bright code, dull code

by fox peterson
@a_fox_box
Can you think of a time you’ve been here? You come across a piece of code, you are expected to be able to work with it, and you’re thinking…

... I DIDN’T EVEN KNOW VARIABLE NAMES COULD START WITH EMOJI...
not all code is created equal

Some ways we use code in science:

- **personal / exploratory**
- **internal use**
- **shared libraries/ packages**
- **software applications**

- **building** is preparing all the resources for use and possible re-use (i.e. compiling, dependency management)
- **deploying** is serving a complete archive of the software and the build files
such programming.
It is a science fact that when you deploy your code as an application, every single bug you didn’t catch will come out of it.
Well-written code (and managing it!)  

Package Management  

Continuous Integration and/or Versioning  

Well-written code (and managing it!)  

The great pyramid of software applications

If you didn’t come to the “git” workshop yesterday—hit me up after this presentation and we can talk package management! (Octocat below is the GitHub mascot)
a skeleton application

helpful hint! build one for any environment you code in! saves setup file
requirement file
installer/uninstaller (optional)
models (classes, data structs)
static files (images, etc)
views (templates, api)

YOUR MAIN APPLICATION (sometimes server and client scripts)

.config
.git
README.md
LICENSE contact info
Etsy: The Early Years

1. Weeks writing code
2. Painful merge
3. Hand off to deployers
4. Deploy, site down
5. Roll back deploy
6. Fix bugs

7. Go back to step 2

no lulz just sads
how to avoid sads

- deploy frequently and get feedback
- keep versions - use tools like git
- use package managers made by external parties - controls dependencies
- only let the user access what they need to access to run the program; let developers use the source responsibly
many tools to help

- javascript/ node.js -> npm
- python -> pip, conda
- julia -> pkg.add (built in)
- matlab -> matlab file exchange
- haskell -> cabal
- php -> composer
- css -> compass

With all these environments (and there’s always more) how do we keep up?

We grow an intuition about what makes good and bad code we can apply across all!
which cow looks sick?

bright code

dull code
Believe it or not, you are probably already attune to bad code. But we’re going to develop your code nostrils a little today. These are practices you can use to make your code more future-proof!

https://sourcemaking.com/refactoring/bad-smells-in-code
Quick List of the Principles:

1. Too many comments
2. YAGNI
3. Duplication
4. Long Methods
5. Big conditionals
6. Data clumps
7. Coupling
8. No null case
9. Leaky Details
10. Coding for the exceptions
Too many comments

- comments can be useful in programs
- too many comments brings the next users attention to implementation details
- focus on making method names clear and method actions short.
- no bonus for LONG CODE
Lines of code/commit ratio

(smaller means - you did less work to make more progress. red means “most used” in github, blue means less used, black means very uncommon/proprietary)
Bright code or dull code?

```python
# get the site code
site_code = get_site_code(site[year])

# if the site code is in Oregon or Washington
if site_code in ['OR', 'WA']:
    # show the user it is in the northwest
    print "the site is in the Pacific Northwest!"
else:
    # show the user the site is not in the Northwest
    print "the site is not in the Pacific Northwest!"
```
def print_locale(site_code):
    """print if a site is in the pacific northwest""
    site_code = get_site_code(site[year])

    if site_code in ['OR', 'WA']:
        print "the site is in the Pacific Northwest!"
    else:
        print "the site is not in the Pacific Northwest!"
YAGNI - “You Ain’t Gonna Need It”

- don’t write “features” for the sake of writing features
- features metamorphosize into bugs

+ too many features
Bright code or dull code?

// isPosInteger returns false if the value is not a positive integer, true is returned otherwise.

private static bool isPosInteger(int inputVal)
{
    return inputVal > 0;
}

Do we really need a function to tell us the truthiness of a number being positive?
Brighter code!

oh wait, there isn’t any. that feature was useless. YAGNI!
Remove duplication and extract common methods

- If code contains duplicate sections, it is hard to maintain.
- Whenever you see something that is repeated, think about how to condense it.
- Extract it and abstract it!
class RainGage(object):
    def __init__(self, data):
        self.data = data

    def print_good_data(self):
        if self.data > 0:
            print self.data

class SnowGage(object):
    def __init__(self, data):
        self.data = data

    def print_data(self):
        if self.data > 0:
            print self.data

Bright code or dull code?
class Gage(object):

    def __init__(self, data):
        self.data = data

    def print_good_data(self):
        if self.data > 0:
            print self.data
String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")){
            return "Don";
        }
        if (people[i].equals("John")){
            return "John";
        }
        if (people[i].equals("Kent")){
            return "Kent";
        }
    }
    return ";
}

String foundPerson(String[] people){
    List candidates = Arrays.asList(new String[] {"Don", "John", "Kent"});
    for (int i=0; i < people.length; i++)
        if (candidates.contains(people[i]))
            return people[i];
    return ";
}
Methods should be small

- the longer a method is, the harder it is to understand
- some say “no more than 5 lines”
- paradigm from older languages limited by memory
- fear not data structures, objects, classes

may thy source [code] be with you.
Dull Code

```python
def test_payment(self):
    # "move" payments: no transaction fee
    # me -> other account in wallet
    old_balance = self.bridge.getbalance(self.user_id)
    old_accounts = len(self.bridge.listaccounts())
    with self.bridge.openwallet():
        result = self.bridge.payment(self.user_id,
                                      self.other_user_id,
                                      self.amount_to_send)
    self.assertTrue(result)

    new_balance = self.bridge.getbalance(self.user_id)
    new_accounts = len(self.bridge.listaccounts())
    self.assertEqual(old_accounts, new_accounts)

    spent = old_balance - new_balance
    # print("Intended to send:", str(self.amount_to_send))
    # print("Actual amount (including fee):", str(spent))
    # print("Fee paid:", str(spent - self.amount_to_send))

    self.assertEqual(spent, self.amount_to_send)
    self.assertEqual(old_balance - new_balance, self.amount_to_send)
    # "sendfrom" payments: transaction fee
    # me -> outside account
    old_balance = self.bridge.getbalance(self.user_id)
    old_accounts = len(self.bridge.listaccounts())
    with self.bridge.openwallet():
        txhash = self.bridge.payment(self.user_id,
                                      self.btc_testnet_faucet,
                                      self.amount_to_send)
    self.assertNotEqual(txhash)
```

note: this is a unit-test for a bitcoin app - this code would not be in a production environment
the functions Scale/1 and Abs/0 modify a vertex structure. Scale takes 1 argument to rescale the vertex's x and y coordinates. Abs does not take arguments, and finds the absolute value
decompose conditionals to “nots”, “elses”, defaults

- when cleaning code, try to rid yourself of switches and “elseifs”.
- instead, use a default case, and test against it, or use a reference
if (date.before(SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;

if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge(quantity);
Avoid data clumps

- Extract an object instead
- If you take one part of a clump away, will the remaining part make sense?
- Implicit reliance between two variables becomes explicit
- REDUCES COUPLING... COUPLING IS BAD
It will never make sense to show start date without end date in this case, and bugs could be introduced if someone tried to only show one part...

It does make sense to separate out date_range as an attribute to keep start date and end date together, so that a class-level fail occurs if they aren’t imported right.

def find_values_in_range(startdate, enddate, data):
    return [data[today] for today in date_list if today >= startdate and today <= enddate]

class DateRange(object):
    def __init__(self, startdate, enddate):
        self.date_range = (startdate, enddate)
    def find_values_in_range(daterange, data):
        return data[today] for today in date_list if today >= daterange[0] and today <= daterange[1]
Coupling is a point of failure

- never rely on some part of the code or data to be accessible universally.
- less inputs (to functions); less coupling.
- a good evaluation tactic is to look for places with > 2 parameter inputs- what can you combine?
The Null Object Pattern

- never assume a future user will know how to put in the right information
- never assume a future programmer will know what types a function can take
- always create a null or default pattern that will return if the function is called in an invalid way
dull code or bright code?

**dull code**

```python
value = d[key]

is the same as...
```

```python
if key not in d:
    value = None
else:
    value = d[key]
```

**brighter code!

```python
value = d.get(key)

is the same as...
```

```python
value = d.get(key, None)
```
Don’t let internal details leak

- in some languages/frameworks a function will not close over itself or its namespace.
- a value may persist until changed, or not allow some future variable to be evaluated
- this is especially problematic in event driven languages and simulations
- this one is hard to detect!
look for reflection words like “this” and “self” and reflection conventions, like “_” prepended

dull code on the left does not reflect on itself with “this”

bright code on the right uses “this” to identify that it is talking about this one instance
To sum up those principles

- Reduce comments
- Remove duplication
- Avoid data clumps/coupling
- Shorten methods
- Decompose conditionals
- Use null or default objects
- Don’t let internal details leak

Notice what kinds of terms we are using---reduce, remove, shorten, avoid, decompose, etc.

Bright code has minimal complexity!
Or, as my spirit animal, “Joe Armstrong” would say

"Only program the happy case, what the specification says the task is supposed to do … the specification says what the code is supposed to do, it does not tell you what you’re supposed to do if the real world situation deviates from the specification, so what do the programmers do… they take ad hoc decisions”

Which is to say, when managing your code, focus first on making the “happy case” - Do not focus on adding features, throwing errors, or making inline comments.
But what if I have errors?

Even the most robust code can get sick and have bugs.

Cows always smell.

Code always smells.

Well, then, we need a form of vaccination!

oddly enough, this actually is part of the higher level deployment strategy we talked about earlier!
Obey the testing goat

- In almost every deployment package, you will see a folder called /tests.
- While you can write your own tests, I advise strongly you use a testing framework specific to the language you work in.
what is the testing goat?

Whether or not you like to test first or program first, ultimately… you must test. My personal preference is to write an outline for my functional tests first, so that I can constrain what functions I will write. Then, I will write each function, and write its test. Every time I run my tests, I have another test, and by the end, I have enough to cover the whole suite.
Tests can be small, unit tests for a few lines of code, or large, and include “mock ups” of database connections. It is good to test in minimal units. No dependencies.

Every language will have supported test packages.

Look up the one relevant to you, for example Python has “unittest” and Ruby has “rspec”.

The big win you get with testing is that, when you deploy your tests along with your application, the future user knows exactly what your environment was set up to do. Using a testing package means that you get all the documentation of those tests “for free”-- no need to write up your own.
In the setup file, you are specifying how your software application will be built. In some languages it is combined with the requirements file; in others not. So, regarding tests...

In the requirement file, you can say which testing package you want to use, for example, in node.js, we use something like:

```
"dependencies": {
  "jshint": "latest"
},
```

When you make your tests, you can set them up so that they use your classes and structures natively, rather than needing to rebuild them. To facilitate this, a good practice is to have a separate file containing generic classes that are implemented in your main application.
depend upon abstractions

- dull: write `<style></style>` within HTML
- bright: use CSS with class selectors (.css file, or LESS/SASS)
- dull: raw SQL specific to MS SQL
- bright: SQL abstraction (Alchemy, Sequel, etc.) so you can go between any RDBMS

Abstractions are a way to make your PACKAGE future proof.
A fun “exercise” you can train with at home
To retain this in your brain, here’s an exercise for you to try.

- WRITE! (not think, write) instructions to yourself on “how to make a sandwich”. Yes, it sounds silly.
- Then, imagine yourself doing every line of that, verbatim.
Now, go back and try to abstract out repeated actions and practices. For example, you might “open” the jam, “open” the bread, and “open” the sandwich—so you might have a function “open” that takes an argument of “thing” and returns a polymorphic function of unscrewing (jam), untying (bread), or separating (sandwich).
Summary thoughts, cont.

- Now, ask yourself, how do I know that I have made a sandwich. At what point is my task complete? Where are checks along the way? (this is testing). Write these down.
- Now, for fun, share this to another person. have them play out how they would make your sandwich if they were completely unfamiliar with sandwich making.
Summary thoughts, cont.

- Evaluate the other person briefly based on your tests. Did they win? Where did they fail?
- If you’re up to it, try writing it again.
- These little tasks will help you to notice where you naturally detail and where you miss out, and how to define tests.
Bright code, dull code

Like the veterinarian, there is no way we can ever have a great standard to obey in code. Code is always changing, and most of us use 2-10 different languages. Instead, we need to make it instinctual that we can diagnose bad code, and even vaccinate healthy applications against it!
Thanks everyone!

Please feel free to check out my git hub at DataRonin or tweet me at @a_fox_box. I’m on gmail at fox@tinybike.net and I run the domain isitpossibleonearth.com.
Resources

- #codenewbie - support community (less scary than stack overflow)
- Martin Fowler “Patterns of Enterprise Application Architecture”
- Ben Orenstein - Giant Robots podcast
- Ruby Rogues Podcast
- Ned Batchelder - created Python’s itertools
- Ray Hettinger - developed superclassing in Python
- Joe Armstrong’s blog
- Google’s GoLang documentation

Just a few places you might go to look for more hints on good practices!