We have therefore introduced a two-day rotation rule and a rotation star to hold a record of the rotation.

Two-day rule – each pair splits up every day, with the one who has stayed longest working on the task rotating out of the pair to a new task, i.e., each team member rotate task at least every second day.

Rotation star – visualize which team member has worked with whom, with one line representing one day of pair programming between team members.

At the start of iteration, the rotation star includes only the team members' names, and at the end of iteration the star should look like Figure 1. This implies that the whole team has gained knowledge about all user stories within the iteration, and competences have been spread throughout the team.

[203]http://programmer.97things.oreilly.com/wiki/index.php/Cecilia_Sj%C3%B6lin
[204]http://programmer.97things.oreilly.com/wiki/index.php/Ida_Hveding_Huse

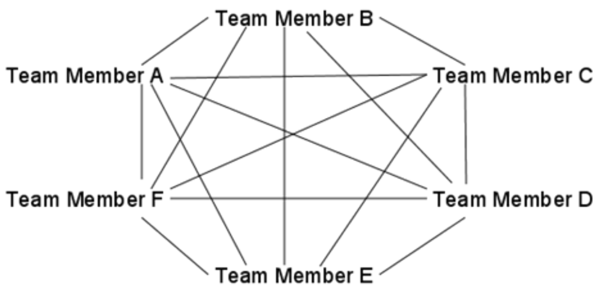

Figure 1. Recommended rotation star to have at the end of a sprint.

If the rotation star looks like Figure 2 at the end of the iteration, the team has failed to involve all its members on all of its tasks. For instance, the curved lines indicate that team members B and F have worked a lot alone most likely on the same task. Figure 2 also illustrates that team members A and B probably have worked together on the same task several days in a row.

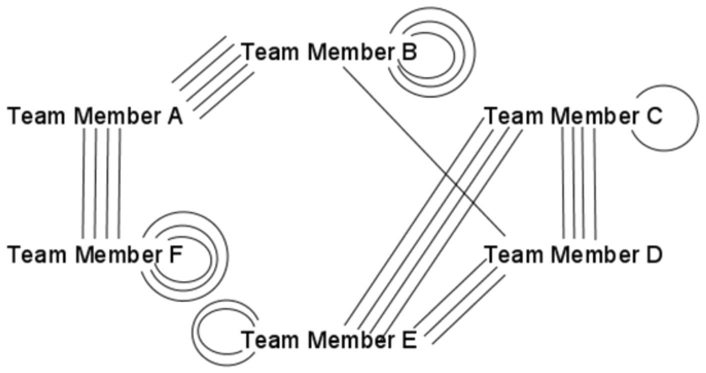

Figure 2. Not a recommended rotation star to have at the end of a sprint.

When a new team member is introduced, we recommend rotating pairs less frequently. In the beginning, it might be confusing rotating to a new task every second day.

The two-day rule and the rotation star ensure project progress regardless of illness or vacation, and therefore the team becomes less vulnerable to unexpected incidents – the team reduces the truck factor.

---

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

[206]http://programmer.97things.oreilly.com/wiki/index.php/Work_with_a_Star_and_Get_Rid_of_the_Truck_Factor

Write a Test that Prints PASSED

By Kevin Kilzer[207]

Some years ago I was writing programs to test circuit boards in an assembly line to prepare them for final assembly. The boss was very brief and clear: Write a program that tests the new product and prints “PASSED.” It seemed obvious (to me at least), that if I actually followed his instructions to the letter, the production yield would be 100%, we would realize a major corporate goal, and my boss would receive high praise. I also knew from experience that the boss did not comprehend the complexity of the problem, had not allocated enough time to properly deal with production failures, and that I (not he) would be the fool when the charade was exposed.

Practical test programs normally print “FAIL” if the product does not pass test, sometimes in large pink letters. The test operator separates the units based on this simple display, which would be OK if the work ended there. Since no one wants to throw away something after making an investment, failed units are consigned to a test technician who, with soldering iron in hand, locates bad connections and produces a “factory refurbished” unit. The guy starts with the same test program, and yes indeed, it does print “FAIL,” but he is no closer to knowing which connection is bad, or even what area of the board has the problem. It could be a hidden problem beneath a component, or just that the LED was installed backwards.

This illustrates the need for good error messages. The sanity of the test technician depends on rapidly finding and repairing the problem, but the program must guide him to that end. A production test is normally a sequence of evermore focused tests, so at least indicate which test failed, and then provide documentation that allows the technician to look up possible causes. A good technician will very quickly memorize the list, becoming quite adept at correcting faults. A better approach is to build that information into the test program, actually making suggestions on the operator’s screen. As production ramps up (because your test program works so well), new technicians can be trained very quickly.

The idea can be applied within code as well. Many academic code examples simply return 0 or 1 at the end of a function, and then propagate this up the call stack, leaving the top level to print vague “access failed” or “RPC error” messages. A better plan is to use small integers as error indicators, with 0 as the no-error code because there is only one way of stating that all is well. Beware though, because messages like “Error 7 in PutMsgStrgInLog,” can leave non-programmers bewildered. The message should use words that the operator can relate to, and you must know your audience to decide if technical jargon or embedded names are OK. Unclear or confusing error messages can get

[207] http://programmer.97things.oreilly.com/wiki/index.php/Kevin_Kilzer

you a call from an irate third-shift manager who “needs you at the plant immediately to sort out this stupid program.”

Specific error messages for every issue make debugging go much faster. Integers and pointers take the same amount of storage, so instead of 0 or 1, return a pointer to a string. It requires no additional code to test if an integer is zero or a pointer is null, but the benefit can be dramatic. When an error is reported, you can show meaningful messages like “The data server connection is not working” or “Oscillator signal is not present.” The exception mechanisms in languages like C# and Java allow for passing strings, but not every embedded coder has that luxury, and someone working on a system written in C has already had their language choice made for them.

---

[208]http://creativecommons.org/licenses/by/3.0/us/
[209]http://programmer.97things.oreilly.com/wiki/index.php/Write_a_Test_that_Prints_PASSED

Write Code for Humans not Machines

By Mario Fusco[210]

Programmers spend more their time reading someone else's code than reading or writing their own. This is why it is important that whoever writes the code pays particular attention not only to what it does but also to how it does it. For a compiler, it makes no difference if a variable is called `p` or `pageCounter`, but of course it makes a big difference for the programmer who has to figure out what kind of information that variable contains. That is why it is easier to write code for compilers than for people.

Variable and method names should be chosen carefully so as to leave no doubt about their meaning in the reader's mind. The names should most likely be taken from the objects and actions typical in your domain. If you feel the need to add a comment to clarify their purpose or behavior, it could be the first symptom that you should spend a minute more to find a more self-describing name. A comment on a method is not necessarily bad, but a meaningful name is still the best place to start: The comment will be available only on the method declaration while its name is the only thing you can read wherever that method is used.

Of course, well-chosen names are not the only things you need to make your code readable. Other rules can be applied to improve the code readability:

- *Keep each method as short as possible*: 15 lines of code is a reasonable upper limit that you should be wary of exceeding.
- *Give each method a single responsibility*: If you are trying to give a meaningful name to the method and you find the name contains an and</code>, there is a good chance that you are breaking this rule.
- *Declare methods with the lowest number of parameters possible*: If you need more than 3 parameters it could be a good idea to do a small refactor by grouping them as properties of a single object.
- *Avoid nested loops or conditions where possible*: You can improve both readability and reusability by putting them in little separated methods.
- *Write comments only when strictly necessary and keep them in sync with the code*: There is nothing more useless than a comment that explains what you can easily read from the code or more confounding than a comment that says something different from what the code actually does.

[210]http://programmer.97things.oreilly.com/wiki/index.php/Mario_Fusco

- *Establish a set of shared coding standards*: Programmers can understand a piece of code faster if they don't encounter unexpected surprises while reading it.

By making your code easily readable by other programmers you are making their job simpler. And this is no bad thing when you consider that the next programmer to read the code could be you.

---

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

[212]http://programmer.97things.oreilly.com/wiki/index.php/Write_Code_for_Humans_not_Machines