Schritt 5: Aufräumen Als Folge dieser Vereinfachungen wird ersichtlich, dass die Variable `strFaultCode` nun unbenutzt und damit überflüssig ist. Damit kann sie entfallen. Auch die Variable `strSource` ermittelt nur Daten aus dem übergebenen `Fault`-Objekt. Wir schreiben einfacher Folgendes:

```
catch (final Fault fault)
{
    this.strErrorMsg = fault.getFaultString() + ": " + fault.getSource();
}
```

Fazit

Das Resultat ist beeindruckend: Aus zehn Zeilen Sourcecode ist eine Zeile geworden. Ähnlich wie sich Bad Smells ausbreiten, wenn man unachtsam ist, gibt es glücklicherweise auch einen gegenteiligen Effekt: ***Je mehr man für Klarheit und Struktur sorgt, desto leichter fallen weitere Verbesserungsmaßnahmen.***

Jedoch haben wir ein Problem nicht entfernen können: Nach wie vor gibt es eine Zuweisung an das Attribut `strErrorMsg` im `catch`-Block. Ein solcher Seiteneffekt ist zu vermeiden, da es an unerwarteter Stelle im Programm zu Zustandsänderungen kommt. ***Durch Refactorings wollen wir normalerweise das nach außen sichtbare Programmverhalten nicht verändern, sondern lediglich die innere Struktur verbessern***. An dieser Stelle können wir im Sourcecode einen Kommentar mit einem Hinweis auf diesen Seiteneffekt einfügen und das Ganze später nochmals prüfen bzw. überarbeiten.

16.3 Kombination von Basis-Refactorings

Bevor wir uns einen Katalog einiger komplexerer Refactorings – und zum Teil genau genommen sogar leicht verhaltensverändernder Transformationen des Sourcecodes – anschauen, wollen wir zunächst an einem Beispiel verschiedene durch die IDE unterstützte Basis-Refactorings, wie sie Martin Fowler in seinem Buch »Refactoring: Improving the Design of Existing Code« [20] beschreibt, betrachten. Die Basis-Refactorings zeichnen sich dadurch aus, dass sie oftmals keine Modifikation am sichtbaren Verhalten vornehmen. Die hohe Kunst ist es, die Schritte so klein und sicher zu gestalten, dass es dabei möglichst selten zu Kompilierfehlern oder anderweitigen Problemen kommt. Damit es klappt, muss man sich der Refactoring-Automatiken aus der IDE bedienen. Die Wahrscheinlichkeit für Komplikationen steigt, wenn man eher freihändig refaktorisiert.

16.3.1 Refactoring-Beispiel: Ausgangslage und Ziel

Als Ausgangsbasis dient eine Utility-Klasse `TimeStampUtils` mit einer Methode `createTimeStampString()`. Schauen wir zunächst auf einen Aufruf eines Nutzers:

```
final String timeStamp = TimeStampUtils.createTimeStampString(currentPeriod,
                                                               frequency);
```

Weil der Aufruf unproblematisch scheint, betrachten wir nun die statische öffentliche Methode an sich, um mögliche Schwachpunkte zu erkennen:

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final LocalDateTime start = currentPeriod.getDateTime();
    final int divisor = frequency == ComplexFrequency.P1M ? 1 : 3;
    final String addition = frequency == ComplexFrequency.P1M ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Ein erster Blick zeigt eine vermeintlich einfache Realisierung, die Jahresangaben gefolgt von Monat oder Quartal ausgeben soll.[4] Das Ganze ist recht kurz, aber vielleicht durch die Abfragen mit dem `?`-Operator ein wenig unübersichtlich. Problematischer ist jedoch, dass die Methode unerwünschte Abhängigkeiten auf die zwei Klassen `ExtTimePeriod` und `ComplexFrequency` besitzt, die aus einem externen Package (`external`) stammen. Ein genauerer Blick offenbart zusätzlich folgende Probleme:

- Die Methode scheint für beliebige Frequenzen des Typs `ComplexFrequency` ausgelegt zu sein. Tatsächlich ist sie es aber nicht, denn durch einen versteckten Logikfehler wird alles außer der Frequenz monatlich auf Quartale abgebildet. Dadurch können nur Monats- oder Quartalswerte korrekt verarbeitet werden.
- Es ist unklar, welches der gewünschte Rückgabewert ist. Gerade im Bereich von Datumsarithmetik findet man 0- oder 1-basierte Werte: Startet `getMonthValue()` also mit 0 oder 1? Und wieso erfolgt eine Subtraktion von 1?

Für die nachfolgenden Refactorings steht zunächst die Auflösung der Abhängigkeiten im Fokus. Auf die beiden anderen Details der Verarbeitung gehe ich später ein.

Definition des Ziels

Die Methode `createTimeStampString(ExtTimePeriod, ComplexFrequency)` soll nun mithilfe von Basis-Refactorings so umgestaltet werden, dass nur Abhängigkeiten auf Standards wie Klassen aus öffentlichen Bibliotheken oder besser noch dem JDK bestehen und der Sourcecode verständlicher wird. Bevor wir mit den Umbauarbeiten beginnen, erstellen wir ein UML-Klassendiagramm von der Ausgangslage und insbesondere auch von einem möglichen Zieldesign. Beides ist in Abbildung 16-1 dargestellt. Das gezeigte Ziel ist nicht ganz starr, sondern eher ein Anhaltspunkt, da man beim Entwickeln gewöhnlich immer noch kleinere Änderungen vornimmt. Das ist auch der Grund, warum wir hier keine Typparameter in den Signaturen angeben.

[4] Dabei wird auf `null`-Prüfungen verzichtet, weil es sich um eine interne Hilfsklasse handelt und wir uns hier auf die Refactoring-Schritte konzentrieren wollen.

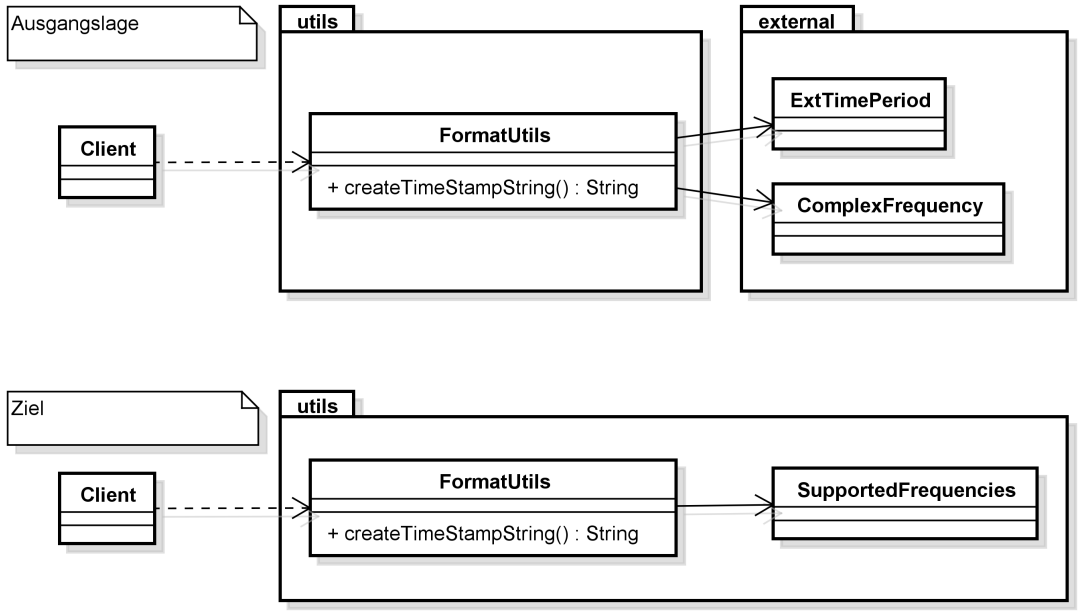

Abbildung 16-1 *Refactoring der Methode* `createTimeStampString()`

Wir wollen nun folgende Schritte ausführen, um die angemerkten Probleme zu beseitigen und das dargestellte Ziel zu erreichen:

- **Auflösen der Abhängigkeiten** – In einem ersten Schritt lösen wir die Abhängigkeiten zum Package `external` auf, indem wir einen `enum` namens `SupportedFrequencies` als Ersatz für `ComplexFrequency` einführen und anstelle der Klasse `ExtTimePeriod` die Klasse `LocalDateTime` aus dem Date and Time API (vgl. Kapitel 8) verwenden.
- **Vereinfachungen** – Einige der Berechnungen in der Methode sind etwas komplex und nicht gut zu lesen. Wir werden ein paar Vereinfachungen vornehmen.
- **Verlagern von Funktionalität** – Abschließend schauen wir, wie wir durch eine kleine Änderung von Zuständigkeiten für mehr Klarheit im Design sorgen.

16.3.2 Auflösen der Abhängigkeiten

Um den Sourcecode klarer und besser verständlich zu gestalten, werden wir folgende Refactorings (mit Windows-Tastaturkürzeln in Eclipse in Klammern) nutzen:[5]

- EXTRACT LOCAL VARIABLE (ALT+SHIFT+L)
- EXTRACT METHOD (ALT+SHIFT+M)
- INLINE (ALT+SHIFT+I)
- CHANGE METHOD SIGNATURE (ALT+SHIFT+C)

[5] Für Mac OS ist es statt ALT+SHIFT die Kombination ALT+COMMAND. Oftmals bietet die in Eclipse integrierte QUICK-FIX-Funktionalität, die man durch CTRL+1 aufruft, die Möglichkeit, die ersten drei Refactorings direkt auszuführen.

Schritt 1: Hilfsvariable einführen (EXTRACT LOCAL VARIABLE)

Zum leichteren Verständnis wird die zu bearbeitende Methode nochmals gezeigt:

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final LocalDateTime start = currentPeriod.getDateTime();
    final int divisor = frequency == ComplexFrequency.P1M ? 1 : 3;
    final String addition = frequency == ComplexFrequency.P1M ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Als Erstes selektieren wir den Ausdruck `ComplexFrequency.P1M` und nutzen das Refactoring EXTRACT LOCAL VARIABLE (ALT+SHIFT+L), um die lokale Variable `isMonthly` zu extrahieren, wodurch sich der Sourcecode vereinfachen lässt:

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final LocalDateTime start = currentPeriod.getDateTime();
    final boolean isMonthly = frequency == ComplexFrequency.P1M;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Schritt 2: Abhängigkeit zur Frequenz entfernen (EXTRACT METHOD)

Als Nächstes wollen wir die Abhängigkeit zur Klasse `ComplexFrequency` auflösen und dazu eine überladene Variante der Methode `createTimeStampString()` erzeugen. Um diese aus der Originalmethode zu extrahieren, ordnen wir einige Zeilen um und nutzen dazu die Tastaturkürzel ALT+UP/DOWN. Damit verschieben wir die boolesche Variable `start` direkt zu der ersten Verwendung, also vor die Definition von `value`. Die Variable `isMonthly` schieben wir an den Methodenanfang:[6]

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final boolean isMonthly = frequency == ComplexFrequency.P1M;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final LocalDateTime start = currentPeriod.getDateTime();
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

[6] Bitte beachten Sie, dass diese Umordnung hier zu keiner Verhaltensänderung führt, aber dass das in der Praxis z. B. wegen möglicherweise versteckter Seiteneffekte nicht immer so ist. Man spricht in dem Zusammenhang von ***Temporal Coupling*** (vgl. Abschnitt 16.6).

Danach selektieren wir alle Zeilen nach der Definition von `isMonthly` und setzen das Refactoring EXTRACT METHOD (ALT+SHIFT+M) ein. Damit entsteht eine gleichnamige Methode mit der Sichtbarkeit `public`, die als Parametertyp `boolean` statt `ComplexFrequency` besitzt.

```
@Deprecated
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final boolean isMonthly = frequency == ComplexFrequency.P1M;
    return createTimeStampString(currentPeriod, isMonthly);
}

public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final boolean isMonthly)
{
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final LocalDateTime start = currentPeriod.getDateTime();
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Die unerwünschte Abhängigkeit zur Klasse `ComplexFrequency` wurde damit aufgelöst – allerdings durch einen booleschen Parameter, was meistens kein gutes Design ist. Später komme ich darauf zurück und wir beheben auch diese Schwachstelle.

Indem wir die ursprüngliche Methode als `@Deprecated` markieren, signalisieren wir, dass zukünftige Nutzer stattdessen die neu erstellte Methode verwenden sollten. Für die bisherigen Aufrufer hat sich nichts geändert.

Schritt 3: Inlining des Methodenaufrufs (INLINE)

Da wir die Abhängigkeit zur Klasse `ComplexFrequency` in der Utility-Klasse eliminieren wollen, sollte überall die neu erstellte statt der alten Methode eingesetzt werden.

Die Aufrufstellen sind ähnlich zu folgender:

```
final String timeStamp = createTimeStampString(currentPeriod, frequency);
```

Die Aufrufstellen könnten wir zwar von Hand korrigieren, aber es ist sinnvoller und weniger fehleranfällig, dazu die in die IDE integrierten Refactorings zu nutzen.[7] Dazu markieren wir den Namen der ursprünglichen Methode und nutzen dann das Refactoring INLINE (ALT+SHIFT+I). Dieses transformiert alle Aufrufstellen folgendermaßen:

```
final boolean isMonthly = frequency == ComplexFrequency.P1M;
final String timeStamp = createTimeStampString(currentPeriod, isMonthly);
```

Optional kann die nicht mehr benötigte Methode automatisch gelöscht werden, wodurch nur noch die zuvor extrahierte, neue Methode in der Utility-Klasse verbleibt.

[7] Allerdings sollten wir uns dabei bewusst sein, dass sich die Verarbeitungsreihenfolge durch das INLINE ändern kann. Auch hier kann also Temporal Coupling eine Rolle spielen.

Die bisher durchgeführten Änderungen lassen erahnen, dass sich für Umgestaltungen die Nutzung von Refactoring-Automatiken anbietet, um die Wahrscheinlichkeit für Fehler zu reduzieren und für konsistente Änderungen zu sorgen.

Schritt 3a (optional): Inlining der Hilfsvariablen (INLINE)

Die Aufrufstellen sehen nun nach Schritt 3 – unter anderem durch den booleschen Übergabeparameter – etwas ungelenk aus, was wir später noch mit einer Designänderung adressieren werden. Zunächst könnte man in einem weiteren Schritt statt der lokalen Variablen den Ausdruck direkt als Methodenparameter angeben. Dabei hilft wiederum das Refactoring INLINE (ALT+SHIFT+I), diesmal für die Variablendeklaration. Zum Ausführen ist die Variable `isMonthly` zu selektieren:

```
final boolean isMonthly = frequency == ComplexFrequency.P1M;
final String timeStamp = createTimeStampString(currentPeriod, isMonthly);
```

Durch das Refactoring INLINE wird der Aufruf folgendermaßen abgewandelt:

```
final String timeStamp = createTimeStampString(currentPeriod,
                                                frequency == MyFrequency.P1M);
```

Jedoch reduziert dieser Schritt mitunter die Verständlichkeit und Lesbarkeit.

Schritt 4: Abhängigkeit zur Klasse `ExtTimePeriod` entfernen

Kommen wir wieder zu der eigentlichen Methode `createTimeStampString()` zurück. Wir wollen nun die Abhängigkeiten auf die Klasse `ExtTimePeriod` auflösen. Dabei hilft uns ein scharfer Blick auf die Methode und das Refactoring EXTRACT METHOD: Abgesehen von der ersten Zeile der Methode selektieren wir den Rest und extrahieren eine gleichnamige Methode. Die Utility-Klasse sieht wie folgt aus, nachdem wir die alte Methode noch als `@Deprecated` markiert haben:

```
@Deprecated
public static String createTimeStampString(final MyTimePeriod currentPeriod,
                                            final boolean isMonthly)
{
    final LocalDateTime start = currentPeriod.getDateTime();
    return createTimeStampString(isMonthly, start);
}

public static String createTimeStampString(final boolean isMonthly,
                                            final LocalDateTime start)
{
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Wie schon zuvor, hat das Refactoring keine Auswirkungen auf bisherige Nutzer, außer, dass diese nun durch das Hinzufügen von `@Deprecated` auf unsere Änderungen in der Utility-Klasse aufmerksam gemacht werden.

Schritt 5: Für konsistente Parameterreihenfolge sorgen

Beim Extrahieren der Methode fällt uns auf, dass durch die Automatik die Parameterreihenfolge vertauscht wurde und `isMonthly` nun der erste Parameter ist. Das wollen wir korrigieren. Dazu nutzen wir das Refactoring CHANGE METHOD SIGNATURE (ALT+SHIFT+C) und vertauschen die beiden Parameter, womit wir wieder eine konsistente Reihenfolge erzielen.

Schritte 6: Inlining des Methodenaufrufs

Wir führen weitere Aufräumarbeiten aus und selektieren die alte Methode und nutzen das Refactoring INLINE. Dadurch verdichten wir die Utility-Klasse:

```
public static String createTimeStampString(final LocalDateTime start,
                                            final boolean isMonthly)
{
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Auch die Aufrufstelle wird automatisch durch die IDE angepasst:

```
final LocalDateTime start = currentPeriod.getDateTime();
final String timeStamp = createTimeStampString(start,
                                                frequency == EasyFrequency.P1M);
```

Schritte 6a (optional): Inlining der Hilfsvariablen

Wenn gewünscht, kann man nochmals das Refactoring INLINE ausführen, um die Hilfsvariable zu entfernen und direkt in den Methodenaufruf zu integrieren:

```
final String timeStamp = createTimeStampString(currentPeriod.getDateTime(),
                                                frequency == EasyFrequency.P1M);
```

Zwischenfazit

Die erstellte Methode besitzt keine Abhängigkeiten auf externe Klassen mehr oder zumindest nur auf Klassen, die Standardbibliotheken des JDKs entstammen. Damit lässt sich das Ganze viel einfacher mit Unit Tests überprüfen. Bisher haben wir allerdings keine Unit Tests ausgeführt, insbesondere weil es schlicht keine gab, was wir nun ändern wollen, und weil wir lediglich sichere Basis-Refactorings angewendet haben.

Tests

Die zuvor genutzten Refactorings haben das nach außen sichtbare Verhalten nicht geändert, was aber selbst für die Basis-Refactorings nicht immer gilt.[8] Allerdings sind diese deutlich sicherer, als die Refactorings von Hand auszuführen.

Bei Änderungen empfiehlt sich generell die Ausführung von Unit Tests. Weil es noch keine gibt, erstellen wir nachfolgend exemplarisch zwei einfache Testfälle:

```
@Test
void testCreateTimeStampString_Monthly()
{
    final boolean MONTHLY = true;
    assertEquals("2000-2", createTimeStampString(
                           LocalDateTime.of(2000, 2, 7, 0, 0), MONTHLY));
    assertEquals("2000-7", createTimeStampString(
                           LocalDateTime.of(2000, 7, 14, 0, 0), MONTHLY));
}

@Test
void testCreateTimeStampString_Quarterly()
{
    final boolean QUARTERLY = false;
    assertEquals("2000-Q1", createTimeStampString(
                            LocalDateTime.of(2000, 2, 7, 0, 0), QUARTERLY));
    assertEquals("2000-Q3", createTimeStampString(
                            LocalDateTime.of(2000, 7, 14, 0, 0), QUARTERLY));
}
```

Listing 16.4 *Ausführbar als* **'TIMESTAMPUTILSTEST'**

Wir führen die Tests als Programm TIMESTAMPUTILSTEST aus und sie zeigen – wie erwartet – Grün. Normalerweise würden wir noch ein paar mehr Testfälle ergänzen. In diesem Kontext sollen uns aber diese zwei reichen, um mögliche Probleme aufzuzeigen. Allerdings sollten Klassen in der Praxis umfangreicher getestet werden, als es der Platz hier erlaubt.

Unzulänglichkeit: Boolescher Parameter

Durch die Refactoring-Schritte haben wir zwar die Abhängigkeiten zum Package `external` gelöst, jedoch – wie schon zuvor erwähnt – auch eine Unschönheit in unsere öffentliche Schnittstelle eingefügt: einen booleschen Parameter. Was ist daran störend? Aufrufer müssen dadurch immer genau wissen, was die Werte `true` bzw. `false` ausdrücken sollen. Das lässt sich nur durch Betrachten der Implementierung der Methode ermitteln – leider nicht nur anhand der Aufrufstelle.

[8] Insbesondere gilt dies für das Refactoring CHANGE METHOD SIGNATURE, um Parameter umzuordnen, deren Typ zu ändern oder neue Parameter einzufügen, wodurch sich schnell Verhalten ändert. Ebenso kann ein Inlining problematisch sein, weil dadurch die Abarbeitungsreihenfolge leicht geändert wird. Man spricht bei dem Problem auch von Temporal Coupling (vgl. Abschnitt 16.6).

Die Verwendung der booleschen Konstanten `MONTHLY` und `QUARTERLY` in den Tests macht deutlich, dass sich ein weiteres Refactoring anbietet, um den booleschen Parameter in der öffentlichen Schnittstelle zu eliminieren. In unserem Beispiel hatten wir die Klassen im Zugriff und durften dort auch ändern. Das ist jedoch nicht immer der Fall, sodass man mitunter mit der Signatur leben muss. Wie kann man trotzdem für besser verständlichen Sourcecode sorgen? Schauen wir uns verschiedene Abhilfen an.

Abhilfen ohne Änderungen der Signatur Wie wir es beim Erstellen der Tests kennengelernt haben, kann man zwei Konstanten mit sprechenden Namen definieren:

```
public static final boolean MONTHLY = true;
public static final boolean QUARTERLY = false;
```

Oftmals besser lesbar ist es, eine Hilfsmethode wie folgt zu implementieren:

```
private static boolean isMonthly(final ComplexFrequency frequency)
{
    return frequency == ComplexFrequency.P1M;
}
```

Bei der zweiten Variante verbleibt allerdings die Abhängigkeit von Aufrufern an das Package `external`, was aber möglicherweise akzeptabel ist.

Es gibt eine weitere Möglichkeit, die so elegant in der Nutzung und offensichtlich ist, dass man sie leicht übersieht: Man definiert zwei Methoden mit sprechendem Namen, die jeweils den erwarteten Wert zurückgeben:[9]

```
private static boolean monthly()
{
    return true;
}

private static boolean quarterly()
{
    return false;
}
```

Die gezeigten Varianten lösen das eigentliche Problem nicht, lindern jedoch ein wenig die »API-Schmerzen«. Das ist für die Fälle praktisch, in denen man die Schnittstelle der Klasse nicht ändern kann, die Aufrufe aber klarer gestalten möchte. Der Aufruf für monatlich würde wie folgt aussehen:

```
final String timeStamp = createTimeStampString(currentPeriod.getDateTime(),
                                                monthly());
```

[9] Das kann man ganz hervorragend auch für andere Rückgabetypen außer `boolean` machen, solange die Wertemenge nicht zu groß wird.

Abhilfen mit Änderungen der Signatur Wenn man auf die Klassen Zugriff hat und die Methode ändern kann, bietet sich die Definition eines `enum`s an:

```
public enum SupportedFrequencies
{
    MONTHLY, QUARTERLY;
}
```

Damit können wir die Signatur der Methode wie folgt abändern und eine Hilfsvariable `isMonthly` einführen:

```
static String createTimeStampString(final LocalDateTime start,
                                     final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Diese Lösung ist für Aufrufer klarer, was ein sehr wichtiger Punkt ist, da damit auch die Benutzbarkeit verbessert wird. Allerdings fängt das Ganze intern an, unübersichtlich zu werden. Es wird höchste Zeit, nach ein paar Vereinfachungen Ausschau zu halten.

16.3.3 Vereinfachungen

In diesem Abschnitt sehen wir uns zwei Arten von Vereinfachungen an. Zunächst entzerren wir die Anweisungen und die etwas komplexere Logik. Daraus ergeben sich weitere Möglichkeiten, die Formel zur Berechnung an sich zu vereinfachen.

Vereinfachung der Anweisungen

Die Komplexität innerhalb der Methode entsteht vor allem dadurch, dass hier die zwei Fälle »Monatlich« und »Quartalsweise« ineinander verwoben behandelt werden:[10]

```
public static String createTimeStampString(final LocalDateTime start,
                                            final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthValue() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Teilen wir das Ganze doch einfach so auf, dass wir abhängig von `isMonthly` zwei Wertebelegungen erhalten. Versuchen wir schrittweise dorthin zu kommen.

Als Vorbereitung führen wir das Refactoring INLINE für die Variable `value` aus, um den Ausdruck zur String- und Wertekonkatenation zusammenzuführen.

[10] Potenziell eine Verletzung des Single Responsibility Principle (SRP) (vgl. Abschnitt 3.5.3).

Den ternären Operator (?-Operator) kann man in eine if-else-Anweisung umwandeln. Dazu nutzen wir das Tastaturkürzel CTRL+1 für QUICK FIX und wählen dort REPLACE CONDITIONAL WITH 'IF-ELSE'. Das machen wir für beide ?-Operatoren. Damit ergibt sich folgende Variante der ursprünglichen Methode, wobei die entstehenden if-else-Anweisungen allerdings (noch) keine Blöcke sind:

```
public static String createTimeStampString(final LocalDateTime start,
                                            final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor;
    if (isMonthly)
        divisor = 1;
    else
        divisor = 3;
    final String addition;
    if (isMonthly)
        addition = "";
    else
        addition = "Q";

    return start.getYear() + "-" + addition + ((start.getMonthValue() - 1)
         / divisor + 1);
}
```

Das Ergebnis sieht ein wenig chaotisch aus und scheint in die falsche Richtung zu gehen, da viel mehr Zeilen entstanden sind. Lassen Sie sich nicht entmutigen. Die Tests zeigen, dass sich das Verhalten der Methode nicht geändert hat. Wir sind wohl auf dem richtigen Weg. Insbesondere sind die einzelnen Abfragen viel weniger komplex.

Jetzt wollen wir die jeweiligen Zeilen für die beiden Bedingungen zusammen gruppieren. Voraussetzung dazu ist aber, dass wir die if-else-Anweisungen in Blöcke umwandeln. Dazu selektieren wir das if und nutzen wiederum das Tastaturkürzel CTRL+1, was uns nun die Option CHANGE 'IF-ELSE' STATEMENTS TO BLOCKS anbietet, die wir wählen. Danach ordnen wir die Zeilen um, indem wir die Zeilen aus den jeweiligen Bedingungen gruppieren. Als Folge entsteht im unteren Teil ein leeres if-else-Gebilde, das wir entfernen. Es ergibt sich folgende Methode:

```
public static String createTimeStampString(final LocalDateTime start,
                                            final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor;
    final String addition;
    if (isMonthly)
    {
        divisor = 1;
        addition = "";
    }
    else
    {
        divisor = 3;
        addition = "Q";
    }
    return start.getYear() + "-" + addition + ((start.getMonthValue() - 1)
               / divisor + 1);
}
```

Der Sourcecode ist erneut länger geworden, aber zumindest sind die logischen Einheiten gruppiert. Bevor wir vereinfachen können, wird es noch etwas unübersichtlicher und wir benötigen auch etwas Handarbeit, um die inverse Variante des Basis-Refactorings CONSOLIDATE DUPLICATE CONDITIONAL FRAGMENT[11] durchzuführen. Normalerweise will man damit duplizierte Elemente in `if`-Zweigen zu einer Anweisung am Ende zusammenfügen. Hier machen wir das Gegenteil und duplizieren die `return`-Anweisung, um sie dann in jedem `if-else`-Zweig bereitzustellen:

```
public static String createTimeStampString(final LocalDateTime start,
                                            final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor;
    final String addition;
    if (isMonthly)
    {
        divisor = 1;
        addition = "";
        return start.getYear() + "-" + addition + ((start.getMonthValue() - 1)
               / divisor + 1);
    }
    else
    {
        divisor = 3;
        addition = "Q";
        return start.getYear() + "-" + addition + ((start.getMonthValue() - 1)
               / divisor + 1);
    }
}
```

Die Variablen `divisor` und `addition` sind eigentlich überflüssig, da sie jeweils nur einfache Konstanten enthalten. Wir können nun die Werte für beide Variablen direkt im Sourcecode ersetzen. Je nach verwendeter IDE müssen wir etwas Handarbeit leisten, um die in den Zweigen jeweils konstanten Werte in die `return`-Anweisungszeile zu integrieren sowie die überflüssigen Variablendeklarationen zu entfernen:

```
public static String createTimeStampString(final LocalDateTime start,
                                            final SupportedFrequencies frequency)
{
    if (frequency == SupportedFrequencies.MONTHLY)
    {
        return start.getYear() + "-" + ((start.getMonthValue() - 1) / 1 + 1);
    }

    return start.getYear() + "-Q" + ((start.getMonthValue() - 1) / 3 + 1);
}
```

[11] Details finden Sie in Martin Fowlers Buch »Refactoring: Improving the Design of Existing Code« [20].

Hinweis: Viele Zwischenschritte

Diese vielen kleinen Schritte für diesen einfachen Sourcecode-Abschnitt sehen möglicherweise übertrieben aus. Allerdings bietet das den Vorteil, dass man sich in jedem Einzelschritt auf das Wesentliche konzentrieren kann. Wenn die Programmabschnitte komplexer werden, dann profitiert man von kleinen Schritten, die im Falle eines Irrwegs leicht zurückgenommen werden können. Flüchtigkeitsfehler lassen sich dadurch eher vermeiden als bei »Freihand«-Refactorings.

Vereinfachung der Berechnung in mehreren Schritten

Wenn man sich die Ausdrücke anschaut, sollte zumindest im ersten Fall eine Vereinfachung möglich sein. Damit wir nicht abgelenkt werden, schauen wir hier wirklich nur auf den Ausdruck der Monatsberechnung an sich, wobei die äußere Klammerung wegen der Stringkonkatenation nicht weiter betrachtet wird:

```
(start.getMonthValue() - 1) / 1 + 1
```

Die Division / 1 ist nutzlos Eine Division durch 1 ändert das Ergebnis nicht, macht aber den Ausdruck komplizierter und den Sourcecode schlechter lesbar. Also entfernen wir diese Division und erhalten folgende Vereinfachung:

```
(start.getMonthValue() - 1) + 1
```

Weitere Schritte 1 Aufgrund der Vereinfachung ergeben sich neue Möglichkeiten. In der Praxis sieht man es immer mal wieder, dass Ausdrücke eher zu viel geklammert sind. Hier ist die äußere Klammerung um die Subtraktion überflüssig und wird entfernt:

```
start.getMonthValue() - 1 + 1
```

Weitere Schritte 2 In einem letzten Schritt kann die Berechnung `- 1 + 1` entfallen, weil sie den Wert `0` ergibt und hier somit nutzlos ist. Damit verbleibt für die Berechnung der Monate nur noch der Aufruf `start.getMonthValue()` und wir können die Methode wie folgt vereinfachen:

```
public static String createTimeStampString(final LocalDateTime start,
                                            final SupportedFrequencies frequency)
{
    if (frequency == SupportedFrequencies.MONTHLY)
    {
        return start.getYear() + "-" + start.getMonthValue();
    }

    return start.getYear() + "-Q" + ((start.getMonthValue() - 1) / 3 + 1);
}
```

Komplexität der Berechnung für Quartale Die Berechnung des Quartals ist komplexer, was sich kaum vermeiden lässt. In der hier genutzten Klasse `LocalDateTime` beginnt die Zählung der Tage und Monate jeweils bei 1, wie es der menschlichen Denkweise entspricht. Die gezeigten Berechnungen für das Quartal bilden die Werte von 1 – 12 durch die Subtraktion von 1 auf 0 – 11 ab, wodurch die Division durch 3 einen Wertebereich von 0 – 3 liefert. Die Addition von 1 ergibt dann den Wertebereich 1 – 4. Allerdings liest sich das `/ 3 + 1` potenziell falsch. Um die Berechnung klarer zu gestalten, kann man die Addition von `1` nach vorne ziehen:

```
return start.getYear() + "-Q" + (1 + (start.getMonthValue() - 1) / 3);
```

Fazit

Wenn man bedenkt, mit welch kompliziertem Ausdruck wir ins Rennen gestartet sind, ist das Erreichte beeindruckend. Bei einem Blick auf die ursprüngliche Berechnung wäre weder eine Aussage möglich gewesen, was das Resultat denn nun genau ist, noch, ob man die Berechnung vereinfachen kann. Durch unseren letzten Schritt ist keine Vereinfachung für die Monatsberechnung mehr nötig und das Ergebnis offensichtlich. Nur die Quartalsberechnung ist eben etwas komplizierter.

16.3.4 Verlagern von Funktionalität

Betrachten wir die letzte Version der Methode `createTimeStampString(LocalDateTime, SupportedFrequencies)` einmal genauer, dann erkennen wir, dass sie im `if-else` jeweils Funktionalität enthält, die stark mit der Aufzählung `SupportedFrequencies` verbunden ist. Diese haben wir beim Erstellen der Unit Tests zur Verbesserung des leicht unhandlichen APIs definiert. Es bietet sich nun an, noch mehr Funktionalität dorthin zu verlagern. Dabei nutzen wir, dass eine `enum`-Aufzählung Attribute und Methoden besitzen kann und Letztere sogar überschrieben werden können:

```
public enum SupportedFrequencies
{
   MONTHLY
   {
      public String createTimeStampString(final LocalDateTime start)
      {
         return start.getYear() + "-" + start.getMonthValue();
      }
   },

   QUARTERLY
   {
      public String createTimeStampString(final LocalDateTime start)
      {
         return start.getYear() + "-Q" + (1 + (start.getMonthValue() - 1) / 3);
      }
   };

   public abstract String createTimeStampString(final LocalDateTime start);
}
```

Die Utility-Klasse enthält nur noch folgende Methode:

```
public static String createTimeStampString(final LocalDateTime start,
                                            final SupportedFrequencies frequency)
{
    return frequency.createTimeStampString(start);
}
```

Tatsächlich sprechen nur noch Rückwärtskompatibilitätsgründe dafür, die Utility-Klasse überhaupt noch beizubehalten. Ansonsten könnte man die Funktionalität durch direkte Aufrufe an die jeweilige `enum`-Konstante realisieren.

16.4 Der Refactoring-Katalog

Dieser Abschnitt stellt einige Refactorings sowie Transformationen als Schritt-für-Schritt-Anleitungen vor. Diese sollen dabei helfen, ein spezielles Problem auf eine definierte Art und Weise zu beheben. Einige Refactorings lassen sich zu »High-Level-Refactorings« kombinieren. Ein Beispiel dafür ist die als MINIMIERE ZUSTANDSÄNDERUNGEN beschriebene Kombination aus Abschnitt 16.4.5.

Im folgenden Text spreche ich häufig der Einfachheit halber von `get()`- und `set()`-Methoden. Diese müssen nicht immer mit einem solchen Präfix anfangen, sondern es sind ganz allgemein Accessor- und Mutator-Methoden gemeint.

16.4.1 Reduziere die Sichtbarkeit von Attributen

Bekanntlich stellen Attribute den internen Zustand eines Objekts dar. Dieser sollte von außen nicht direkt sichtbar oder sogar änderbar sein. Dem OO-Grundgedanken der Datenkapselung (vgl. Abschnitt 3.1) folgend, ist es das Ziel bei diesem Refactoring, eine direkte Änderbarkeit von Attributen auf ein Minimum zu reduzieren. Im Idealfall können Details der Implementierung ohne Rückwirkungen auf Nutzer modifiziert werden. Beispielsweise können Daten entweder als Attribut gehalten, bei Bedarf berechnet oder aus einer externen Quelle gelesen werden. Als Voraussetzung sollte die Sichtbarkeit von Attributen möglichst weit eingeschränkt werden.

Schauen wir dazu auf die in Abbildung 16-2 als Klassendiagramm visualisierte Klasse `Person`, die die zwei Attribute `name` und `age` besitzt und die Ausgangsbasis für dieses Refactoring darstellt. Die Attribute sind zu Demonstrationszwecken `public` ('+') und `protected` ('#').

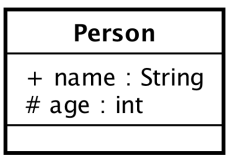

Abbildung 16-2 *Ausgangszustand*