Art in Code and Quality

Sophia McKeever

s.mckeever@pokemon.com

Abstract

Code. What do you think of when you hear or see that word? Maybe you think of programs or applications. Or perhaps you think of services, APIs or specific technologies. Code can represent many and more of these things but maybe there is something you're not thinking of when you read code. What if the development of code and the process of qualifying it is a creative process? What if the act of writing code is, in fact, an act of creating art and reviewing that code is an act of artistic appreciation?

This paper will take you on a journey beyond the scientific and the mathematical. A sojourn past the technical aspects of the work that you do and bring that work contextually into the heart of the humanities. This paper is an invitation for you to consider the very last thing on your mind when you consider code and the quality processes used to validate it. An invitation for you to think about your work as art.

Biography

Sophia McKeever (She/Her/Hers) is a self-taught Software Development Engineer in Test currently working at The Pokémon Company International with over ten years of experience within the Software Quality industry. Throughout her career she has positioned herself as a test automation framework architect at the various companies she's worked at including Apple Inc., Microsoft Azure, and DataSphere Technologies Inc. She holds a Certificate in Python Programming from the University of Washington's Continuing Education Program and has experience in a wide array of qualitative technologies including Selenium, qTest, mabl, and Cypress. She is an artist at heart with a deep love for digital illustration and building artistic code projects.

1 Introduction

Be you a software engineer or a quality assurance engineer you likely spend a good portion of your day reading or writing code. Chances are you are looking at pull requests, reading the code in a repository, looking at snippets on the web, or writing the code yourself. You have probably read thousands if not millions of lines of code throughout your career; given feedback on a senior engineer's code or perhaps chuckled at the mistakes of a junior engineer. Have you ever stopped, however, and taken in the true nature of what you're seeing? Have you taken a moment to appreciate the syntactical splendor on your screen and understand the magnitude of what it represents? There is the obvious answer that the code is a list of instructions to build an application or service. Although technically correct, this is not the full picture of the code you see on your screen. What you're not realizing is that when you read that code you are interpreting a pure creative work of art.

Our work as software engineers is a unique form of art that combines scientific and mathematical principles with poetic prose complete with stanzas and verses. Our product owners come to us with problems, and we return to them with solutions, but those solutions are truly unique to us individually. In fact, if you were a product owner requesting a solution to a problem from two engineers on the same team, the solutions would be very different. Their code would, in the end, produce the same result but the inner workings of their code would be unique to everyone built upon their own understanding, past experiences, and even their emotional state at the time of writing. Chances are you've seen this in action and not given it much thought. You may have been wowed by the 'beauty and elegance' of a solution or indifferent to one that seems 'run of the mill'. In those moments you are engaging in artistic appreciation for the code you are reading.

We've all had that one person on our team at some point who takes feedback and constructive criticism on their code personally. Maybe they're on your team right now, maybe you're them. Why do they fight or get upset? We often chalk that up as someone who thinks they are better that everyone else, someone who things they are above the rest. Truly think about that person. What drives them? Is it that they are stubborn or self-centered, or is it that they are an artist defending their work? One could argue that this individual needs to be able to accept constructive criticism and feedback, and this is arguably true; however, as an artist it can be hard to accept this criticism and not also feel the need to stand by your work. Constructive criticism for artists can be difficult to accept, even when it's a mutually agreed upon two-way street. Rather than thinking the programmer is self-centered, we need to think of them as an artist and better know them as a human in the context of their work.

Throughout this paper, we'll explore the philosophy behind the act of writing code as a creative process in both the methodology and aesthetic beauty of the code. Then we will take an introspective look at the influence our emotion plays in the work that we do as creative individuals. We'll learn how our emotional intelligence in addition to providing gentle, constructive, and collaborative feedback in our code reviews can help us better connect with the human behind the code. By learning how to engage our emotional IQ, we'll be able to foster integrity and respect while building relationships within our teams and partners, improve the state of quality in the products we work, and ultimately delight our customers through improved team cohesion.

2 On the Philosophy of Writing Code as a Creative Process

2.1 The Definition of Art According to the Dictionary

To understand how code is an art form we need to first define what art is. Merriam-Webster has six definitions of art as a noun but there are two definitions we should focus on. We'll start with the first definition.

"*Art (noun) - Skill acquired by experience, study, or observation.*" (Merriam-Webster)

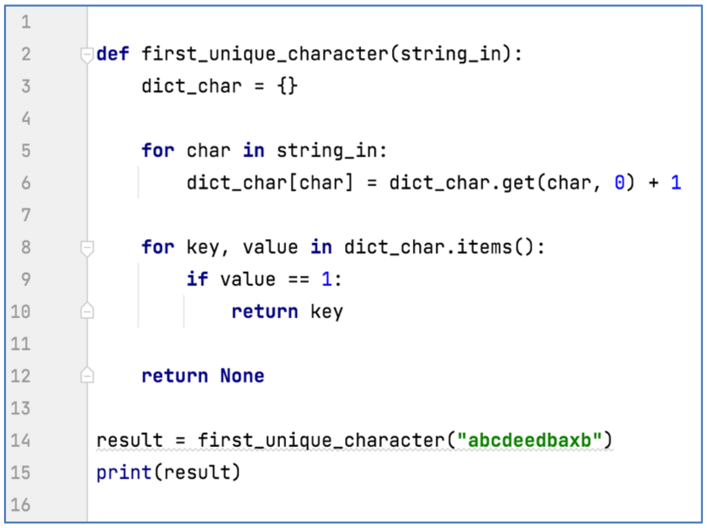
Whether you are self-taught, taught through an educational institution, or learnt by watching or reading the work of others you have acquired a skill. That skill grants you the ability to make a computer do a specific set of tasks. Line by line you create an application, service, automated test and so on, drawing on that knowledge and your previous experiences to write that application. It takes knowledge of writing in your preferred programming language, knowledge of the ecosystem you're building for, and knowledge of the type of program you're building. You've gained that skill through a mix of either study or observation and through experience. You have acquired the art of writing code, however there is more nuance to this philosophy, so we need to now look at the next important definition of art.

"*Art (noun) - The conscious use of skill and creative imagination especially in the production of aesthetic objects*" (Merriam-Webster)

The first definition allows us to the define our work as art because it is an acquired skill while this second definition expands that concept by taking the concept of that skill and using it with creative imagination to produce aesthetic objects. Let's break down the process of developing. Whether you are assigned a task or just wanting to build a new application on your own, you are starting with a blank canvas. Since the solution doesn't exist, you begin to really think about how to build it, drawing on your skill and experience, engaging your imagination on how to build it. Of course, you're using the principles of mathematics and logic to think through how the code of the application will work, however, in addition your imagination is powering the dream to bring that application to life and to bring it into the real world.

2.2 Meter, Rhythm, Poetry - The Aesthetic Object of Our Code

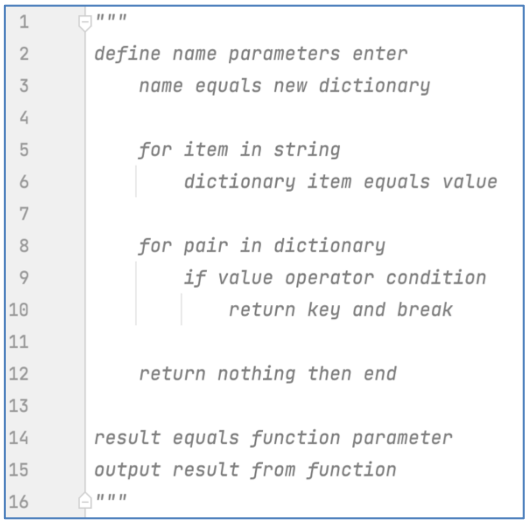
The aesthetic part of the application comes in several forms. There is the obvious piece, if the application has some sort of UI or output you could consider that aesthetic, but we can go deeper. Truly look at the code you produce. Read it thoroughly, but more importantly read past the words in the code sheet. When you look at the code, not for the words or their meaning, but something more you begin to see it. The meter, rhythm, rhyme. Stanzas, verse, prose. The use of whitespace to separate parts of the code to make it readable? No, beautiful. This code snippet will help illustrate this concept.



```python

def first_unique_character(string_in):
    dict_char = {}

    for char in string_in:
        dict_char[char] = dict_char.get(char, 0) + 1

    for key, value in dict_char.items():
        if value == 1:
            return key

    return None

result = first_unique_character("abcdeedbaxb")
print(result)

```

(Figure 1: A Python code snippet for returning the first unique character of a given string)

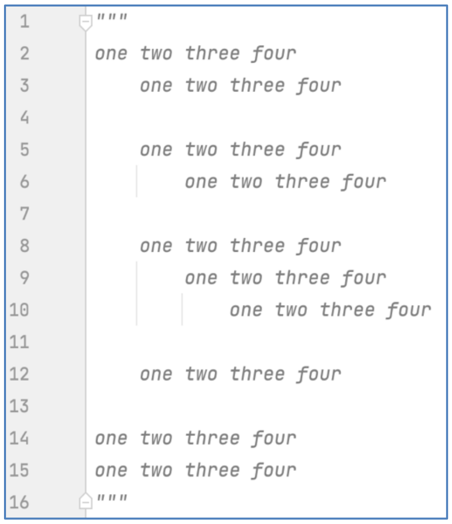
Read the code above. It should look obvious as a function that finds and returns the first unique value of a string in Python. Perhaps it's not the most efficient method of the desired result but that's not important for this concept. Read the snippet again. What do you see, the same function? Keep reading it until the names, keywords and values lose meaning. You might begin to see it, the meter inside the code. This might help:



```
"""
define name parameters enter
    name equals new dictionary

    for item in string
        dictionary item equals value

    for pair in dictionary
        if value operator condition
            return key and break

    return nothing then end

result equals function parameter
output result from function
"""
```

(Figure 2: The Python code snippet represented in pseudo code)

This pseudo code represents the same function, however its more than that. It shows the meter of the code as it is read. "Define name parameter enter", "name equals new dictionary", and so on. The meter is four beats per line. Let's break down the snippet one more time to see the true nature of it.



```
"""
one two three four
    one two three four

    one two three four
        one two three four

    one two three four
        one two three four
            one two three four

    one two three four

one two three four
one two three four
"""
```

(Figure 3: The meter of the code)

And there it is. When you realize it uses mathematics as its meter and logic as its prose, you see the aesthetic object of the code. Poetry. Simple and beautiful and yet profoundly complex. A piece of creation that came from the heart and mind, a marriage of talent and skill. As a developer, you have been writing poetry to the computer and probably never realized it.

2.3 Art in Almost Everything We Do

In his book *Understanding Comics: The Invisible Art*, American cartoonist and comic theorist, Scott McCloud asserts that "in almost everything we do there is at least an element of art" [2] because he defines art as "any human activity which doesn't grow out of either of our species' two basic instincts: survival and reproduction." [2]. This philosophy is a driving force to understanding the art within our craft as software engineers. The fact that we need to follow a creative process to create our code is proof that we, by nature, are truly artists under the guise of Computer Science. Related to the creative process, McCloud breaks down the creative, artistic process into a six-step process, one that we follow as developers without even being conscious of it.

1. "Idea/Purpose: The impulses, the ideas, the emotions, the philosophies, the purposes of the work. The work's context." [2]
2. "Form: The form it will take… Will it be a book? A chalk drawing? A chair? A song? A sculpture? A comic book?" [2]
3. "Idiom: The 'school' of art, the vocabulary of styles or gestures or subject matter, the genre that the work belongs to… Maybe a genre of its own." [2]
4. "Structure: Putting it all together… What to include, what to leave out... How to arrange, how to compose the work." [2]
5. "Craft: Constructing the work, applying skills, principal knowledge, invention, problems solving, getting the 'job' done." [2]
6. "Surface: Production values, aspects most apparent on the first superficial exposure to the work." [2]

2.4 The Creative Process as It Applies to Development

Consider, for a moment, how you write your code and design your projects. You likely have a process of build that you adhere to, be it consciously or unconsciously, that ensures you get the job done. There's a possibility you approach your work in a similar methodology outlined in these six steps. It may look something like this:

1. Idea/Purpose: Upon being assigned a task you begin to react to impulses and ideas on how to do the work. The task you receive or app you imagine may defines the context of your work and then you apply your knowledge to generate ideas for the solution.
2. Form: You decide the form the task or app will take. Maybe it's a command line app, maybe a Lambda in AWS, maybe a service or a script, or perhaps a mobile app.
3. Idiom: In this case, idiom defines a lot about the process of writing the code. What programming language to use, the coding paradigms you align to, how are you going to unit test it?
4. Structure: Putting the coding project together, creating the directory structure, defining the namespaces, perhaps adding your tests if you subscribe to Test Driven Development (TDD).
5. Craft: Writing the code. Applying your knowledge and skills to solve the problem and bring the code to life.
6. Surface: Building and running the code. Handing off to your test team for validation, building test automation against it. Delivering the solution to your customers or stakeholders.

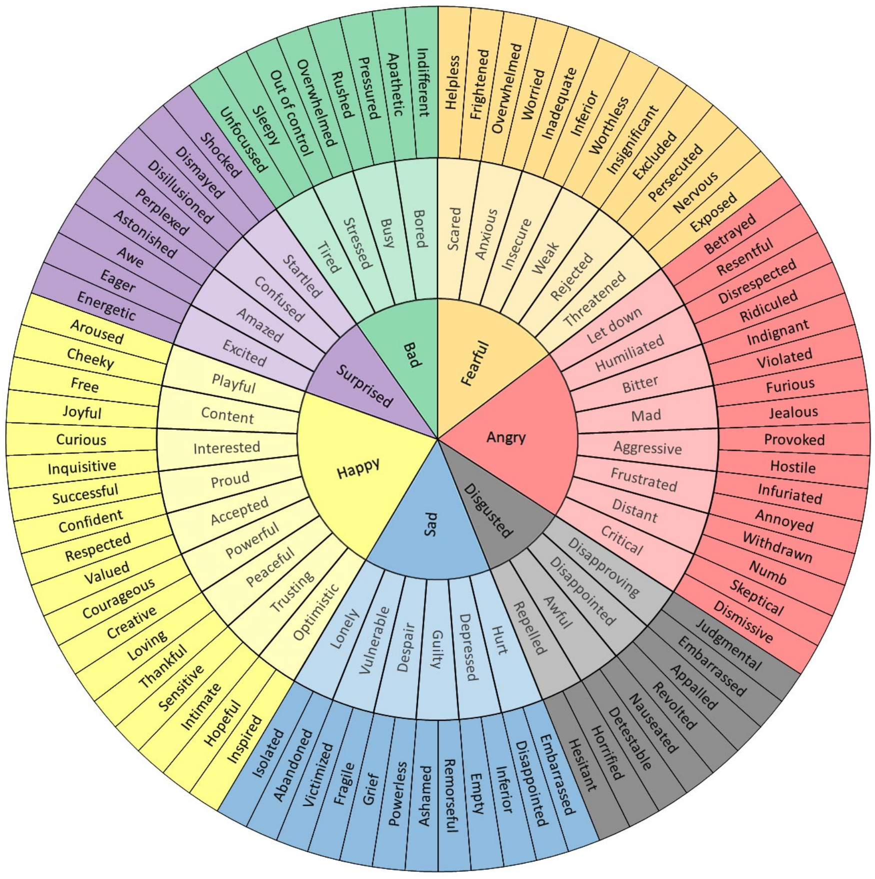
Perhaps you don't engage in all six steps, perhaps you've been engaging them in every task, and you haven't even noticed, perhaps you have additional steps. Regardless of your process, you are engaging in an act of creation with each line you write and each task you complete. Creativity, passion, skill, art flowing from finger to key to code file. These principles apply whether you're a software engineer, software development engineer in test, software test engineer, or any other member of a development team. There are aspects of art in nearly everything we do, writing code, software development is no exception.

3 Emotions and Their Effect on the Creative Process

Now that we've established the philosophy that we, as engineers, are artists, let us consider how our emotional state affects the quality of the work we do. The concept might seem obvious, "feel bad, do bad work"; however, how our emotions affect how we code is more nuanced than that. Our emotions can affect more than just the way our code functions- it can affect how it looks, how it reads, the amount of white space we have, our commenting, the way we name things, the way we test our code and more.

3.1 Defining Our Emotions and Feelings

Think about your base line for writing code. You're in the groove, you feel confident, perhaps a little happy, you can think clearly, the task is simple to achieve without being complex. You likely have a very distinct style to your code that shines through on the code sheet in this state. Now consider a time you were writing code for a project, and you were frustrated, or sad, anxious, or upset; likewise consider a time when you were extremely excited, elated, happy, or maybe even creative. Have you ever noticed a difference between the times you're operating in at your baseline and times when you are operating under a given emotion? You may have noticed that your emotional state affected the way you wrote your code. To see this in action, let's first define the veritable continuum of emotions a human can express.



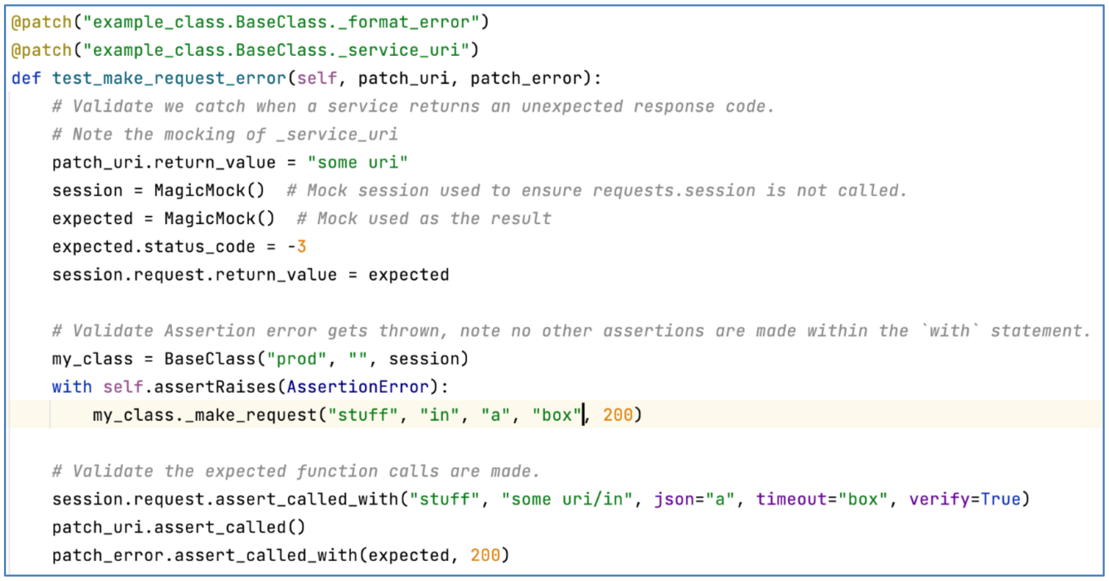
(Figure 4: *The Emotion Wheel* by Geoffrey Roberts, based on *The Feelings Wheel* by Dr. Gloria Wilcox)

In October of 1982, Dr. Gloria Wilcox published the original Feelings Wheel in volume 12, issue 4 of the Transactional Analysis Journal. In 2015, Imgur user Geoffrey Roberts derived the Emotions Wheel from her work [3]. This version expanded the feelings listed on the original wheel and represents a wide, albeit not comprehensive, range of human emotions and feelings. The intended use of this tool is to help people identify what they are feeling to better understand their current emotional state and raise their self-awareness. We will use this tool to identify feelings associated with specific code samples to see how they differ from a baseline.

3.2 How Emotions Can Affect the Code You Build

Important to Note: *Some of the code snippets below have been modified or recreated by the original artists to ensure the removal any proprietary or confidential information. All snippets come from the same* *writer who requested that their identity not be revealed for privacy purposes, for the purposes of this paper they will simply be known as The Poet.*

3.2.1 A Sterile Baseline – Code Written for Educative Purposes

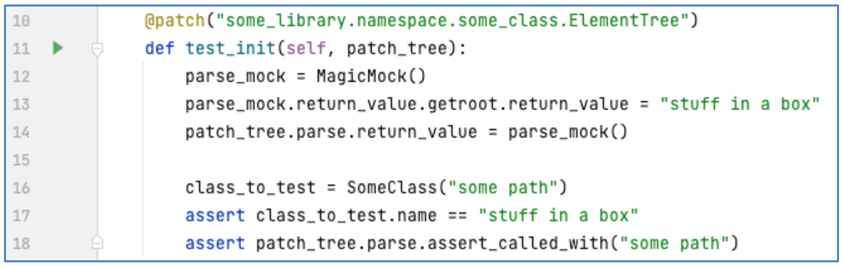


```python
@patch("example_class.BaseClass._format_error")
@patch("example_class.BaseClass._service_uri")
def test_make_request_error(self, patch_uri, patch_error):
    # Validate we catch when a service returns an unexpected response code.
    # Note the mocking of _service_uri
    patch_uri.return_value = "some uri"
    session = MagicMock()  # Mock session used to ensure requests.session is not called.
    expected = MagicMock()  # Mock used as the result
    expected.status_code = -3
    session.request.return_value = expected

    # Validate Assertion error gets thrown, note no other assertions are made within the `with` statement.
    my_class = BaseClass("prod", "", session)
    with self.assertRaises(AssertionError):
        my_class._make_request("stuff", "in", "a", "box", 200)

    # Validate the expected function calls are made.
    session.request.assert_called_with("stuff", "some uri/in", json="a", timeout="box", verify=True)
    patch_uri.assert_called()
    patch_error.assert_called_with(expected, 200)
```

(Figure 5: The "Base Line" Code Snippet written in Python)

The Poet asserted that this is the type of code that they are known to write on a day-to-day basis, especially since it was used as an example for writing unit tests for their development team. As we can see in this code snippet, the code is tidy, neatly organized, and well named. It has no inherent flaws, no major issues. The code might not be efficient; however, it gets the task done of unit testing a function within the code of the application. We'll use this as a base line as we look at examples from the same developer.

3.2.2 Code Influenced by Feeling Unfocused

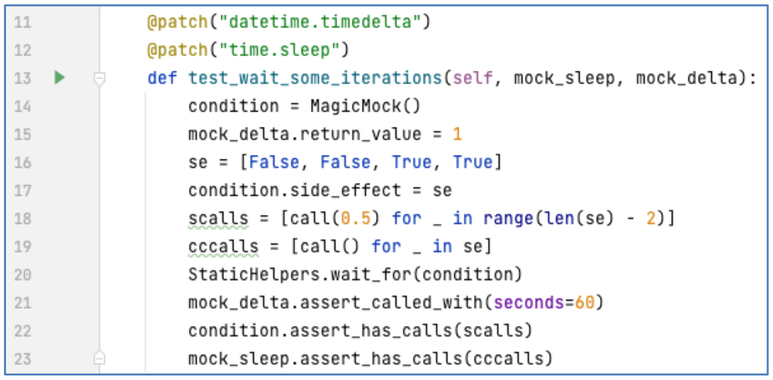


```python
10  @patch("some_library.namespace.some_class.ElementTree")
11  def test_init(self, patch_tree):
12      parse_mock = MagicMock()
13      parse_mock.return_value.getroot.return_value = "stuff in a box"
14      patch_tree.parse.return_value = parse_mock()
15
16      class_to_test = SomeClass("some path")
17      assert class_to_test.name == "stuff in a box"
18      assert patch_tree.parse.assert_called_with("some path")
```

(Figure 6: The "Unfocused" Feeling Code Snippet written in Python)

The Poet explained that they wrote this when they were feeling "unfocused". They mentioned they were able to resolve the issue but only after a half hour of debugging and triaging. If you were reading this code in either a pull request or a peer programming session and they said they were unable to get the test to pass, what would you see? The immediate issue is the mistake on line 18 as part of the assertion on the "assert called with" call. In this case, its attempting to assert that the return value of the function is not none, however that function has no return value and thus asserts as false and causes the test to fail. Additionally, a minor mistake on line 14 exists where The Poet initializes the mocked document parser even though it is already initialized on line 12 as a magic mock. This isn't that critical of a problem however it may result in some odd behavior when investigating assignments and calls within the mock itself later if the test was more complex. If we're unfocused, we tend to forget these little nuances. We tend to overlook minor facts of our jobs and it takes longer for us to figure out the issues.

3.2.3 Code Influenced by Feeling Infuriated

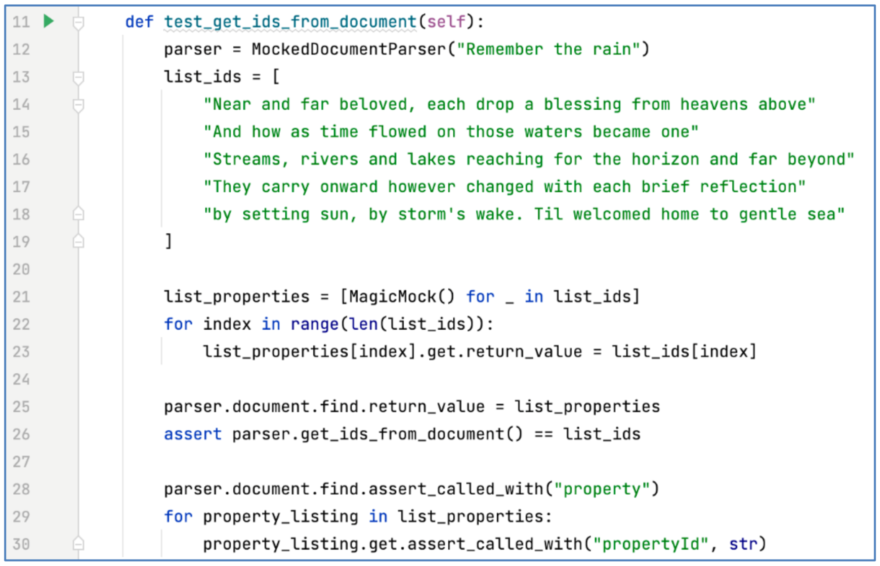


```python
@patch("datetime.timedelta")
@patch("time.sleep")
def test_wait_some_iterations(self, mock_sleep, mock_delta):
    condition = MagicMock()
    mock_delta.return_value = 1
    se = [False, False, True, True]
    condition.side_effect = se
    scalls = [call(0.5) for _ in range(len(se) - 2)]
    cccalls = [call() for _ in se]
    StaticHelpers.wait_for(condition)
    mock_delta.assert_called_with(seconds=60)
    condition.assert_has_calls(scalls)
    mock_sleep.assert_has_calls(cccalls)
```

(Figure 7: The "Infuriated" Feeling Code Snippet written in Python)

In this snippet, The Poet explained that they were struggling to mock the functionality of the Time Delta function in Python. In Python, Time Delta is treated as immutable which means the function itself cannot be mocked. The Poet wanted an easy way to control the function so they can test and make sure the proper deltas were being applied. With each try attempt that failed, they got more and more angry at the situation. They got to the point where they felt "infuriated" and were making simple mistakes, naming things "whatever" to complete the task. As we can see, many variable names have poor naming, it's hard to understand what the different variable names represent. Additionally, there is no white space separating out logical areas in the code. Finally, the code is so incoherent that even after revisiting it in this state several days later, The Poet themselves couldn't really make sense of it. According to them, it took them quite a while to finally give up and just make the code easier to mock by extracting the time calculation into its own function. They claimed that when they finally let go of the track they were following and refactored the code they felt disappointed that they couldn't make their original design work.

3.2.4 Code Influenced by Feeling Inspired



```python
def test_get_ids_from_document(self):
    parser = MockedDocumentParser("Remember the rain")
    list_ids = [
        "Near and far beloved, each drop a blessing from heavens above"
        "And how as time flowed on those waters became one"
        "Streams, rivers and lakes reaching for the horizon and far beyond"
        "They carry onward however changed with each brief reflection"
        "by setting sun, by storm's wake. Til welcomed home to gentle sea"
    ]

    list_properties = [MagicMock() for _ in list_ids]
    for index in range(len(list_ids)):
        list_properties[index].get.return_value = list_ids[index]

    parser.document.find.return_value = list_properties
    assert parser.get_ids_from_document() == list_ids

    parser.document.find.assert_called_with("property")
    for property_listing in list_properties:
        property_listing.get.assert_called_with("propertyId", str)
```

(Figure 8: The "Inspired" Feeling Code Snippet written in Python)

Lastly, we'll look at a time when The Poet was coding while feeling inspired. They mentioned that during this time they had solved a problem in a novel way and that it made them feel inspired and hopeful. As they unit tested the code, they added some lyrics to the song they were listening to as part of the return value of a given mock. They mentioned that it was probably overkill for what they were doing but it didn't take any additional time to build the test. Ultimately, they were happy to build the test in this manner and they ended up finishing this specific suite in this manner. Their colleagues even commented on the tests noting that they found it a fun way to write tests that also accomplished the task.

3.2.5 Summarizing The Impact of Emotions on Our Code

Emotion	***Effects***	***Recommendations***
Unfocused	• Syntactical Errors • Lapses in Critical Thinking • Decline in Debugging Skills • Lack of enthusiasm	• Resolve whatever is taking your focus. • Engage in simple mental exercises to free up your thought process. • Self-Care/Handle basic needs (Eat, Drink, use the bathroom, etc.)
Infuriated	• Lapses in critical thinking • Difficult to understand or inappropriate variable names. • Creation of difficult to read code. • Compounding frustration when trying to "force it to work"	• Step away - let yourself calm down. • Engage in grounding exercises to restore calm, rational thinking. • Learn to let go of solutions that won't work or that need refactoring. • Reach out to friends or colleagues to vent your frustration.
Inspired	• Playfulness in code or comments • Fluidity in solutions • Easier to read and parse code	• Continue as you are. Shine on.

(Figure 9: A table summarizing the emotions The Poet experienced, their effects on the code, and recommendations for resolving them)

The table above outlines the feelings The Poet encountered while developing and the effects that they had on their code. As we can see, our emotions can influence the way we build our code. It truly can mean the difference between a clean, well-written function and a one that has bugs, poor naming conventions, and inconsistencies. This doesn't mean we shouldn't do our jobs under the influence of given emotions, but it can inform us and our colleagues of how we're feeling in that moment. This information can help you not only better understand yourself but also understand your colleagues and inform how your colleagues understand you as well.

4 Artistic Appreciation, Emotional Intelligence, and the Quality Mindset

We've now established that emotions affect the work we do, the art we create code we write. Not only can they affect the whitespace and naming conventions, or rather the aesthetic object of the code, directly but emotions can affect how we approach engaging in the creative process and building our code. When you look back at code that you built under the influence of your emotions, it has the power to invoke those emotions again. It can also invoke additional feelings- embarrassment, pride, joy, insecurity, and so on because we are creative, and we place meaning in the work we do. The same can be said about when we look at code our colleagues have built under their emotions.

When the code you read invokes feelings, it connects you better to the person who wrote the code. Additionally, you can begin to pick out patterns in their code, ways that communicate their feelings in the context of their work. You can begin to identify when the person was feeling sad, frustrated, or any other host of emotions simply by reading the code and connecting with the human behind it. This is an incredible opportunity to help build a relationship with that individual and reach out to check in on them. The more you connect with the people on your team, the more it'll foster integrity and respect within the team and partners that you work with.

As a quality engineer, we excel at finding flaws in code and calling out the need to refactor code; although this is critical to our job, we can often alienate those we work with. By taking that opportunity to truly understand the people on our team we can help them truly shine and be their best selves. Not only does it help us build relationships with our partners, but it can also bring new opportunities for mentorship, knowledge sharing, and general improvement in the quality of the work the team does. We have opportunities every day to engage our artistic appreciation and emotional intelligence to help foster respect and comradery in our teams. By doing so, we can motivate our partners teams to engage in good quality practices and delight our customers in the process.

5 Conclusion

Code. What do you think of when you read that word now? Perhaps now there is a new synonym in your mind when you read it: Art. Whether as a quality engineer or a software engineer, our artistic ability grants us an ability no other artist has. The ability to bring life to the inanimate. The ability to conjure something from nothing as we paint across the canvas of a code sheet. Our art sets alight the fire of creation within lifeless circuits and pixels. An art that is so profound that not only does it influence the way that human and machine interact together bit it inspires the very fabric of our universe to come alive and dance in harmony. Our poetry inspires electrons to flow from atom to atom as bits and bytes flowing along traces within the computer and help it understand, calculate, store and recall data. A beauty that transcends its mathematical and scientific roots, piercing right into the heart of the humanities. As computer programmers that is the power we wield.

As we are artists, it is our duty to see the beauty and humanity in our partners' work. We should take an opportunity to realize that even though we might think we have "the right answers" when we're developing our code or giving feedback on other's code, that others developing code are artists as well. The way they write their code may be influenced by the emotions they feel at the time, and we should be conscious of those emotions and how they define them as a person. By changing the perspective on how we think about code, realizing the artistic nature of code, we can reframe how we give feedback and nurture the relationships between us and our peers and partners. In nurturing those relationships and approaching our feedback from the lens of artistic appreciation, we can begin to foster a new level of integrity and respect in those we work with which will ultimately help improve the acceptance of our qualitative feedback.

References

1. Merriam-Webster.com Dictionary. n.d. *art*. Accessed June 1st, 2023. https://www.merriam-webster.com/dictionary/art
2. McCloud, Scott. 1993. *Understanding Comics: The Invisible Art*. New York: Kitchen Sink Press.
3. Roberts, Geoffrey. 2015. *Emotions Wheel*, Imgur, entry posted March 15, 2015, https://imgur.com/a/CkxQC (accessed June 1st, 2023).