The Pragmatic Bookshelf

PragPub

The First Iteration

IN THIS ISSUE

* Maik Schmidt builds an Arduino Video Game System
* Aaron Bedra introduces ClojureScript
* Trevor Burnham explores Event-Driven CoffeeScript
* Tim Ottinger and Jeff Langr promote Virtuous Code
* Brian Tarbox ponders Code As Dialog
* Dan Wohlbruck recalls the Origins of HyperCard, the Web, and Grunge
* ..and John Shade sounds off

Issue #26
August 2011

Contents

FEATURES

DEPARTMENTS

Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail support@pragprog.com, phone +1-800-699-7764. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

Up Front

Hardware Hacking and JavaScript Alternatives

by Michael Swaine

We're looking at CoffeeScript and ClojureScript in this issue, as well as breaking out the wire cutters to build a video game machine.

This issue includes articles on both CoffeeScript and ClojureScript, two new languages that allow you to write JavaScript without having to write JavaScript, if you follow me.

CoffeeScript was immediately embraced by the great majority of developers who looked at it, while ClojureScript is being viewed more skeptically. I think this is at least in part because of confusion about what ClojureScript is for. It's really not trying to do what CoffeeScript is trying to do. (The starting point for any evaluation of ClojureScript is the official rationale [U1].)

The articles are written by Trevor Burnham, who wrote the book on CoffeeScript [U2], and Aaron Bedra, who is cowriting Programming Clojure, Second Edition [U3], which will include coverage of ClojureScript. We hope this issue will encourage people on the fence about one or the other of these tools to explore both and see what problem each is intended to solve. You might decide that both have a place in your toolkit.

You may have another kind of toolkit, one with a soldering iron and wire cutters. Hardware hacking is enjoying a renaissance thanks to the popular Arduino single-board computers. This month our Arduino guru Maik Schmidt, author of Arduino: A Quick-Start Guide [U4], returns with another Arduino project, this time showing you how to build your own video game machine for rediscovering the innocent fun of classic games like Asteroids and Breakout.

Tim Ottinger and Jeff Langr are back with some agile advice on making your code virtuous. In doing so, they shuffle their Agile in a Flash [U5] cards and draw out card #42, which you have to figure has the answer to everything [U6].

In fact, all of our writers this month are familar to these pages. Dan Wohlbruck is back with another history article, both Brian Tarbox and Andy Hunt get insights into programming from theater, and John Shade wants to talk to your mother.

Enjoy the issue and be sure to read all the way through to the end to see what's in the pipeline for next month. Oh, and for those of you in the Northern Hemisphere, we hope the cover images makes you feel cool.

External resources referenced in this article:

[U1] https://github.com/clojure/clojurescript/wiki/Rationale

[U2] http://pragprog.com/refer/pragpub26/titles/tbcoffee/coffeescript

[U3] http://www.pragprog.com/refer/pragpub26/titles/shcloj2/

[U4] http://pragprog.com/refer/pragpub26/titles/msard/arduino

[U5] http://www.pragprog.com/refer/pragpub26/titles/olag/Agile-in-a-flash

[U6] http://en.wikipedia.org/wiki/Phrases_from_The_Hitchhiker's_Guide_to_the_Galaxy

How Virtuous Is Your Code?

Can there be a timeless and universal definition of "good" for software?

by Tim Ottinger, Jeff Langr

That code you thought was so good last year doesn't look so virtuous today. Were you just wrong then, or has the meaning of "good" changed?

Want to get in an argument with a developer? Tell them their code isn't very good. The hackles rise, adrenaline kicks in, and then you get that "how dare you" look. But even though we may defend the quality of our code today in passionate arguments and reasoned apologetics, three years in the future—or even three weeks later—we may look back on it with embarrassment.

Disgust with past code is the sure indicator that we've learned since then.

We've been through myriad definitions of "good code" in the past, some defensible and some quite dubious:

- Good code is code that gets the right answer
- Good code is written perfectly to spec
- Good code is written defensively
- Good code uses the language's features the most elegantly
- Good code is expressive
- Good code complies to local style guides
- Good code is well-commented
- Good code uses patterns
- Good code can be read by non-programmers
- Good code passes its tests
- Good code is anything not written here

We find that our definition of "good" doesn't always transfer as we move from statically typed languages to dynamic, from procedural languages to functional, from waterfall methods to agile, or even between contracting and product development. Perhaps we were good at writing code perfectly "to spec," but now that our needs are always changing, we collaborate instead of depending on detailed specs. Maybe our code was highly defensive, and now we see such overzealous effort as waste. Did we change, or did "good" change? Were we really writing good code to begin with, or was it self-congratulatory nonsense?

Can there be a timeless and universal definition of "good" for software? It's like looking for the answer to life, the universe, and everything, but Agile in a Flash [UI] offers a possible answer with card #42 (hat tip to Douglas Adams).

We present each virtue in two parts: the virtue and its opposite. Why? Because the words on the left side are not very crisp in common usage. By providing the opposite for each virtue, we help programmers to arrive at our intended meaning quickly and clearly.

42 The Seven Code Virtues

- Working, as opposed to incomplete
- Unique, as opposed to duplicated
- Simple, as opposed to complicated
- Clear, as opposed to puzzling
- Easy, as opposed to difficult
- Developed, as opposed to primitive
- Brief, as opposed to chatty

Working, As Opposed to Incomplete

That code should work seems to state the glaringly obvious. Yet *working* is a virtue that generates the most heat as the discussion continues. Here is our favorite formulation of the rule:

"Code that works now is superior to code that may work some day."

Perhaps you have diagrams of an astrotecture for a future system that, once written, will never have to change again. However cleverly considered, this planned system is far inferior to the prototype your neighbor has already written over the weekend. Perhaps the brilliantly designed and planned software will be superior when it is built, perhaps not. Perhaps by the time the planned system is built, the prototype will have evolved through use and feedback into an amazingly useful program that leads the market for decades. All we can say is that working code now is better than code that is planned to work someday.

Consider also:

"Code continually proven to work is better than code that may once have worked."

Code that is written without a solid suite of tests may work, or may have worked once, but virtue is more clearly displayed by code that passes a battery of tests many times per day. Automated tests not only demonstrate this virtue, but in the course of writing tests programmers usually have to improve the simplicity and clarity of code, not to mention the removal of duplication. We have become so accustomed and even habituated to TDD that we tend to see untested code as incomplete.

A real-time system is one in which timeliness is a component of correctness. In a real-time system, there is a barrier at which being one microsecond later will render an answer useless or wrong. When the time budget is so tight that it strains our software technology and computer architecture, dire measures may be required to meet computational deadlines. In such a situation, working may be the one virtue that matters. In more workaday situations, however, working is just a starting point.

Sadly, some programmers assemble code from various files in the project, reach some semblance of a working state, and then walk away. This is programming

at its lowest, basest state. Once your code works, it becomes a matter of raising the signal-to-noise ratio in ways that are useful to the team. To help guide the improvement process, we provide the remaining six virtues.

Unique, As Opposed to Duplicated

Code wants to be a unique little flower in the universe, just like most people do. Code that's replicated like stubborn little dandelions throughout your system not only detracts from its beauty, but is very difficult to eradicate once you let it grow rampant. Pluck the dandelion by its head—clean only the surface of the code problem—and it ultimately grows back, and stronger too.

Now that we've exhausted the analogy, let's look at some duplication issues and then see how we might resolve them.

In a popular plugin for a CI tool, the Python code has five near-identical copies of these seven source lines:

```
if self.key is None:
    raise APIKeyError("Your have not set an API key.")
if data is None:
    data = {}
if build_data:
    self._build_data(comment, data)
url = '%ssubmit-spam' % self._getURL()
```

Programmers copy and paste for many reasons[U2]. The jump-start that copying code gives us may make code duplication [U3] seem like a virtue, but we find the goodness of copying ends before you check in, and the trouble it causes extends years into the future. A change to copied code (in our example, imagine a new choice of exception) will have to be made to many places, and missing one might have expensive repercussions.

Because duplication is so easy to create, so rampant in the typical system, and so costly, we list the *unique* virtue second only to *working*.

What about the following chunk of code?

```
data.setdefault('referrer', os.environ.get('HTTP_REFERER', 'unknown'))
data.setdefault('permalink', '')
data.setdefault('comment_type', 'comment')
data.setdefault('comment_author', '')
data.setdefault('comment_author_email', '')
data.setdefault('comment_author_url', '')
data.setdefault('SERVER_ADDR', os.environ.get('SERVER_ADDR', ''))
data.setdefault('SERVER_ADMIN', os.environ.get('SERVER_ADMIN', ''))
data.setdefault('SERVER_NAME', os.environ.get('SERVER_NAME', ''))
data.setdefault('SERVER_PORT', os.environ.get('SERVER_PORT', ''))
data.setdefault('SERVER_SIGNATURE', os.environ.get('SERVER_SIGNATURE', ''))
data.setdefault('SERVER_SOFTWARE', os.environ.get('SERVER_SOFTWARE', ''))
data.setdefault('HTTP_ACCEPT', os.environ.get('HTTP_ACCEPT', ''))
data.setdefault('blog', self.blog_url)
```

Looks pretty good, right? *Something* is unique about every line. But what if it looked more like this:

```
for (key,value) in [
        ('permalink', ''),
        ('comment_type', 'comment'),
        ('comment_author', ''),
        ('comment_author_email', ''),
        ('comment_author_url', ''),
        ('blog', self.blog_url),
    ]:
    data.setdefault(key,value)
for (key,env,default) in [
        ('referrer', 'HTTP_REFERER', 'unknown'),
        ('SERVER_ADDR', 'SERVER_ADDR', ''),
        ('SERVER_ADMIN', 'SERVER_ADMIN', ''),
        ('SERVER_NAME', 'SERVER_NAME', ''),
        ('SERVER_PORT', 'SERVER_PORT', ''),
        ('SERVER_SIGNATURE', 'SERVER_SIGNATURE', ''),
        ('SERVER_SOFTWARE', 'SERVER_SOFTWARE', ''),
        ('HTTP_ACCEPT', 'HTTP_ACCEPT', ''),
    ]:
    data.setdefault(key,os.environ.get(env,default))
```

"Wait, that's actually longer!" Yet paradoxically there is still less to read because more of the content is unique. The for loop that unpacks the data also tells you the meanings of the columns, adding a bit of the *clear* virtue for free. (We leave further reduction of duplication here as an exercise for the reader.)

Duplication is a tough enemy. The first challenge is to spot it, not always easy in a large code base. Once you've spotted it, getting rid of it requires either a lot of tests or a strong stomach for the high risk of shipping potentially defective code. After all, we want the code to still exhibit the *working* virtue.

Simple, As Opposed to Complicated

Whenever a developer says "simple," a dozen of his peers nod their heads but very few of them have the same understanding of the word. People may mean "easy to read" or "easy to write" or "uses no advanced features." We have a different definition.

We like to think of simplicity as a measurable attribute, consisting of the number of unique names and operators in a given code component (class, method, function, subprogram, etc.). Some people have suggested that we call this virtue "structural simplicity." That works for us.

If a function has dozens of variables and hundreds of operators with dozens of paths, it is not simple even if it is clear and works. Code that is well-named is not necessarily simpler than code that is ill-named, because naming doesn't affect the number of operators and entities in the passage of code.

Likewise, copied code is usually no simpler than hand-written code. Terse code can be simple, or a plethora of operators and side-effects may make it quite complicated. Working code may be simpler or more complicated than nonworking code. These virtues are pretty much orthogonal.

Complexity may be moved, but can seldom be entirely removed from a system. Here are examples of techniques for taming complexity:

- The null object pattern can eliminate dozens and dozens of if..else statements. Fewer paths is simpler.

- Refactoring to replace switch..case statements with polymorphism can take a decision that is duplicated throughout a code base and replace it with a decision made once on object construction.
- Using the strategy pattern instead of flags can likewise reduce the number of decision points in a program.
- Grouping variables into classes and structures can result in code that manipulates many fewer symbols and is simpler even though the new class absorbs the collection of variables.
- Treating any set of two or more attributes as a list can reduce copied-and-edited sections of code with a simple loop.
- Extracting functions can move operations into a new, small, simple method and reduce the caller's complexity. This often enhances clarity and reduces duplication in addition to simplifying the caller.
- Unneeded architectural layers can be collapsed.
- An array of objects is simpler than cross-indexing a number of arrays that use the same index. It is less primitive, and yet simpler.

There are many other techniques, but these are commonly used to good effect.

Clear, As Opposed to Puzzling

Uncle Bob once mentioned the notion of a new quality metric: WTFs per minute. (We will claim the delicate expansion of the acronym here: What's This Foolishness? …) A WTF is the very opposite of clear, causing its readers to scratch their heads in puzzlement.

Here is an example of code that is not clear in its intent:

```
list1 = []
for x in theList:
  if x[0] == 4:
     list1 += x
return list1
```

And here is a version that is much clearer without being any less complex (algorithmically, in terms of symbols being manipulated):

```
flaggedCells = []
for cell in theBoard:
    if cell.isFlagged():
        flaggedCells += cell
return flaggedCells
```

The second version differs in the following ways:

- Variable names are more expressive (variable naming affects only clarity).
- A primitive integer type is replaced with a class (see *developed*, below).
- Its introduction of an explanatory function isFlagged eliminates an index and magic number comparison, making code both clearer and simpler.

There are really no good excuses for not constructing your code so that other developers can understand and maintain it easily, but we've heard plenty:

- "I'm in a mad rush."

- "I couldn't find a good example out there."
- "You'll figure out what I meant, just study it a while longer."
- "I don't really care about this section of code."
- "I was experimenting, but I didn't have the chance to reimplement it."
- "It just needs to work" (*aka* "one virtue is good enough for me").

All else being equal, at least developers don't usually try to argue that puzzling code is better than clear and obvious code. The argument is usually about the trade-off between clarity and other virtues. In such a case, it's good to experiment with naming, introducing explanatory variables or explanatory (inline) function calls. There may be ways of improving clarity that do not sacrifice other virtues at all.

Is clarity a matter of making the code more "English-like?" Not necessarily. This example *does* show improved clarity with its implementation that reads fairly well as English prose. But in our *unique* example above, we changed a step-by-step approach into one arguably less English-like and yet clearer.

Easy, As Opposed to Difficult

Software projects, agile or otherwise, may experience schedule pressure. There is a lot of eagerness among customers for new features, and developers have an eagerness to produce them. Eagerness for features and awareness of costs are healthy drivers.

We want to write, test, and release code more quickly so we can fulfill our promises, but often we don't get past the wishing phase. We don't always go to the trouble of making our systems more easily workable. For instance, what if the programmer in the first example (*unique*) had taken a little time to put his data in tables and to write subroutines instead of copying code? It would be easier to add data to the table than to add commands the old way, and it would be easier to create a new interface point if there was less code to copy and test. The payback would have been immediate, and the changes are simple, but they were "not what he was here to do."

Oddly, we find people more willing to add complexity in the form of `if` statements, `switch` statements, unnecessary design patterns, and bloated architectural ideals than to invest a fraction of that time to make code more maintainable.

A little syntactic sugar can often turn a difficult, problem-fraught job into a simple and pleasant one. When Python or Ruby metaprogramming is done well, the code is easier to write. When it's done badly, it confounds problem-solving (making it less easy).

In a system with the *easy* virtue, everyone on the team moves a little faster. It's not hard or tricky to add new code, and nobody is wrestling with obscure bugs or tedious syntax.

How much time does your entire team spend in a month to make their jobs easier?

Developed, As Opposed to Primitive

The word *developed* in this context means that a good set of supporting mechanisms has been created and groomed. (Think "well-developed.") It may be a set of functions, classes, libraries, or even servers.

In an undeveloped system, the programmer is doing everything by hand. Instead of using a list comprehension, he must write a for loop with an integer index. Instead of walking through a list of tuples, he has to correlate a number of arrays on the same index variable. Instead of creating an XML node and adding properties, he must write angle brackets and strings to a stream. In this way, a developed language system has virtue over one that is less rich in mechanism and data types.

This virtue is an enabler to the virtue of *easy*, and indeed may be a side-effect of *unique* and *simple*. One difference that makes *developed* stand out is the choice of moving complexity out of user code and into service routines.

Where the underlying system is underdeveloped, developers will often create code generators or resort to copy-and-edit programming. One can hardly blame them for not wanting to deal with primitive containers, primitive variables, and the like.

When we say *developed*, we're measuring the amount of aid a component provides, not the number of hours of effort that have been put into it. A large, complex system may have had many man-years of development activity, yet provide little aid to programmers. Such a system may actually be net-negative, costing more in frustration and development time than it saves. We find a developed system to be rich in mechanism, with just enough policy to help programmers do a good job without restricting them to preconceived use cases.

Brief, As Opposed to Chatty

If we never had to maintain code, code would need only the virtue of *working*. However, the vast majority of programming work requires some level of maintenance to existing code. As a considerate developer, then, we take care to craft our code to say exactly what we mean, nothing more. Prolix code requires more comprehension time and generally more maintenance effort.

Remember that brief does not mean cryptic, however. Brief code must remain easy, simple, and clear if it is to retain all of the necessary virtues.

Here is a semi-virtuous example from earlier:

```
flaggedCells = []
for cell in theBoard:
    if cell.isFlagged():
        flaggedCells += cell
return flaggedCells
```

Our C# friends are about to write follow-up comments telling us that this is a good place to use LINQ. Pythonistas are ready to suggest list comprehensions. We agree. Such features allow us to put code like this into a single statement:

```
return [ cell for cell in theBoard if cell.isFlagged() ]
```

The non-Python, non-LINQ, non-FP folks may find it puzzling at first glance, and then realize with a start that this terse code is actually more readable than the longer, explicit loop above.

The more terse code allows us to see the entire algorithm in a glance. We do not have to wade through a sea of variables and operators. This is the kind of brevity that we look for (and for which we expect to see many followups from functional programmers).

The virtue of brevity speaks directly to the concept of abstraction in programming. We seek for our code to quickly express its intent, by emphasizing more of *what* it is doing and deemphasizing *how* the code is accomplishing it.

Warriors Of Virtue

As developers, our first job is to make the code work. Yet our task does not end there. We accept that sometimes one virtue may be locally diminished a bit for the sake of other virtues, but code that merely works (or is merely easy to write) is not going to serve developers as well as code that is virtuous in many ways.

Our employers and clients need us to build systems that are economical to amend and improve. If we churn out great heaps of steaming refuse that worked once, we are neither serving their interests nor our own.

As programmers over the magical age of 40, having survived bubbles and downturns and recessions and corporate takeovers, we have learned that your reputation for good work is the only real job security. Writing virtuous code serves us in the long term as well as the short.

Are these virtues a universal definition of good? We think they might be, but we welcome your suggestions, arguments, improvements, and corrections.

About Tim

Tim Ottinger is the originator and co-author of Agile in a Flash [U4], a contributor to Clean Code, and a 30-year (plus) software developer. Tim is a senior consultant with Industrial Logic where he helps transform teams and organizations through education, process consulting, and technical practices coaching. He is an incessant blogger and incorrigible punster. He still writes code, and he likes it.

About Jeff

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash [U5] with Tim, he's written over 100 articles on software development and a couple books, *Agile Java* and *Essential Java Style*, and contributed to Uncle Bob's *Clean Code*. Jeff runs the consulting and training company Langr Software Solutions from Colorado Springs.

Send the authors your feedback [U6] or discuss the article in the magazine forum [U7].

External resources referenced in this article:

[U1] http://www.pragprog.com/refer/pragpub26/titles/olag/Agile-in-a-flash

[U2] http://agileotter.blogspot.com/2010/07/copy-and-edit-revisited.html

[U3] http://en.wikipedia.org/wiki/Duplicate_code

[U4] http://www.pragprog.com/refer/pragpub26/titles/olag/Agile-in-a-flash

[U5] http://www.pragprog.com/refer/pragpub25/titles/olag/Agile-in-a-flash

[U6] mailto:michael@pragprog.com?subject=virtue

[U7] http://forums.pragprog.com/forums/134