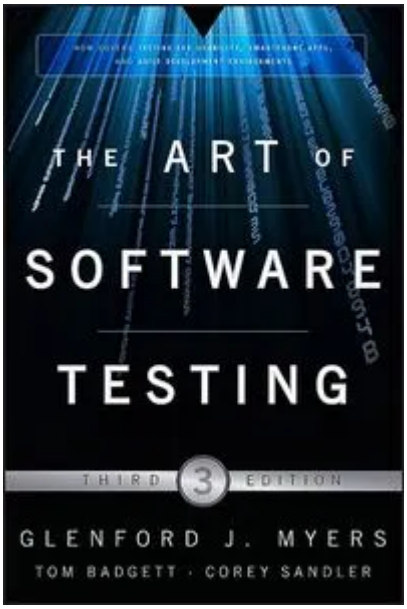
The Art of Software Testing, from Glenford Myers

11 min read · Aug 28, 2023

José Sobral

Listen Share



The rise of AIs has revolutionized numerous industries, among them, our software industry couldn't be an exception: a survey from Github shows that over 90% of developers have already used some type of AI and 40% of them use AIs on a daily routine.

However, despite seeming and indeed being a significant transformation, we need to be aware of some current model limitations, like the **lack of understanding of question full context**, the **biases on responses** and the **model knowledge limitations.**

When it comes to software testing, these limitations of current models turns out to be even more significant:

- **Lack of Context**: When designing tests, it is crucial to have a deep understanding of the application's purpose, its target audience, the most common usage scenarios, and the potential risk areas. For example, a banking app and a casual game will have vastly different risk profiles and user behaviors. A human tester brings to the table a wealth of contextual knowledge, such as which inputs might be more commonly used, which flow controls are more at risk, or which areas of an app are more crucial to its function.

- **Biases on Responses**: The AI model is only as good as the data it was trained on. If the model was trained using data from sources that consistently follow poor practices or outdated methodologies, the AI would reflect those biases. For instance, if all sources suggest an outdated testing approach, the model might recommend it even if it's not the best choice for a given scenario.

- **Knowledge Limitation**: AI models, like ChatGPT, have a "knowledge cutoff", meaning they are not aware of developments, techniques, tools, or best practices introduced after this date. Software testing is a field that evolves rapidly, with new methodologies, tools, and best practices emerging frequently. Human testers can continuously learn and adapt, absorbing the latest knowledge from conferences, workshops, forums, and community interactions. Right now, the model can not

`collections` and associated `products` based on `store` entity:

```
RSpec.describe Stores::CollectionsController, type: :controller do
  let(:store) { create(:store) }
  let(:collection) { create(:collection, store: store) }
  let(:product) { create(:product, collection: collection) }

  describe "GET #show" do
    context "when collection exists" do
      before { get :show, params: { store_id: store.id, collection_id: collecti

      it "assigns the collection" do
        expect(assigns(:collection)).to eq(collection)
      end

      it "assigns the products" do
```

```
          expect(assigns(:products)).to include(product)
        end

        it "renders the show template" do
          expect(response).to render_template(:show)
        end
      end

      context "when collection does not exist" do
        before { get :show, params: { store_id: store.id, collection_id: -1 } }

        it "redirects to store path" do
          expect(response).to redirect_to(store_path(store.slug))
        end
      end
    end
  end
```

There are several mistakes in this code provided by AI, lets take a look on two of them:

1. **`render_template` is deprecated**: `render_template` has been deprecated in more recent versions of RSpec, since 2015.
2. **`assigns` is an implementation test**: `assigns` was traditionally used to access the instance variables set in the controller. The problem with this is that you're testing the internal implementation of the controller rather than its external output or behavior. Unit tests should focus on the expected behavior and not on how this behavior is internally achieved.

Given this kind of limitations on tests generated by AI's, it's necessary to understand the fundamentals beyond the test writing.

This article aims to provide a summary of concepts presented in the book "The Art of Software Testing" by Glenford Myers, in an attempt to shed light on and introduce the science of testing.

The Psychology of Testing

In the software world, it's common for us to learn by solving problems. This approach is usually very effective, and there's a plethora of articles that correlate problem-solving with software.

However, a thesis raised by the author in Chapter 2 — "The Psychology and Economics of Software Testing" — is that there's a philosophy behind the thinking of constructing tests. Often, this philosophy gets misconstrued, and problem-solving is grounded on incorrect premises. The author states:

> **The purpose of testing is to show that a program performs its intended functions correctly.**
>
> **Testing is the process of establishing confidence that a program does what it's supposed to do.**

This portrays a testing psychology focused on ensuring that your system works correctly.

However, according to the author, the real psychology behind testing isn't to ensure your system works correctly, but rather to break it. In the author's words:

> **Testing is the process of executing a program with the intent of finding errors.**

This entirely shifts our problem-solving perspective; when our goal is to demonstrate that a program is error-free, we subconsciously select data that has a low likelihood of causing the program to fail.

Another way to spot this testing psychology is the use of the terms "*successful*" and "*unsuccessful*", especially when used by product managers. In most projects, PMs categorize a "successful" test when the test finds no errors. We need to start with the assumption that all programs contain errors and the purpose of testing is to locate them.

In summary, it's as if the process of software testing is seen as a process of trying to constantly break your code. In the next section, we'll explore methods to help structure our thinking when attempting to "break our code", that is, to test it.

Black Box Testing

The author introduces two techniques to assist developers in the testing process and on “break things”. In the first one, black box testing, we operate under the assumption that our code is a “black box” and we don’t test control flows and business rules; we only test how our “black box” handles inputs and what its outputs are.

> *also known as data driven or input/output-driven testing*

Thus, the goal of black box testing is to attempt to break our objects and methods based on different inputs. Some strategies to apply black box testing include:

Equivalence partitioning

This technique divides the software’s input space into partitions of equivalent values. The idea here is that if one input in a given partition is valid, then all inputs from that partition are. So, instead of testing each input individually, testers can simply test one representative input from each partition.

in practice

Suppose you have a registration form in your application that accepts an age range of users between 18 and 65 years. The equivalent partitions here would be:

- Under 18 years old
- Between 18 and 65 years old
- Over 65 years old

To test, you can choose a representative age from each partition (for instance, 15, 30, and 70) and check if the system correctly processes each entry.

```
from django.test import TestCase
from .models import User

class EquivalencePartitioningTest(TestCase):
    def test_underage(self):
        response = self.client.post('/signup/', {'age': 15})
        self.assertContains(response, 'Age is too low')

    def test_valid_age(self):
        response = self.client.post('/signup/', {'age': 30})
        self.assertContains(response, 'Congratulations!')
```

```
    def test_over_age(self):
        response = self.client.post('/signup/', {'age': 70})
        self.assertContains(response, 'Age is too high.')
```

Boundary value analysis

In this technique, the focus is on the boundary values or edge points of the input ranges. Errors commonly occur at these extremities, which is why it's crucial to test values just above, below, and right at the acceptance limits.

in practice

Using the same registration form example mentioned above, the boundary values would be 17, 18, 65, and 66. The goal would be to test these specific values because it's at these boundaries or extremities that errors are most likely to occur. For instance, a user who is 18 years old should be accepted, while a 17-year-old user should not be.

```
class BoundaryValueAnalysisTest(TestCase):
    def test_age_just_below_boundary(self):
        response = self.client.post('/signup/', {'age': 17})
        self.assertContains(response, 'Age is too low.')

    def test_age_at_lower_boundary(self):
        response = self.client.post('/signup/', {'age': 18})
        self.assertContains(response, 'Congratulations')

    def test_age_at_upper_boundary(self):
        response = self.client.post('/signup/', {'age': 65})
        self.assertContains(response, 'Congratulations!')

    def test_age_just_above_boundary(self):
        response = self.client.post('/signup/', {'age': 66})
        self.assertContains(response, 'Age is too high.')
```

Cause-effect graphing

This technique involves creating a chart that maps out the relationships between different causes (inputs) and their effects (outcomes). The graph is used to identify and devise test cases that cover all possible combinations of inputs and their resulting effects.

in practice

Let's say you have a form that accepts entries for an event. Causes could include: type of ticket selected, seat availability, event date. The effects could be: purchase confirmation, unavailable seat error, invalid date error, etc. You would create a chart to map all these inputs and their possible outcomes, ensuring that all scenarios are tested. In summary:

Let's consider a simple event booking system.

Causes (Inputs):

- Ticket type: Regular, VIP, Student Discount
- Seat availability: Available, Unavailable
- Event date: Valid Date, Past Date

Effects (Outcomes):

- Purchase Confirmation
- Unavailable Seat Error
- Invalid Date Error
- Discount Verification (for student discount)

The graph can be described as:

- If *Regular* or *VIP* ticket is selected with *Available* seat and *Valid Date*, then the outcome is *Purchase Confirmation*.
- If *Student Discount* ticket is selected with *Available* seat and *Valid Date*, then the outcomes are both *Purchase Confirmation* and *Discount Verification*.
- If any ticket type is chosen with *Unavailable* seat, regardless of the event date, then the outcome is *Unavailable Seat Error*.
- If any ticket type is selected with either *Available* or *Unavailable* seat but with *Past Date*, then the outcome is *Invalid Date Error*.

With this cause-effect perception, you can build several tests to try to "break your code".

Error guessing

As the name suggests, this technique is less systematic and more based on the tester's intuition and experience. Testers try to "guess" where errors might be present and craft test cases based on those assumptions. For instance, if a tester knows that a particular kind of input caused issues in previous software, they might decide to test that specific input again.

in practice

```
class ErrorGuessingTest(TestCase):
    def test_special_characters_in_name(self):
        response = self.client.post('/signup/', {'name': '@John!'})
        self.assertContains(response, 'Invalid chars')
```

White box Testing

> *also known as logic-driven*

The other technique, white box testing, aims to test the flow control of our code. This approach is commonly used to know the test coverage of our application. However, as the author himself illustrates, in many cases, it's impossible to test all the flows of a particular system. Consider the example below.

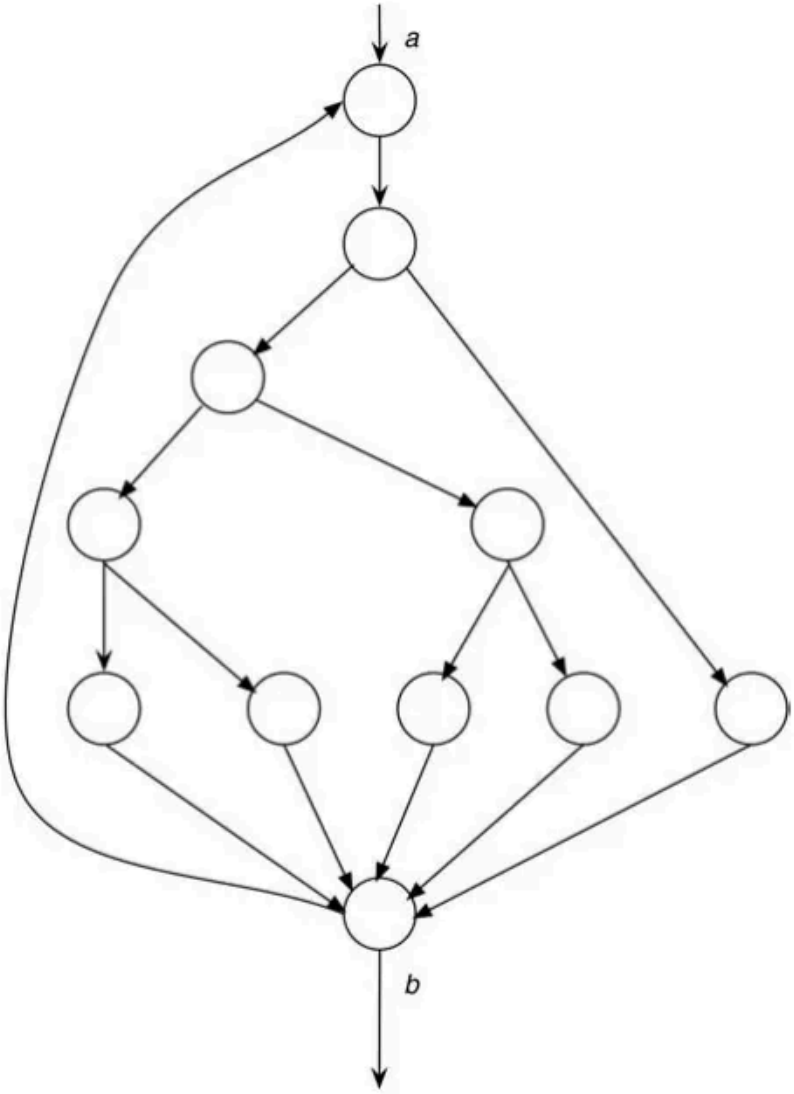


FIGURE 2.1 Control-Flow Graph of a Small Program.

In this image, there are 10^{13} possible flows between **a** and **b**. It would be impractical to create this number of test cases for our system.

That's one of the reasons why the models limitation mentioned above of the **Lack of Context** is crucial: only humans can have knowledge of full context of an application and decide which flow control is more critical to test.

In this regard, some techniques are employed in the use of white box testing to guide our process to "break our code" (test):

Statement Coverage

Aims to ensure that each statement or line of code is executed at least once; essential for detecting parts of the code that are not executed under any test scenario — this is what `coverage` files typically show.

in practice

Let's suppose we have the following authentication function.

```
def authenticate(username, password):
    user = CustomUser.objects.filter(username=username)

    if not user:
        return 'User not found'

    if not user.is_active:
        return 'User is not active'

    if user.failed_attempts >= 3:
        return 'Account locked due to too many failed attempts'

    if user.password != password:
        user.failed_attempts += 1
        user.save()
        return 'Invalid password'

    user.failed_attempts = 0
    user.save()
    return 'Authentication successful'
```

This technique ensures that each statement in the code is executed at least once during testing. In the authenticate function, we use statement coverage to ensure that each of the if statements is executed at least once during testing.

```
def test_authenticate_statement_coverage(self):
    # Test case where user is not found
    result = authenticate('nonexistent_user', 'password')
    self.assertEqual(result, 'User not found')

    # Test case where user is not active
    inactive_user = CustomUser.objects.create(username='inactive_user', is_acti
    result = authenticate('inactive_user', 'password')
    self.assertEqual(result, 'User is not active')

    # Test case where account is locked due to too many failed attempts
    locked_user = CustomUser.objects.create(username='locked_user', failed_atte
    result = authenticate('locked_user', 'password')
    self.assertEqual(result, 'Account locked due to too many failed attempts')
```

```
    # Test case where password is invalid
    user = CustomUser.objects.create(username='user', password='password')
    result = authenticate('user', 'wrong_password')
    self.assertEqual(result, 'Invalid password')

    # Test case where authentication is successful
    result = authenticate('user', 'password')
    self.assertEqual(result, 'Authentication successful')
```

Decision Coverage

Ensures that each decision or branch (like an `if-else` statement) is tested for both options: `True` or `False` .

in practice

This technique ensures that each decision point in the code is executed both when the condition is `True` and when it is `False` . In the `authenticate` function, we use decision coverage to ensure that the `if user.password != password` decision point is executed both when the password is correct and when it is incorrect.

```
def test_authenticate_decision_coverage(self):
    # Test case where password is invalid
    user = CustomUser.objects.create(username='user', password='password')
    result = authenticate('user', 'wrong_password')
    self.assertEqual(result, 'Invalid password')
    self.assertEqual(user.failed_attempts, 1)

    # Test case when password is valid
    result = authenticate('user', 'password')
    self.assertEqual(result, 'Authentication successful')
```

Multiple-Condition Coverage

Similar to condition coverage, but goes further. It ensures that all possible combinations of conditions in a decision are tested.

in practice

This technique ensures that each possible combination of conditions in a decision point is executed at least once. In the `authentication` function, we use multiple-

condition coverage to ensure that each possible combination of conditions in the decision point is executed at least once during testing.

```
def test_authenticate_multiple_condition_coverage(self):
    # Test case where user is not found
    result = authenticate('nonexistent_user', 'password')
    self.assertEqual(result, 'User not found')

    # Test case where user is not active
    inactive_user = CustomUser.objects.create(username='inactive_user', is_acti
    result = authenticate('inactive_user', 'password')
    self.assertEqual(result, 'User is not active')

    # Test case where account is locked due to too many failed attempts
    locked_user = CustomUser.objects.create(username='locked_user', failed_atte
    result = authenticate('locked_user', 'password')
    self.assertEqual(result, 'Account locked due to too many failed attempts')

    # Test case where password is invalid
    user = CustomUser.objects.create(username='user', password='password')
    result = authenticate('user', 'wrong_password')
    self.assertEqual(result, 'Invalid password')
    self.assertEqual(user.failed_attempts, 1)

    # Test case where authentication is successful
    result = authenticate('user', 'password')
    self.assertEqual(result, 'Authentication successful')
    self.assertEqual(user.failed_attempts, 0)

    # Test case where password is correct but user is not active
    inactive_user = CustomUser.objects.create(username='inactive_user2', is_act
    result = authenticate('inactive_user2', 'password')
    self.assertEqual(result, 'User is not active')

    # Test case where password is correct but account is locked
    locked_user = CustomUser.objects.create(username='locked_user2', failed_att
    result = authenticate('locked_user2', 'password')
    self.assertEqual(result, 'Account locked due to too many failed attempts')
```

Big Apps

To provide some real world examples of these techniques been applied into our industry, I've brought some open source projects code snippets:

React

This code block provides an example of how React codebase uses black box testing to check if given a `className` (input) that contains `\n` char, the actual `className` rendered (output) will still be available.

```
it('can scryRenderedDOMComponentsWithClass with className contains \\n', () =
  class Wrapper extends React.Component {
    render() {
      return (
        <div>
          Hello <span className={'x\ny'}>Jim</span>
        </div>
      );
    }
  }

  const renderedComponent = ReactTestUtils.renderIntoDocument(<Wrapper />);
  const scryResults = ReactTestUtils.scryRenderedDOMComponentsWithClass(
    renderedComponent,
    'x',
  );
  expect(scryResults.length).toBe(1);
});
```

Python

This code block provides an example of how Python codebase uses black box testing to check if a `Tuple` responds to correctly (output) to different arguments (input)

```
 def test_constructors(self):
      super().test_constructors()
      # calling built-in types without argument must return empty
      self.assertEqual(tuple(), ())
      t0_3 = (0, 1, 2, 3)
      t0_3_bis = tuple(t0_3)
      self.assertTrue(t0_3 is t0_3_bis)
      self.assertEqual(tuple([]), ())
      self.assertEqual(tuple([0, 1, 2, 3]), (0, 1, 2, 3))
      self.assertEqual(tuple(''), ())
      self.assertEqual(tuple('spam'), ('s', 'p', 'a', 'm'))
      self.assertEqual(tuple(x for x in range(10) if x % 2),
                       (1, 3, 5, 7, 9))
```

Gitlab

This code block provides an example of how Gitlab codebase uses white box testing to cover multiple status of a `custom_email_verification` service:

```
  context 'when status is :finished' do
    before do
      subject.mark_as_started!(user)
      subject.mark_as_finished!
    end

    it { is_expected.to validate_absence_of(:token) }
    it { is_expected.to validate_absence_of(:error) }
  end

  context 'when status is :failed' do
    before do
      subject.mark_as_started!(user)
      subject.mark_as_failed!(:smtp_host_issue)
    end

    it { is_expected.to validate_presence_of(:error) }
    it { is_expected.to validate_absence_of(:token) }
  end
end
```

Conclusion

The Glendford' book covers much more about testing and i highly recommend you to read the book. The topics I've covered are just some of them that really helps me to criticize tests generated by AI's and have a deeper knowledge of what software test is really about.

If there are specific topics from the book that have been particularly enlightening for you, I'd love to hear about them in the comments. Additionally, if you're aware of any strategies that can improve AI-generated tests, please share them with the community in the comments section.

Software Engineering | Software Testing | Software Books | Django

Written by José Sobral

38 followers · 46 following

Responses (2)

See all responses