#23 TECHARTIKEL

Automatisches Testen mit GitHub Copilot: Eine Revolution in der Software-Qualität

27.11.2025

Kai Weidner

Inhalt

Automatisches Testen mit GitHub Copilot: Eine Revolution in der Software-Qualität

In der modernen Softwareentwicklung ist das automatisierte Testen ein unverzichtbares Element, um die Qualität und Zuverlässigkeit von Code sicherzustellen. Entwickler sind zunehmend auf Tools angewiesen, die ihnen helfen, wiederholende Aufgaben zu automatisieren und die Effizienz zu steigern. GitHub Copilot, ein KI- gesteuertes Code-Autocompletion-Tool, hat sich als eine der vielversprechendsten Technologien erwiesen, um diesen Prozess zu unterstützen - insbesondere im Bereich des automatisierten Testens. In diesem Artikel werden wir untersuchen, wie GitHub Copilot zur Automatisierung von Tests beiträgt, die Herausforderungen, die dabei auftreten, sowie Best Practices, um das Potenzial dieser Technologie vollständig auszuschöpfen.

1. GitHub Copilot: Ein Überblick

GitHub Copilot ist ein von OpenAI entwickeltes KI-Tool, das in den Entwicklungsworkflow integriert wird und Code-Vervollständigungen in Echtzeit anbietet. Es nutzt maschinelles Lernen und das OpenAI Codex-Modell, um Code basierend auf Kommentaren oder Eingaben des Entwicklers zu generieren. Die Nutzung von Copilot ist besonders vorteilhaft für sich wiederholende oder einfache Programmieraufgaben, aber auch für die Erstellung von Unit-Tests, Integrationstests und anderen Testarten.

2. Automatisiertes Testen: Warum es wichtig ist

Automatisierte Tests sind ein grundlegender Bestandteil jeder modernen Softwareentwicklungspraxis. Sie ermöglichen es, Fehler frühzeitig zu erkennen, und gewährleisten, dass der Code über verschiedene Versionen hinweg stabil bleibt. Unit-Tests, Integrationstests und End-to-End-Tests sind übliche Testarten, die automatisch ausgeführt werden können, um die Funktionsfähigkeit eines Systems zu überprüfen.

Die Herausforderung besteht jedoch darin, dass das Schreiben von Tests oft zeitaufwendig und repetitiv ist. Entwickler müssen sicherstellen, dass jeder Teil des Codes abgedeckt ist, was besonders bei komplexen Anwendungen eine aufwendige Aufgabe sein kann. GitHub Copilot kann hier Abhilfe schaffen, indem es Code-Vorschläge für Tests generiert und so den Testprozess erheblich beschleunigt.

3. GitHub Copilot im automatisierten Testen

3.1. Generierung von Unit Tests

Eine der häufigsten Anwendungen von GitHub Copilot im automatisierten Testen ist die Generierung von Unit-Tests. Indem Entwickler einfach die zu testende Funktion oder Methode kommentieren, kann Copilot automatisch passende Unit-Tests erstellen. Zum Beispiel könnte ein Entwickler eine Funktion zur Berechnung der Fibonacci-Zahlen kommentieren und GitHub Copilot würde dazu passende Unit-Tests generieren, um die Korrektheit der Implementierung zu überprüfen. Das

spart nicht nur Zeit, sondern auch den Aufwand, selbst alle Tests manuell zu schreiben.

3.2. Test-Coverage und Edge Cases

Copilot hilft auch bei der Identifizierung von Edge Cases, die in vielen Fällen leicht übersehen werden. Durch seine Fähigkeit, Muster aus bestehenden Codebasen zu erkennen, kann GitHub Copilot dazu beitragen, Testfälle für seltene oder ungewöhnliche Eingaben zu generieren, die eine Anwendung möglicherweise abstürzen lassen würden, falls sie nicht ausreichend getestet werden. Diese Funktion ist besonders hilfreich, um eine hohe Testabdeckung sicherzustellen.

3.3. Integrationstests

Neben Unit-Tests kann GitHub Copilot auch beim Schreiben von Integrationstests unterstützen. Besonders in komplexen Systemen, bei denen verschiedene Module oder Microservices miteinander kommunizieren, sind Integrationstests unerlässlich, um sicherzustellen, dass alle Teile des Systems wie erwartet zusammenarbeiten. Copilot kann dazu beitragen, Tests für API-Endpunkte zu erstellen oder das Zusammenspiel von Modulen zu überprüfen, was eine zusätzliche Schicht der Qualitätssicherung hinzufügt.

3.4. End-to-End Tests

End-to-End-Tests (E2E-Tests) sind darauf ausgelegt, die Anwendung aus Sicht des Endbenutzers zu testen. Sie überprüfen, ob das gesamte System korrekt funktioniert, von der Benutzeroberfläche bis hin zu den Backend- Diensten. GitHub Copilot kann helfen, Testskripte für gängige Testautomatisierungs-Frameworks wie Selenium oder Cypress zu schreiben, wodurch der Aufwand für die Implementierung von E2E-Tests erheblich reduziert wird.

4. Praxisbeispiel: Automatisierte Tests für eine API mit GitHub Copilot

Stellen wir uns vor, ein Entwicklerteam arbeitet an einer RESTful API, die eine Vielzahl von Daten zu verschiedenen Themen liefert, darunter auch Benutzerinformationen und Bestelldaten. Die API stellt mehrere Endpunkte zur Verfügung, um Daten zu abrufen, zu erstellen, zu aktualisieren und zu löschen. Um sicherzustellen, dass diese API unter verschiedenen Bedingungen zuverlässig funktioniert, müssen umfangreiche Tests geschrieben werden.

4.1. Schritt 1: Das Szenario Definieren

Der Entwickler möchte sicherstellen, dass der Endpunkt zur Benutzerabfrage (GET /users/{id}) korrekt funktioniert und die richtigen Benutzerinformationen zurückgibt. Die API sollte bei einem gültigen Benutzer-ID die entsprechenden Daten liefern und bei einer ungültigen ID eine Fehlermeldung zurückgeben.

4.2. Schritt 2: Kommentieren des Codes

Anstatt alle Tests manuell zu schreiben, könnte der Entwickler GitHub Copilot nutzen. Zuerst schreibt er einen kurzen Kommentar, um dem Tool zu helfen, den gewünschten Testfall zu verstehen. Zum Beispiel:

```
1 // Test: Validiert, dass der API-Endpunkt '/users/{id}' bei einer ungültigen ID den
      Fehlercode 404 zurückgibt
```

4.3. Schritt 3: GitHub Copilot generiert den Test

Basierend auf diesem Kommentar schlägt GitHub Copilot automatisch eine Testfunktion vor, die dem Entwickler als Grundlage dient:

```
using System.Net.Http;
using System.Threading.Tasks;
using NUnit.Framework;

namespace ApiTests
{
  [TestFixture]
  public class UserApiTests
  {
    private static readonly HttpClient client = new HttpClient();

    [Test]
    public async Task TestGetUserById_ValidId_ReturnsCorrectUser()
    {
      // Definiere die gültige Benutzer-ID
      var userId = 1;

      // Sende eine GET-Anfrage an den API-Endpunkt
      var response = await client.GetAsync($"https://example.com/api/users/{userId}");

      // Überprüfe, ob der Statuscode 200 (OK) zurückgegeben wird
      Assert.AreEqual(200, (int)response.StatusCode, $"Fehler: Statuscode {response.
    StatusCode}");

      // Überprüfe, ob die Rückgabedaten die richtigen Benutzerinformationen enthalten
      var userData = await response.Content.ReadAsAsync<User>();
      Assert.AreEqual(userId, userData.Id, "Fehler: Benutzer-ID stimmt nicht überein");
      Assert.IsNotNull(userData.Name, "Fehler: Benutzername fehlt");
      Assert.IsNotNull(userData.Email, "Fehler: Benutzeremail fehlt");
    }
  }

  public class User
  {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
  }
}
```

4.4. Schritt 4: Erweiterung des Tests für Edge Cases

GitHub Copilot kann auch Tests für Randfälle generieren. Der Entwickler möchte sicherstellen, dass der API-Endpunkt für eine ungültige Benutzer-ID den Fehlercode 404 zurückgibt. Der Kommentar könnte folgendermaßen aussehen:

```
// Test: Validiere, dass der API-Endpunkt '/users/\{id\}' für eine gültige ID die richtige
    Benutzerinformation zurückgibt.
```

Daraufhin schlägt GitHub Copilot folgenden Test vor:

```
[Test]
public async Task TestGetUserById_InvalidId_ReturnsNotFound()
{
  // Definiere eine ungültige Benutzer-ID
  var invalidUserId = 9999;

  // Sende eine GET-Anfrage an den API-Endpunkt
  var response = await client.GetAsync($"https://example.com/api/users/{invalidUserId}");

  // Überprüfe, ob der Statuscode 404 (Not Found) zurückgegeben wird
  Assert.AreEqual(404, (int)response.StatusCode, $"Fehler: Statuscode {response.StatusCode}
    ");
}
```

4.5. Schritt 5: Integration in den CD/CI Prozess

Nach der Erstellung der Tests kann der Entwickler diese in den Continuous Integration (CI)-Prozess integrieren, sodass sie automatisch bei jedem Commit oder Pull Request ausgeführt werden. Dies gewährleistet, dass die API jederzeit stabil bleibt und der Endpunkt weiterhin korrekt funktioniert.

4.6. Schritt 6: Weiterführende Tests und manuelle Anpassungen

Obwohl GitHub Copilot bei der schnellen Erstellung grundlegender Tests geholfen hat, führt der Entwickler eine manuelle Überprüfung und Erweiterung durch. Beispielsweise könnte er zusätzliche Tests hinzufügen, um die API unter verschiedenen Bedingungen zu testen, oder sicherstellen, dass Authentifizierungsmechanismen wie OAuth oder API-Schlüssel korrekt implementiert sind.

4.7. Ergebnisse und Vorteile

Mit GitHub Copilot konnte der Entwickler die Tests für die API deutlich schneller erstellen, ohne die Testqualität zu gefährden. Besonders bei größeren Projekten und Teams hilft Copilot, wiederholbare Tests schnell zu generieren und zu integrieren, sodass die Qualität der Software gesichert bleibt. Dieses Beispiel zeigt, wie GitHub Copilot dazu beitragen kann, API-Tests in C# effizient zu erstellen, und die Notwendigkeit für manuelle Tests zu reduzieren, während es gleichzeitig sicherstellt, dass kritische Randfälle abgedeckt werden.

5. Herausforderungen und Best Practices

Obwohl GitHub Copilot bei der Erstellung von Tests äußerst hilfreich sein kann, gibt es auch einige Herausforderungen, die beachtet werden müssen:

5.1. Qualität der generierten Tests

Die Qualität der von Copilot generierten Tests hängt stark von der Qualität des Ausgangscodes und den Kommentaren ab. In einigen Fällen könnte Copilot ungenaue oder unvollständige Testfälle erzeugen. Entwickler sollten daher immer

sicherstellen, dass die generierten Tests gründlich überprüft und gegebenenfalls manuell angepasst werden.

5.2. Fehlende Domänenkenntnis

GitHub Copilot ist ein leistungsstarkes Tool, jedoch fehlt ihm die vollständige Verständnisfähigkeit für die spezifischen Anforderungen eines Projekts. Während es oft gute Vorschläge basierend auf allgemeinen Programmiermustern macht, kann es die komplexen geschäftlichen Logiken oder Randbedingungen eines Systems nicht immer korrekt antizipieren.

5.3. Vertrauen und Abhängigkeit

Es besteht die Gefahr, dass Entwickler sich zu sehr auf Copilot verlassen und aufhören, ihre Teststrategien regelmäßig zu hinterfragen. Ein unkritisches Vertrauen in Copilot kann dazu führen, dass Fehler unentdeckt bleiben. Entwickler sollten sicherstellen, dass Copilot lediglich als unterstützendes Werkzeug dient und nicht als Ersatz für das eigene Verständnis und die kritische Analyse.

5.4. Best Practices

- **Verwendung als Ergänzung:** GitHub Copilot sollte als Ergänzung und nicht als Ersatz für manuell geschriebene Tests verwendet werden. Entwickler sollten weiterhin sicherstellen, dass alle generierten Tests ihren spezifischen Anforderungen entsprechen.

- **Regelmäßige Code-Reviews**: Auch wenn Copilot Tests automatisch generiert, sollten Code-Reviews und manuelle Tests nicht vernachlässigt werden. Dies stellt sicher, dass die Teststrategie robust bleibt und alle relevanten Szenarien abgedeckt sind.

- **Schrittweise Einführung**: Entwickler sollten Copilot schrittweise in den Testprozess integrieren und sicherstellen, dass die generierten Tests durch eine sorgfältige manuelle Überprüfung validiert werden.

6. Fazit

GitHub Copilot hat das Potenzial, den Prozess des automatisierten Testens erheblich zu vereinfachen und zu beschleunigen. Durch die Integration von KI in den Entwicklungsworkflow können Entwickler schneller und effizienter hochwertige Tests erstellen. Dabei ist jedoch wichtig, dass Copilot als unterstützendes Werkzeug und nicht als vollständiger Ersatz für manuelles Testing und Fachwissen genutzt wird. Durch den kombinierten Einsatz von KI gestützten Tools und traditionellen Testmethoden kann die Softwarequalität signifikant verbessert werden.