LIESERS CLEAN CODE

Don't Repeat Yourself! – Und wenn doch?

Ein einzelnes Prinzip zu beachten reicht oft nicht: Die Einhaltung des DRY–Prinzips kann schnell zur Verletzung des IOSP führen.

Ein wichtiges Prinzip der Softwareentwicklung besagt, dass wir uns im Code nicht wiederholen sollten: Don't Repeat Yourself (DRY). Dopplungen sind zu vermeiden, weil sie gleich drei Clean-Code-Developer-Werte [1] gefährden: Dopplungen haben einen schlechten Einfluss auf die Korrektheit, die Wandelbarkeit und die Produktionseffizienz.

Wenn ein Codebereich einen Bug enthält und dann munter kopiert wird, ist der Bug gleich mitkopiert. Insofern ist es einleuchtend, dass DRY-Verletzungen schlecht für die Korrektheit sind. Die Wandelbarkeit wird erschwert, weil die Kopien häufig leicht modifiziert wurden. DRY-Verletzungen liegen selten 1:1 vor, sondern an der Kopie wurden kleine Veränderungen vorgenommen. Diese Änderungen im Nachhinein zu erkennen und zu verstehen erschwert die Wandelbarkeit häufig. Die Produktionseffizienz ist in Gefahr, weil durch DRY-Verletzungen nun potenziell an mehreren Stellen geändert werden muss. Das bedeutet, dass Änderungen gegebenenfalls mehrfach durchgeführt werden, was weniger effizient ist, als eine einzige Stelle zu ändern.

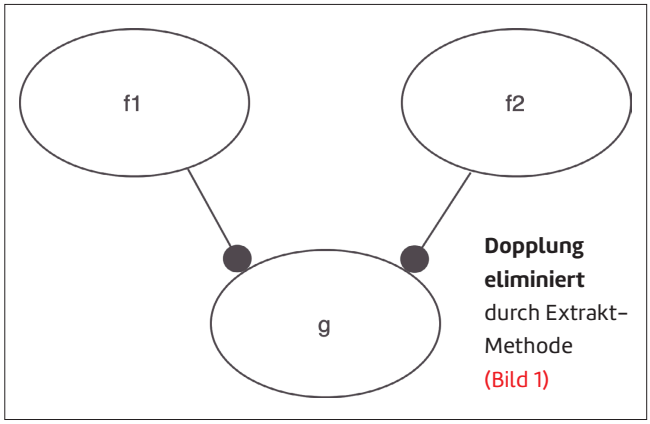
Hat ein Entwickler eine DRY-Verletzung erkannt, entsteht die Frage, wie das Problem behoben werden kann. In der Regel wird dann geraten, den doppelten Code in eine eigene Methode zu verlagern, um diese an mehreren Stellen aufrufen zu können. Bild 1 zeigt dies schematisch.

In den Methoden *f1* und *f2* lag eine DRY-Verletzung vor. Durch ein Extract Method Refactoring wurde diese in die Methode *g* herausgezogen. Damit ist die Dopplung beseitigt und das DRY-Prinzip eingehalten. Listing 1 zeigt das beispielhaft.

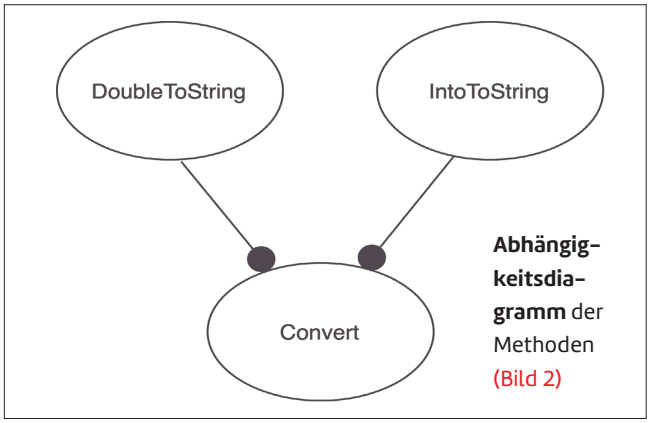
In diesem Beispiel sind die beiden Methoden *DoubleToString* sowie *IntToString* implementiert. Sie dienen dazu, einen *double*- beziehungsweise *int*-Wert in seine String-Repräsentation zu konvertieren. Es kommt hier zu folgenden Dopplungen: Innerhalb der *DoubleToString*-Methode ist ein Codeabschnitt doppelt vorhanden. Der Unterschied liegt lediglich in der Benennung der Variablen (*wholeNumber* versus *fraction* und *sWholeNumber* versus *sFraction*). Ferner kommt es zu einer Dopplung über die beiden Methoden hinweg, da der gleiche Codeausschnitt auch noch einmal in der Methode *IntToString* auftaucht.

Der klassische Umgang mit einer solchen Dopplung lautet: Extract Method. Zieht man die Dopplungen mit einem Extract Method Refactoring heraus und benennt einige Variablen um, entsteht die Lösung, die Sie in Listing 2 sehen. In Bild 2 ist die refaktorisierte Lösung schematisch dargestellt.

Damit ist das DRY-Problem eliminiert. Doch nun kommt das große „Aber": Nun ist das IOSP-Prinzip in der Methode *DoubleToString* verletzt! Die Abkürzung IOSP steht für „Integration Operation Segregation Principle". Das besagt, dass eine Methode entweder andere Methoden integriert oder Details enthält. Die Methode *DoubleToString* ruft die Methode *Convert* auf. Darüber hinaus enthält die Methode aber auch Details. Folgendes ist in den beiden Kategorien jeweils erlaubt:



Dopplung eliminiert durch Extrakt–Methode (Bild 1)



Abhängigkeitsdiagramm der Methoden (Bild 2)

Listing 1: Code mit Dopplungen

```
public static class ToStringUtils
{
  public static string DoubleToString(double d) {
    var wholeNumber = d;
    var sWholeNumber = "";

    do {
      var i = (int)(wholeNumber % 10);
      sWholeNumber = (char)(i + '0') + sWholeNumber;
      wholeNumber /= 10;
    } while ((int)wholeNumber > 0);

    var sFraction = "";
    var fraction = d % 1 * 10000.0;
    do {
      var i = (int)(fraction % 10);
      sFraction = (char)(i + '0') + sFraction;
      fraction /= 10;
    } while ((int)fraction > 0);

    return sWholeNumber + "." + sFraction;
  }

  public static string IntToString(int d) {
    var wholeNumber = d;
    var sWholeNumber = "";

    do {
      var i = wholeNumber % 10;
      sWholeNumber = (char)(i + '0') + sWholeNumber;
      wholeNumber /= 10;
    } while (wholeNumber > 0);

    return sWholeNumber;
  }
}
```

- **Integration:** Darf andere Methoden der Lösung aufrufen.
- **Operation:** Darf Methoden der Runtime oder aus Frameworks aufrufen (sogenannte APIs) und darf Ausdrücke enthalten (beispielsweise „x + 1“ oder „y < 42“)

Anhand dieser einfachen Regeln kann eine Methode der einen oder anderen Kategorie eindeutig zugeordnet werden. Werden die beiden Aspekte Integration und Operation vermischt, hat dies häufig negative Auswirkungen auf die Wandelbarkeit und die Korrektheit. Nehmen wir dazu ein simples Beispiel:

```
void f(int x) {
  if (x < 42) {
    g();
  }
}
```

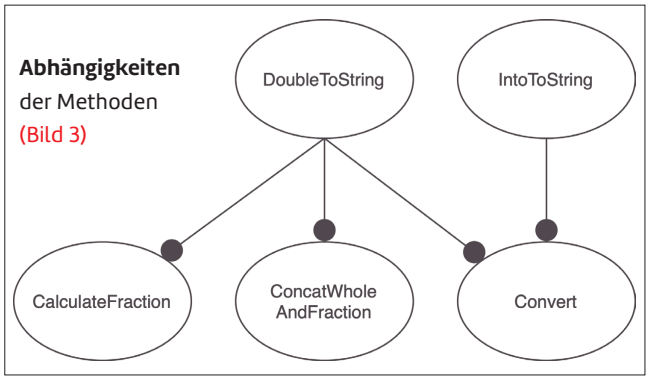
Listing 2: Code nach Extract Method Refactoring

```
public static class ToStringUtils
{
  public static string DoubleToString(double d) {
    var wholeNumber = d;
    var sWholeNumber = Convert(wholeNumber);

    var fraction = d % 1 * 10000.0;
    var sFraction = Convert(fraction);

    return sWholeNumber + "." + sFraction;
    }

  public static string IntToString(int d) {
    var sWholeNumber = Convert(d);
    return sWholeNumber;
  }

  private static string Convert(double d) {
    var result = "";
    do {
      var i = (int)(d % 10);
      result = (char)(i + '0') + result;
      d /= 10;
    } while ((int)d > 0);
    return result;
  }
}
```

Die Methode *f* ist eine Integrationsmethode, da sie eine andere Methode der Lösung, nämlich *g*, aufruft. Gleichzeitig steht in der Integrationsmethode *f* der Ausdruck *x < 42*. Das IOSP ist in *f* verletzt. Welche Konsequenzen ergeben sich daraus? Warum ist es so wichtig, das IOSP einzuhalten?

In diesem Fall kann der Ausdruck *x < 42* nur automatisiert getestet werden, indem die Methode *f* aufgerufen wird. Das führt allerdings dazu, dass dann auch *g* aufgerufen wird, sofern *x* kleiner als *42* ist. Wie lässt sich das in einem automatisierten Test überprüfen? Die gängige Antwort lautet: Interface, Dependency Injection und Mock Framework. Mit diesen Werkzeugen ist ein Entwickler in der Lage, im Test eine Attrappe für *g* zu verwenden. So kann der Test prüfen, ob die Attrappe für *g* aufgerufen wurde oder nicht. Wird also das Dependency Inversion Principle (DIP) eingehalten, lässt sich der Ausdruck *x < 42* automatisiert testen.

Doch warum so kompliziert? Das Problem ist hier, dass in *f* die Aspekte Integration (der Aufruf von *g*) und Operation (der Ausdruck *x < 42*) vermischt sind. Schauen wir uns an, was passiert, wenn man die Methode auf andere Weise implementiert und dabei das IOSP einhält: ▶



Abhängigkeiten der Methoden (Bild 3)

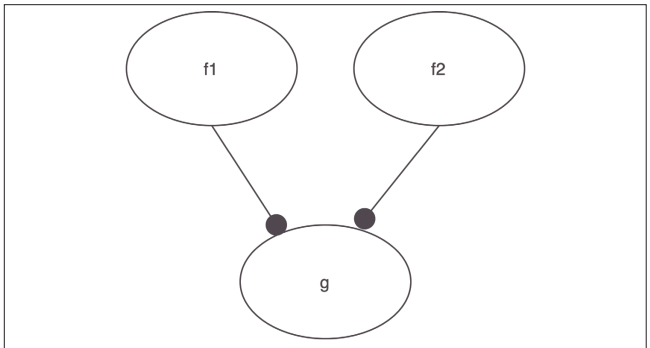
```
void f(int x) {
  if (h(x)) {
    g();
  }
}

bool h(int x) {
  return x < 42;
}
```

Nun ist *f* eine reine Integration. Die Verantwortlichkeit der Methode besteht darin, *h* und *g* aufzurufen. Durch das Herausziehen des Ausdrucks in eine eigene Funktion *h* kann der Ausdruck nun isoliert getestet werden. Zusätzlich kann immer noch ein Integrationstest von *f* erfolgen. Wir haben somit mehr Möglichkeiten beim Testen erlangt.

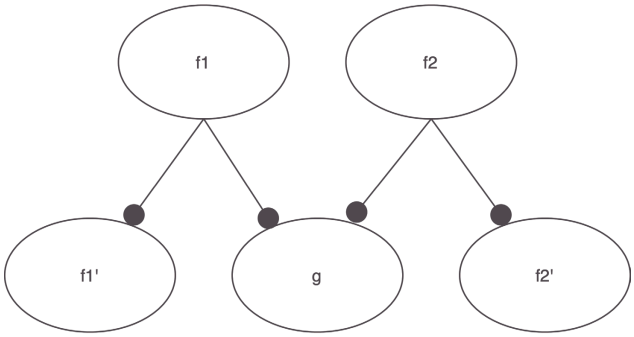
Beachten Sie, dass das IOSP ein Prinzip ist – und kein Dogma. Dies gilt im Übrigen für alle Clean-Code-Developer-Prinzipien. Ein Dogma müsste immer eingehalten werden, koste es, was es wolle. Ein Prinzip sollte eingehalten werden, weil sich daraus sehr oft Vorteile ergeben. Und gleichzeitig gibt es eben auch Fälle, in denen sich nur ein geringer oder gar kein Vorteil ergibt. In diesem Fall kann ich mich als erfahrener Entwickler über ein Prinzip hinwegsetzen. Ich vergleiche das gerne mit dem Weg zum Bäcker zu Fuß am Sonntagmorgen um 6 Uhr (Na gut, wer macht das schon? Betrachten Sie es als Gedankenexperiment). Bleibe ich am Sonntagmorgen um 6 Uhr zu Fuß an einer roten Fußgängerampel stehen, obschon weit und breit kein Auto zu sehen ist? Nein, natürlich nicht. Bleibt ein vierjähriges Kind in derselben Situation an der roten Ampel stehen? Hoffentlich ja! Der Unterschied liegt darin, dass der erfahrene Fußgänger die Situation einschätzen kann. Daher ist er in der Lage, sich über die Regel hinwegzusetzen. Ohne die Erfahrung droht Lebensgefahr.

So ist es auch mit den Clean-Code-Developer-Prinzipien. Für einen erfahrenen Entwickler stehen nicht die Prinzipien, sondern die Werte im Vordergrund. Wenn trotz Verletzung des Prinzips den Werten Wandelbarkeit und Korrektheit Genüge getan ist, kann ich mich im Einzelfall über ein Prinzip hinwegsetzen. Allerdings muss ich dazu über die Erfahrung verfügen. Im Zweifel sollte ich also eher das Prinzip einhalten.

Doch zurück zum DRY-Problem. Durch das Herausziehen des gemeinsamen Codes verletzt nun die Methode *DoubleToString* das IOSP. Besser ist es daher, das DRY-Problem so zu lösen, wie es Listing 3 zeigt. In Bild 3 sehen Sie die Abhängigkeitsstruktur dieser Lösung.



DRY und IOSP im Zusammenspiel (Bild 4)



Diese Lösung hält nun sowohl das DRY-Prinzip als auch das IOSP ein. *DoubleToString* ruft nur noch Methoden der Lösung auf und ist damit eine reine Integration. Die anderen Methoden sind reine Operationen, da sie nur Framework-/Runtime-Methoden aufrufen beziehungsweise Ausdrücke enthalten. Ich sehe zwei Vorteile in dieser Variante. Erstens dient diese

Listing 3: DRY und IOSP sind eingehalten

```
public static string DoubleToString(double d) {
  var wholeNumber = d;
  var fraction = CalculateFraction(d);

  var sWholeNumber = Convert(wholeNumber);
    var sFraction = Convert(fraction);

  return ConcatWholeAndFraction(
    sWholeNumber, sFraction);
}

private static string ConcatWholeAndFraction(
    string sWholeNumber, string sFraction) {
  return sWholeNumber + "." + sFraction;
}

private static double CalculateFraction(double d) {
  return d % 1 * 10000.0;
}
```

Lösung der Wandelbarkeit, da der Code nun abstrakter ist. Als Leser der Methode *DoubleToString* muss ich mich nicht mehr mit den Details befassen. Ich muss nicht interpretieren, was die Bedeutung eines Ausdrucks sein könnte. Stattdessen lese ich Methodenaufrufe. In den ausgelagerten Methoden stehen dagegen ausschließlich Details. Auch das ist in Verbindung mit dem Single Responsibility Principle (SRP) eine gute Sache, weil die Methoden dadurch leicht verständlich sind.

Neben der Wandelbarkeit ergibt sich hier auch ein Vorteil für die Korrektheit. Als Entwickler habe ich nun die Wahl, ob ich ausschließlich über das öffentliche API *DoubleToString* testen möchte oder ob ich die Operationen *Convert*, *CalculateFraction* und *ConcatWholeAndFraction* zusätzlich mit Unit-Tests überprüfe. Dabei handelt es sich nicht mehr um Black-Box-Tests, da die internen Details getestet werden. Doch auch hier lohnt es sich, die dogmatische Sichtweise zu lockern. Es gilt immer abzuwägen, wie stark die einzelnen Prinzipien und Konzepte gewichtet werden. Manchmal bieten reine Black-Box-Tests einen Vorteil, weil ich damit leichter in der Lage bin, die interne Implementation zu verändern. Manchmal bieten aber White-Box-Tests einen Vorteil, weil sie es leichter machen, mit der kombinatorischen Explosion von Testfällen umzugehen.

In Bild 4 sehen Sie nochmals am abstrakten Beispiel, wie sich bei einem Refactoring eine Kombination aus DRY und IOSP entwickelt. Zunächst wird aus *f1* und *f2* der gemeinsame Code nach *g* herausgezogen. Dadurch ist das DRY-Problem behoben. Unterstellt, dass in *f1* und *f2* nun IOSP-Probleme auftreten, werden die Details ebenfalls herausgezogen. So entstehen die Methoden *f1'* und *f2'*. Damit bleiben *f1* und *f2* als reine Integrationsmethoden zurück, während *g*, *f1'* und *f2'* Operationen sind.

Fazit

Es ist nicht immer damit getan, ein einzelnes Prinzip zu beachten. Manchmal entstehen aus einem Refactoring im Sinne eines Prinzips neue Herausforderungen. Im Falle des DRY-Prinzips entsteht meist die Herausforderung, sich im Anschluss mit dem IOSP auseinandersetzen zu müssen. Dieser nächste Schritt lohnt sich, weil der Code leichter verständlich wird (Wandelbarkeit) und mehr Flexibilität beim Testen (Korrektheit) entsteht. ■

[1] https://clean-code-developer.de/das-wertesystem

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 A2111CleanCode