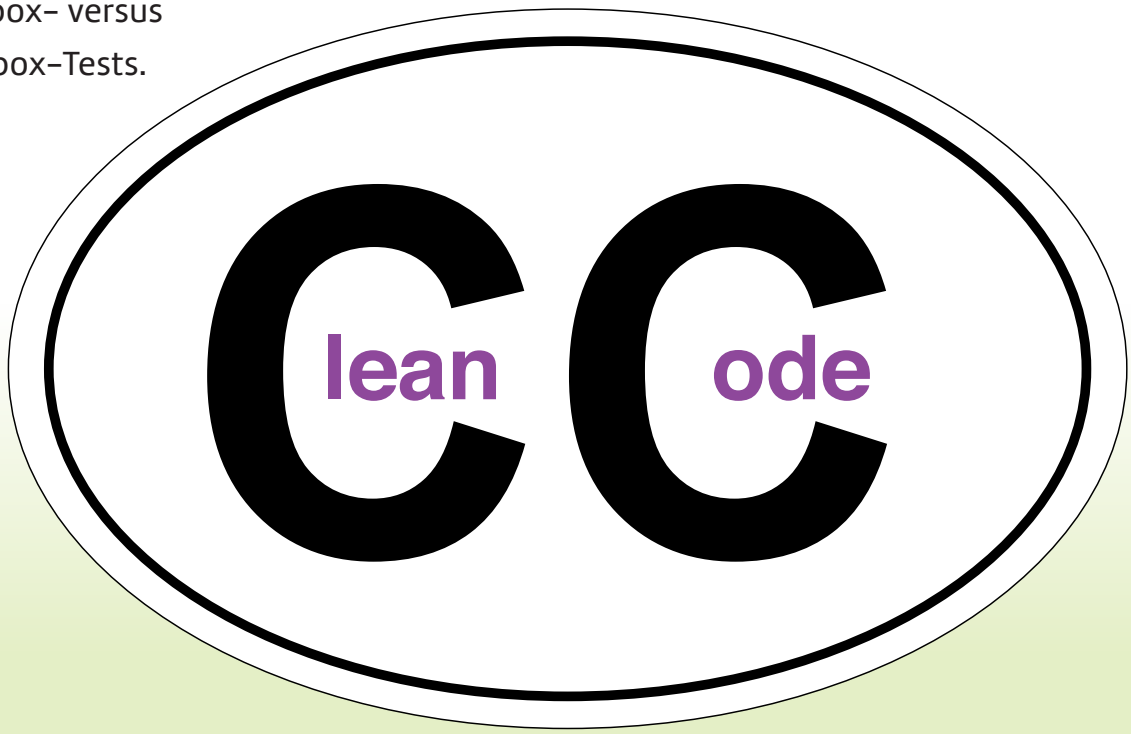
LIESERS CLEAN CODE

Zwischen Schwarz und Weiß liegt jede Menge Grau

Blackbox- versus Whitebox-Tests.



Von Teilnehmern in meinen Seminaren höre ich immer wieder den Einwand, dass man private Methoden nicht testen solle. Alle Tests mögen doch bitte ausschließlich durch das öffentliche API erfolgen. Es gibt hier also eine Präferenz für Blackbox-Tests. Sicher hat dies Vorteile. Doch meist stecken hinter einseitigen oder sogar dogmatischen Aussagen Potenziale, die man mit einer differenzierten Betrachtung heben kann. Darum soll es im Folgenden gehen.

Mit Blackbox-Test bezeichnet man einen Test, der die Implementation wie eine Blackbox betrachtet, deren Inhalt nicht nach außen sichtbar ist. Es wird die öffentliche Schnittstelle, das *public* API getestet. Solche Tests sind unbestritten notwendig, um herauszufinden, ob die öffentlich sichtbare Methode tut, was sie soll. Die Frage, die sich nun stellt, ist allerdings, ob es vorteilhaft ist, ausschließlich über das öffentliche API zu testen. Hat es ausschließlich Vorteile, nur mit Blackbox-Tests zu arbeiten? Welche Vorteile haben gegebenenfalls Whitebox-Tests? Damit werden Tests bezeichnet, die auch die Interna eines öffentlichen API kennen und testen.

Gegen Whitebox-Tests wird angeführt, dass damit die Tests von den Interna abhängen. Das hat zur Folge, dass Tests angepasst werden müssen, wenn sich die Interna ändern sollten. Vordergründig ist das Argument natürlich zutreffend. Jedoch lohnt es sich, einmal genauer hinzuschauen, welche Vor- und Nachteile jeweils mit Black- und Whitebox-Tests verbunden sind. Ich bin überzeugt, dass eine differenzierte Betrachtung zu einer besseren Ausschöpfung der Potenziale beider Vorgehensweisen führt und somit den Nutzen der Tests maximieren kann.

Worum geht es?

Im Kern geht es wieder einmal um Abhängigkeiten. Schauen wir uns dazu eine simple Struktur an, bestehend aus drei Methoden, wie in Bild 1 zu sehen.

Das Abhängigkeitsdiagramm ist so zu verstehen, dass die öffentliche Methode *f* die beiden privaten Methoden *f1* und *f2* aufruft. Durch den Aufruf ist *f* von *f1* und *f2* abhängig. Solange *f1* und *f2* nur von *f* beziehungsweise nur innerhalb der umschließenden Klasse verwendet werden, können sie als *private* markiert werden. Die Sichtbarkeit von *f1* und *f2* zu erhöhen, sodass sie auch außerhalb der Klasse verwendet werden können, würde nur Sinn ergeben, wenn dies inhaltlich zu begründen ist. Solange es sich um Implementationsdetails der Methode *f* handelt, gibt es keinen Grund, sie ebenfalls *public* zu machen.

Anhand dieses einfachen Beispiels können wir nun die Unterschiede zwischen Blackbox- und Whitebox-Tests darstellen. Bei reiner Blackbox-Vorgehensweise werden die Tests ausschließlich für die öffentliche Methode *f* geschrieben. Somit ist in den Tests nicht erkennbar, dass *f* die beiden privaten Methoden verwendet. Daraus ergibt sich der Vorteil, dass die Interna nicht im Test relevant sind. Ändert man dann die Implementation, können die Tests unangetastet bleiben. Insbesondere kann so mithilfe der Tests sichergestellt werden, dass die Implementation auch nach der Änderung noch korrekt ist, beziehungsweise dass sie sich zumindest im Bereich der Testabdeckung so verhält wie zuvor.

Jedoch hat diese Vorgehensweise auch einen Nachteil. Solange *f1* und *f2* nämlich nur implizit durch den Aufruf von *f* getestet werden, ist es gegebenenfalls aufwendig, Testfälle zu formulieren, mit denen eine ordentliche Testabdeckung von *f1* und *f2* erreicht wird. Insbesondere dann, wenn in *f* Logik enthalten ist, kann dies zu einer Herausforderung werden. Doch dazu später mehr.

Whitebox, ein Problem der Sichtbarkeit

Doch wie würde man nun Whitebox-Tests schreiben? Dazu müssen zunächst die beiden Methoden *f1* und *f2* für Tests erreichbar gemacht werden. Leider gibt es zwischen den Sichtbarkeiten *public* und *private* kein *testable private*. Damit könnte ich als Entwickler ausdrücken, dass die Methode *private* ist, jedoch für Tests sichtbar sein soll. Weil dieses Konzept in C# nicht existiert, müssen wir uns mit *internal* behelfen.

Mir ist bewusst, dass dies einen Kompromiss darstellt. Denn Methoden, die mit *internal* markiert sind, sind innerhalb der gesamten Assembly sichtbar, in der sie definiert sind. Besteht mein Softwaresystem nur aus einem einzigen Projekt, was dann zu einer einzigen Assembly führt, ist alles quasi *public*. Die Sichtbarkeit aufzuweichen mag zunächst abschrecken, doch es geht hier ja weiterhin um eine differenzierte Betrachtung, bei der man gegebenenfalls auch Nachteile in Kauf nehmen muss, um Vorteile zu ernten. Ist ein System sinnvoll auf mehrere Projekte verteilt, reduziert sich das Problem der *internal*-Methoden bereits deutlich.

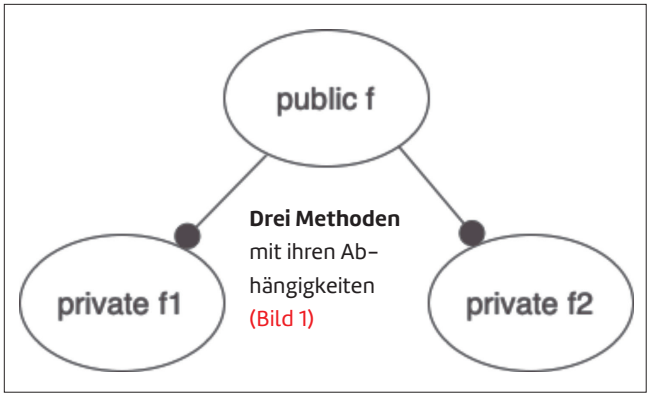
Die Verwendung von *internal* als Sichtbarkeit ist meiner Einschätzung nach ohnehin nur selten sinnvoll, sodass der Kompromiss vertretbar ist. Es muss dann im Team die Spielregel vereinbart werden, dass *internal* nur für Tests eingesetzt wird und ansonsten die gleiche Bedeutung hat wie *private*. Niemand darf sich also an *internal*-Methoden binden, obschon dies technisch möglich wäre.

Verwendet man bei Methoden, die man Whitebox testen möchte, die Sichtbarkeit *internal* und ergänzt man in der Assembly das Attribut *InternalsVisibleTo*, können diese Methoden in Tests aufgerufen werden. Damit ist das technische Problem der Sichtbarkeit gelöst. Alternativ können die Methoden *private* bleiben und man ruft sie über Reflection auf. In dem Fall sind allerdings Methodennamen in Strings abgelegt, was beim Refactoring eine zusätzliche Herausforderung bedeutet.

Sicher kann man das Thema vertiefen, wie *private*-Methoden für Tests am besten erreichbar gemacht werden. Lösungsansätze könnten auch mit Codegeneratoren oder Codewevern weiter verfolgt werden. Gehen wir hier zunächst davon aus, dass private Methoden rein technisch für Tests erreichbar gemacht werden können. Dann stellt sich immer noch die Frage, ob Whitebox-Tests sinnvoll sind, beziehungsweise welche Vor- und Nachteile eine solche Strategie mit sich bringt.

Kehren wir zum Beispiel der Methoden *f*, *f1* und *f2* aus Bild 1 zurück. Wenn wir im Whitebox-Test die Methoden *f1* und *f2* testen, ergibt sich daraus der Vorteil einer größeren Flexibilität in der Teststrategie. Wir können nun zwischen Integrationstests auf *f* und Unit-Tests auf *f1* und *f2* unterscheiden. Bei den Unit-Tests für *f1* und *f2* können wir somit potenziell leichter eine hohe Testabdeckung erreichen, da wir die Methoden unmittelbar aufrufen statt über den Aufruf von *f*.

Dazu ein Beispiel. Die folgende Implementation zeigt eine Lösung für das Problem „ToDictionary“. Die Funktion *ToDictionary* soll für einen String ein Dictionary erzeugen. Folgendes Beispiel gibt eine Idee dazu: Der String „a=1;b=2;c=3“ soll zum Dictionary {{„a“, „1“}, {„b“, „2“}, {„c“, „3“}} führen. Es muss also zunächst der gesamte String an den Semikolons



zerlegt werden. Anschließend müssen jeweils Key und Value, getrennt durch ein Gleichheitszeichen, ermittelt und in das Dictionary eingefügt werden.

Natürlich ist dieses Beispiel klein genug, um alles über das öffentliche API zu testen. Doch wir können davon abstrahieren und es auf kompliziertere Szenarien übertragen. Die Implementation ist in Listing 1 zu sehen.

Die öffentliche Methode *ToDictionary* ruft drei Methoden auf. Sofern diese nur innerhalb von *ToDictionary* benötigt werden, könnten sie *private* gemacht werden. Um sie aber einzeln testen zu können, habe ich sie *internal* gemacht. Das ermöglicht es mir im Test, die einzelnen Aspekte, aus denen sich die gesamte Lösung zusammensetzt, isoliert zu testen.

Um beispielsweise herauszufinden, zu welchem Ergebnis die Eingabe „a=1;;b=2“ (doppeltes Semikolon) führt, kann dies über einen Test der Methode *SplitIntoSettings* erfolgen. Diese Methode ist dafür zuständig, den String an den Semikolons in Teilstrings zu zerlegen. Ein Test könnte nun überprüfen, was mit der Eingabe „x;;y“ passiert. Werden die Strings „x“ und „y“ geliefert und das doppelte Semikolon ignoriert? Oder werden die Strings „x“, „“ und „y“ geliefert? Enthält also das Resultat einen leeren String? ▶

Listing 1: Implementation des ToDictionary-Beispiels

```
public class StringUtilities
{
  public IDictionary<string, string> ToDictionary(
      string configuration) {
    var settings = SplitIntoSettings(configuration);
    var keyValuePairs =
      SplitIntoKeyAndValue(settings);
    var dictionary = CreateDictionary(keyValuePairs);
    return dictionary;
  }

  internal IEnumerable<string> SplitIntoSettings(
      string configuration) {
    return configuration.Split(new [] {";"},
      StringSplitOptions.RemoveEmptyEntries);
  }

  internal IEnumerable<KeyValuePair<string, string>>
      SplitIntoKeyAndValue(IEnumerable<string> settings) {
    foreach (var setting in settings) {
      var keyAndValue = setting.Split('=');
      if (keyAndValue[0] == "") {
        throw new Exception();
      }
      yield return new KeyValuePair<string,
        string>(keyAndValue[0], keyAndValue[1]);
    }
  }

  internal IDictionary<string, string> CreateDictionary(
      IEnumerable<KeyValuePair<string, string>>
      keyValuePairs) {
    var result = new Dictionary<string, string>();
    foreach (var keyValuePair in keyValuePairs) {
      result[keyValuePair.Key] = keyValuePair.Value;
    }
    return result;
  }
}
```

Natürlich kann diese Fragestellung auch über den Integrationstest auf *ToDictionary* geprüft werden. Doch wie schon erwähnt handelt es sich hier um ein überschaubares Beispiel, an dem die Idee gezeigt werden soll. Im realen Code dürfte dieser Vorteil oft deutlich größer sein.

Fazit

Wir können eine Methode, die andere Methoden unserer Lösung aufruft, ausschließlich über das öffentliche API testen. In diesem Fall sprechen wir von Blackbox-Tests. Alternativ können wir zusätzlich auch die nicht öffentlichen Methoden testen und sprechen dann von Whitebox-Tests. Ich sehe bei beiden Strategien die folgenden Vor- und Nachteile:

Test auf *public f*:

- **Vorteil:** Test des öffentlichen API, somit können Interna leicht verändert werden.
- **Nachteil:** reine Integrationstests, wodurch die Testabdeckung eventuell schwieriger zu erreichen ist.

Test auf *public f* sowie *private f1*, *f2*:

- **Vorteil:** Kombination aus Integrations- und Unit-Tests.
- **Nachteil:** Sichtbarkeit aufgeweicht, daher eventuell schwieriger zu refaktorisieren.
- **Herausforderung:** *private*-Methoden müssen für den Test erreichbar gemacht werden.

Es ist zu einfach, sich für eines der beiden Extreme zu entscheiden. Whitebox-Tests kategorisch abzulehnen ist genauso kurzsichtig, wie die Herausforderungen von Whitebox-Tests zu ignorieren. Eine fachkundige Entscheidung und Auswahl ist gefragt. Code, der sehr starken Veränderungen unterliegt, sollte man eher mit Blackbox-Tests abdecken. So ist es einfacher, die Interna zu ändern, und man erhält durch die Tests die Rückmeldung, ob dabei etwas kaputtgegangen ist. Wenn es dagegen eher unwahrscheinlich ist, dass die Interna geändert werden, liegt in der Whitebox-Strategie der große Vorteil, dass hier mit einer guten Kombination aus Integrations- und Unit-Tests eine hohe Testabdeckung erreicht werden kann und gleichzeitig die einzelnen Tests überschaubar bleiben.

Vielleicht werden Sie denken, dass die Wahrscheinlichkeit für Änderungen der Interna immer hoch ist, und somit stark zu Blackbox-Tests tendieren. Lassen Sie sich in diesem Fall einmal darauf ein, es mit der sorgfältigen Planung einer Lösung zu versuchen. Als Entwickler tendieren wir dazu, zu schnell ins Codieren zu springen, statt zuvor über die Lösung nachzudenken und einen Entwurf zu zeichnen. In der Folge müssen wir dann die Interna häufig ändern, weil wir feststellen, dass die Lösung noch nicht ausreichend durchdacht war. Doch glauben Sie mir: Nachdenken hilft. ■

Stefan Lieser
sucht ständig nach Verbesserung und neuen Wegen, um die innere Qualität von Software zu optimieren. Gemeinsam mit Ralf Westphal hat er die Clean Code Developer Initiative (https://clean-code-developer.de) ins Leben gerufen.
https://lieser-online.de

dnpCode A2112CleanCode