Kapitel 3

Codebasierte Verbesserung eines Tests

Auch wenn Ihre Anwendung Legacy Code ist, können und sollten Ihre Tests von Anfang an Clean Code sein. Wie Sie das sicherstellen, erfahren Sie in diesem Kapitel.

Im ersten Abschnitt dieses Kapitels stelle ich eine Testmethode vor, mit der ein Aspekt der Funktionalität »Hervorheben von Änderungen« unserer Stammdatenmanagement-Anwendung automatisch überprüft werden kann. Dieser Test basiert zwar auf manuell angelegten Daten und ist damit nicht ohne Weiteres in ein Nachfolgesystem übertragbar, funktional ist er aber richtig und würde auch bei einer erweiterten Programmprüfung nicht negativ auffallen. Wenn wir annehmen, dass die Datenbasis mit organisatorischen Mitteln stabil gehalten werden kann, scheint es fast, als ob es keinen Bedarf für eine Verbesserung dieses Tests gibt. Mit diesem und den folgenden Kapiteln möchte ich Ihnen jedoch demonstrieren, dass solche Verbesserungsmaßnahmen sogar sehr sinnvoll sind. Unter anderem können sie die Effizienz eines Entwicklungsteams signifikant steigern.

Die dargestellte Ausgangssituation ist typisch für meine Arbeit als Agile Software Engineering (ASE) Coach. Ich möchte immer, dass das von mir unterstützte Team schon mindestens einen repräsentativen Test geschrieben hat. Damit hat sich das Team bereits im Testschreiben versucht, und wir haben etwas, worauf wir aufsetzen können. Bei den meisten Verbesserungen handelt es sich um Entwurfsmuster, die sich auf nahezu jede Anwendung in ähnlicher Weise anwenden lassen. Wenn Sie möchten, stellen Sie sich doch vor, Sie seien Mitglied eines Teams, das von mir gecoacht wird. Sie haben die im vorangegangenen Kapitel vorgestellte Stammdatenanwendung übernommen und haben gerade Ihren ersten Test entwickelt.

3.1 Einführung in den Testcode

Die folgende Testklasse enthält einen ersten, selbstüberprüfenden Integrationstest, der auf manuell angelegte Testdaten aufbaut. Konkret handelt es sich bei den Testdaten um einen Änderungsantrag mit der ID 3, mit dem die Flugverbindung `AIR 0001`

geändert und diese Änderung gesichert wurde. Welcher Art die Änderung war, spielt dabei keine Rolle. Die Testmethode überprüft nur, ob die zu testende Methode das Kennzeichen richtig setzt, dass die gegebene Flugverbindungsentität mindestens eine gesicherte Änderung enthält.

3.1.1 Erste Definition einer Testklasse

Die Definition dieser *Testklasse* befindet sich in der Regel im *lokalen Testinclude* der Produktklasse, deren Methoden von der Testklasse getestet werden. Im Class Builder (Transaktion SE24) rufen Sie das Testinclude einer Klasse über den Menüpfad **Springen • Lokale Definitionen/Implementierungen • Lokale Testklassen** auf. Alternativ können Sie die Tastenkombination Strg + ⇧ + F11 verwenden. Eine neue Klasse hat noch kein Testinclude. Das System fragt Sie deshalb, ob Sie ein Testinclude anlegen wollen (allerdings nur, wenn Sie sich im Änderungsmodus befinden).

Die erste Zeile von Listing 3.1 definiert die lokale Testklasse LTC_HIGHLIGHT_CHANGES_0 mit dem Zusatz FINAL. Dies ist zwar technisch nicht notwendig, die erweiterte Programmprüfung fordert diese *Finalität* allerdings für jede Klasse ohne Unterklassen ein, damit es nicht zu einer von Ihnen nicht beabsichtigten Vererbungsbeziehung kommen kann.

```
CLASS ltc_highlight_changes_0 DEFINITION FINAL
FOR TESTING DURATION SHORT RISK LEVEL HARMLESS.
  PRIVATE SECTION.
    METHODS get_entity_field_changes FOR TESTING.
    DATA mo_conv_api TYPE REF TO if_usmd_conv_som_gov_api.
ENDCLASS.
```

Listing 3.1 Definition einer Testklasse für die Convenience API

Die Zusätze in der Klassendefinition haben die folgenden Bedeutungen:

- FOR TESTING: Die Klasse dient dem Testen und darf nicht produktiv verwendet werden. Es kann sich um eine Testklasse, eine Doubleklasse oder eine sonstige Hilfsklasse handeln.
- DURATION SHORT: Es handelt sich um eine Testklasse, deren Ausführung nicht länger dauern darf, als es der im Customizing hinterlegte Höchstwert für eine kurze Laufzeit vorgibt (siehe auch Kapitel 26, »ABAP Unit«).
- RISK LEVEL HARMLESS: Die Ausführung der Testklasse hat, etwa in Bezug auf Datenbankänderungen, keine oder nur unbedeutende Auswirkungen. Auch dazu gibt es einen im Customizing hinterlegten Höchstwert.

Für die Ausführung einer Testklasse ist innerhalb des *ABAP-Unit-Testframeworks* der sogenannte *ABAP-Unit-Testrunner* zuständig. Da er im Hintergrund eine freundschaftliche Beziehung zu jeder Testklasse unterhält, können und sollen Sie deren Komponenten so weit wie möglich in der privaten Sektion definieren. Im vorliegenden Fall betrifft das zum einen die Methode GET_ENTITY_FIELD_CHANGES, die über den Zusatz FOR TESTING als *Testmethode* definiert ist. Zum anderen betrifft es das Attribut MO_CONV_API für das zu testende Objekt der Convenience API.

Bezeichnung des Attributs für das zu testende Objekt

Viele ABAP-Unit-Entwickler bevorzugen CUT (als Abkürzung für *Class Under Test* oder allgemeiner Code Under Test) als Attributnamen, weil damit in jeder Testklasse das zu testende Objekt sofort ins Auge fällt. In einer Entwicklungsumgebung hat diese generische Bezeichnung keine praktischen Nachteile, weil die Definition und damit die Bedeutung von CUT nur einen Tooltip oder einen Navigationsschritt entfernt sind. Da Ihnen diese Hilfsmittel als Leser nicht zur Verfügung stehen, habe ich mich für dieses Buch klar für die semantische Namensgebung (hier MO_CONV_API) entschieden, die ich persönlich auch wegen ihrer besseren Lesbarkeit bevorzuge.

3.1.2 Erste Implementierung der Testklasse

Die Implementierung einer Testklasse befindet sich in der Regel unterhalb ihrer Definition im selben lokalen Testinclude. Listing 3.2 zeigt die Implementierung der Testklasse LTC_HIGHLIGHT_CHANGES_0. Die Testmethode GET_ENTITY_FIELD_CHANGES ist stark vom prozeduralen Programmierstil geprägt. In den folgenden Kapiteln werde ich Ihnen unter anderem demonstrieren, wie Sie die Methode objektorientiert umsetzen können.

```
CLASS ltc_highlight_changes_0 IMPLEMENTATION.
  METHOD get_entity_field_changes.
    DATA:
      lt_entity_type_keys    TYPE usmd_gov_api_ts_ent_tabl,
      ls_entity_type_keys    TYPE usmd_gov_api_s_ent_tabl,
      lt_entity_type_changes TYPE usmd_t_changed_entities,
      lsr_pfli_key           TYPE REF TO data,
      lv_entity_found        TYPE abap_bool.
    FIELD-SYMBOLS:
      <s_key>                TYPE any,
      <s_pfli_key>           TYPE any,
      <t_pfli_key>           TYPE INDEX TABLE,
      <carr_id>              TYPE mdg_s_carr_id,
      <conn_id>              TYPE mdg_s_conn_id,
      <s_entity_type_changes> TYPE usmd_s_changed_entities,
      <s_entity_changes>     TYPE usmd_s_changed_entity.
```

```
*   Erzeugen und Konfigurieren der zu testenden API
    mo_conv_api = cl_usmd_conv_som_gov_api=>get_instance( 'SF' ).
    mo_conv_api->set_environment( iv_crequest_id = '3' ).

*   Erzeugen und Spezifizieren eines Entitätsschlüssels
    mo_conv_api->get_entity_structure(
      EXPORTING
        iv_entity_name = 'PFLI'
      IMPORTING
        er_structure   = lsr_pfli_key
    ).
    ASSIGN lsr_pfli_key->* TO <s_pfli_key>.
    ASSIGN COMPONENT 'CARR' OF STRUCTURE <s_pfli_key> TO <carr_id>.
    <carr_id> = 'AIR'.
    ASSIGN COMPONENT 'PFLI' OF STRUCTURE <s_pfli_key> TO <conn_id>.
    <conn_id> = '0001'.

*   Einfügen des Entitätsschlüssels in eine neue Schlüsseltabelle
    ls_entity_type_keys-entity = 'PFLI'.
    mo_conv_api->get_entity_structure(
      EXPORTING
        iv_entity_name = 'PFLI'
      IMPORTING
        er_table       = ls_entity_type_keys-tabl
    ).
    ASSIGN ls_entity_type_keys-tabl->* TO <t_pfli_key>.
    INSERT <s_pfli_key> INTO TABLE <t_pfli_key>.
    INSERT ls_entity_type_keys INTO TABLE lt_entity_type_keys.

*   Berechnen der Änderungen für Entitäten in der Schlüsseltabelle
    lt_entity_type_changes = mo_conv_api->get_entity_field_changes(
        iv_struct            = zif_usmd=>c_struct_key_attr
        it_entity_keys       = lt_entity_type_keys
        iv_saved_changes     = abap_true
        iv_unsaved_changes   = abap_false
        iv_contained_changes = abap_false
    ).

*   Suchen der Änderungstabelle für den betrachteten Entitätstyp
    READ TABLE lt_entity_type_changes
      ASSIGNING <s_entity_type_changes>
      WITH KEY entity_type = 'PFLI'
               struct      = zif_usmd=>c_struct_key_attr.
```



```
    cl_abap_unit_assert=>assert_subrc( exp = 0 ).

*   Suchen der Änderungen für die betrachtete Entität
    LOOP AT <s_entity_type_changes>-changed_entities
    ASSIGNING <s_entity_changes>.
      ASSIGN <s_entity_changes>-entity->* TO <s_key>.
      IF <s_key> = <s_pfli_key>.
        lv_entity_found = abap_true.
        EXIT.
      ENDIF.
    ENDLOOP.
    cl_abap_unit_assert=>assert_true( lv_entity_found ).

*   Überprüfen der Änderungen dieser Entität
    cl_abap_unit_assert=>assert_true(
        <s_entity_changes>-saved_change
    ).
  ENDMETHOD.
ENDCLASS.
```

Listing 3.2 Implementierung der Testklasse für die Convenience API

Im Folgenden möchte ich Sie mit der Bedeutung der einzelnen Anweisungen vertraut machen. Damit werde ich auch meine Einführung in die Stammdatenmanagement-Anwendung und ihre Funktion »Hervorheben von Änderungen« abschließen. Meiner Erfahrung nach genügt Anwendungswissen dieses Umfangs, um weitreichende Verbesserungen an existierenden Testmethoden durchführen zu können. Für das Schreiben weiterer Testmethoden ist in der Regel zwar zusätzliches Wissen notwendig, aber eben nur wenig, und dieses kann bedarfsorientiert zügig erworben werden.

Für eine bessere Lesbarkeit habe ich in Listing 3.2 alle Anweisungen, die zusammen einen logischen Schritt darstellen, direkt untereinandergeschrieben und mit einem einleitenden Kommentar versehen. Sie sind von anderen logischen Schritten jeweils durch eine Leerzeile getrennt.

Erzeugen und Konfigurieren der zu testenden API

Listing 3.3 zeigt noch einmal den ersten logischen Schritt. Er umfasst die Bereitstellung eines gültigen Convenience-API-Objekts im Attribut `MO_CONV_API` der Testklasse.

```
mo_conv_api = cl_usmd_conv_som_gov_api=>get_instance( 'SF' ).
mo_conv_api->set_environment( iv_crequest_id = '3' ).
```

Listing 3.3 Erzeugung und Konfiguration eines Convenience-API-Objekts

Neben der Erzeugung des Convenience-API-Objekts gehört die Ausrichtung auf den manuell angelegten Änderungsantrag mit der ID 3 zu diesem Schritt. Das Ergebnis entspricht der Situation im Backend, nachdem ein Benutzer die Einzelobjektpflege (SOM-Benutzeroberfläche) für diesen Änderungsantrag aufgerufen hat.

Erzeugen und Spezifizieren eines Entitätsschlüssels

Mit der Anweisung in Listing 3.4 erzeugt die Testmethode eine initiale Schlüsselstruktur für eine Flugverbindung.

```
mo_conv_api->get_entity_structure(
  EXPORTING
    iv_entity_name = 'PFLI'
  IMPORTING
    er_structure   = lsr_pfli_key
).
```

Listing 3.4 Erzeugung eines initialen Schlüssels für eine Flugverbindung

Genauer gesagt beschafft sich die Testmethode den Speicher für diese Schlüsselstruktur dynamisch mithilfe der Convenience API. Alternativ wäre eine statische Erzeugung der Struktur auf Basis einer lokalen Definition möglich. Das würde die Testmethode sogar unabhängiger vom Customizing für das Stammdatenmodell machen. Insgesamt bedeuten diese lokalen Definitionen aber viel Aufwand, und diesen Aufwand sollten wir aus Gründen der Effizienz erst dann erbringen, wenn er wirklich notwendig ist.

Eine solche Notwendigkeit wäre gegeben, wenn das Testdatenmodell mit organisatorischen Mitteln nicht stabil gehalten werden kann, wenn also Tests immer wieder fehlschlagen, weil jemand unbeabsichtigt das Customizing geändert hat. Dieses Problem hat sich allerdings in der Praxis über Jahre hinweg nicht eingestellt. Des Weiteren wird dieser Code schon in Kürze Teil einer Testinfrastruktur sein, und deren Hilfsmethoden sollen natürlich, wie die Anwendung selbst, verschiedenste Stammdatenmodelle unterstützen.

In Listing 3.5 wird die initiale Schlüsselstruktur mit den Schlüsselwerten derjenigen Flugverbindung belegt, die mit dem in Listing 3.3 spezifizierten Änderungsantrag manuell bearbeitet wurde.

```
ASSIGN lsr_pfli_key->* TO <s_pfli_key>.
ASSIGN COMPONENT 'CARR' OF STRUCTURE <s_pfli_key> TO <carr_id>.
<carr_id> = 'AIR'.
ASSIGN COMPONENT 'PFLI' OF STRUCTURE <s_pfli_key> TO <conn_id>.
<conn_id> = '0001'.
```

Listing 3.5 Spezifikation der Schlüsselfelder einer Flugverbindung

Bei LSR_PFLI_KEY handelt es sich, wie an dem Präfix erkennbar, um eine lokale (L) Referenz (R) auf eine anonyme Struktur (S) im Speicher. Mit dem Dereferenzierungsoperator ->* erhalten Sie dynamischen Zugriff auf diese Struktur. Wenn Sie dieses Zwischenergebnis mit der Anweisung ASSIGN dem Feldsymbol <S_PFLI_KEY> zuweisen, können sie diese Struktur im weiteren Verlauf über den Namen <S_PFLI_KEY> direkt ansprechen. Da das Feldsymbol <S_PFLI_KEY> vom Typ ANY ist, erfolgt der Zugriff auf die beiden Schlüsselkomponenten CARR und PFLI dynamisch über die Feldsymbole <CARR_ID> und <CONN_ID>.

Einfügen des Entitätsschlüssels in eine neue Schlüsseltabelle

Listing 3.6 fügt diesen Flugverbindungsschlüssel in die geschachtelte Schlüsseltabelle LT_ENTITY_TYPE_KEYS ein, die dann von der zu testenden Methode GET_ENTITY_FIELD_CHANGES importiert wird.

```
ls_entity_type_keys-entity = 'PFLI'.
mo_conv_api->get_entity_structure(
  EXPORTING
    iv_entity_name = 'PFLI'
  IMPORTING
    er_table       = ls_entity_type_keys-tabl
).
ASSIGN ls_entity_type_keys-tabl->* TO <t_pfli_key>.
INSERT <s_pfli_key> INTO TABLE <t_pfli_key>.
INSERT ls_entity_type_keys INTO TABLE lt_entity_type_keys.
```

Listing 3.6 Erzeugung und Befüllung einer Schlüsseltabelle für Flugverbindungen

Jeder Eintrag in der Tabelle LT_ENTITY_TYPE_KEYS enthält alle Schlüssel zu einem bestimmten Entitätstyp. Diese Schlüssel liegen jeweils in einer eigenen Schlüsseltabelle vor. Diese Tabellen werden ebenfalls mit der Methode GET_ENTITY_STRUCTURE dynamisch erzeugt.

Leider passt der Name der Methode GET_ENTITY_STRUCTURE nicht zur Erzeugung einer Tabelle. Die Zuständigkeit dieser Methode wurde offensichtlich erweitert, ohne ihren Namen anzupassen, und nun gibt ausschließlich der Ausgabeparameter ER_TABLE Aufschluss über die Verwendung zur Tabellenerzeugung. Ebenso bezeichnet der Komponentenname ENTITY der Zeilenstruktur eigentlich einen Entitätstyp. Mängel dieser Art sind bei Bestandscode keine Seltenheit. Da mit diesen missverständlichen Bezeichnungen eine gute Lesbarkeit nicht erreichbar ist, sollten diese Anweisungen so nicht in der Testmethode verbleiben.

Mithilfe des Feldsymbols <T_PFLI_KEY> kann der weiterhin über das Feldsymbol <S_PFLI_KEY> zugängliche Flugverbindungsschlüssel direkt eingefügt werden. Zuletzt

wird die vollständige Struktur LS_ENTITY_TYPE_KEYS für den Entitätstyp PFLI noch in die geschachtelte Tabelle LT_ENTITY_TYPE_KEYS eingefügt.

Berechnen der Änderungen für Entitäten in der Schlüsseltabelle

Mit Listing 3.7 ruft die Testmethode die zu testende Methode unter anderem mit der geschachtelten Schlüsseltabelle auf.

```
lt_entity_type_changes = mo_conv_api->get_entity_field_changes(
    iv_struct            = zif_usmd=>c_struct_key_attr
    it_entity_keys       = lt_entity_type_keys
    iv_saved_changes     = abap_true
    iv_unsaved_changes   = abap_false
    iv_contained_changes = abap_false
).
```

Listing 3.7 Aufruf der zu testenden Methode

Mithilfe der booleschen Argumente macht die Testmethode klar, dass sie ausschließlich an gesicherten Änderungen derjenigen Entitäten interessiert ist, für die sie auch einen Schlüssel übergeben hat. Wie schon erwähnt, ist der Strukturtyp, übergeben in dem Parameter IV_STRUCT, für das weitere Verständnis nicht von Bedeutung.

[»]

Lokale Objekte in den Codebeispielen

Um die Codebeispiele für dieses Buch zu konservieren, habe ich für die meisten Codebeispiele lokale Kopien der ausgelieferten Klassen angelegt und geringfügig zugunsten einer besseren Lesbarkeit geändert. Für das Interface ZIF_USMD etwa gibt es keine direkte Entsprechung, aber das reale Interface für die Konstante C_STRUCT_KEY_ATTR hätte mit seinem langen Namen einen unleserlichen Zeilenumbruch nötig gemacht.

Suche nach der Änderungstabelle für den betrachteten Entitätstyp

Mit Listing 3.8 sucht die Testmethode in der geschachtelten Tabelle, die alle Änderungen enthält, diejenigen Änderungen, die für den Entitätstyp PFLI gelten.

```
READ TABLE lt_entity_type_changes
  ASSIGNING <s_entity_type_changes>
  WITH KEY entity_type = 'PFLI'
           struct      = zif_usmd=>c_struct_key_attr.
cl_abap_unit_assert=>assert_subrc( exp = 0 ).
```

Listing 3.8 Bestimmung der Änderungen aller Flugverbindungen

Die letzte Anweisung in Listing 3.8 ist eine sogenannte *Assertion*, also ein Methodenaufruf, mit dem die Testmethode einen bestimmten Zustand oder ein bestimmtes Ergebnis einfordert. Die Klasse CL_ABAP_UNIT_ASSERT ist Teil des ABAP-Unit-Testframeworks. Mit ihren sogenannten *Assertion-Methoden* kann die Testmethode nicht nur eine Behauptung formulieren, sondern im Fehlerfall, also wenn diese Behauptung nicht zutrifft, auch eine passende Meldung ins ABAP-Unit-Protokoll schreiben.

Die Assertion in Listing 3.8 ist noch nicht die zentrale Behauptung dieser Testmethode. Vielmehr handelt es sich dabei um eine Zwischenbedingung, ohne deren Erfüllung der Testfall nicht mehr zum Erfolg geführt werden kann. Die Testmethode drückt mit dieser sogenannten *Guard-Assertion* ihre Erwartung aus, dass für mindestens eine gegebene Flugverbindung eine Änderung berechnet wurde. Letzten Endes geht es natürlich um die betrachtete Flugverbindung AIR 0001, aber für diese sind eben höchstens dann Änderungen in der Rückgabetabelle enthalten, wenn es überhaupt irgendwelche Änderungen für Flugverbindungen gibt. Mit Guard-Assertions werden aber auch Situationen verhindert, in denen der Test nicht mehr geordnet fortgeführt werden kann.

Suche nach den Änderungen für die betrachtete Entität

Mit Listing 3.9 versucht die Testmethode, Änderungen der betrachteten Flugverbindungsentität zu finden.

```
LOOP AT <s_entity_type_changes>-changed_entities
ASSIGNING <s_entity_changes>.
  ASSIGN <s_entity_changes>-entity->* TO <s_key>.
  IF <s_key> = <s_pfli_key>.
    lv_entity_found = abap_true.
    EXIT.
  ENDIF.
ENDLOOP.
cl_abap_unit_assert=>assert_true( lv_entity_found ).
```

Listing 3.9 Bestimmung der Änderungen der betrachteten Flugverbindung

Da in einer Tabelle nicht mithilfe eines generischen Schlüssels gesucht werden kann, muss diese Suche in einer Schleife so lange durchgeführt werden, bis ein gleicher Entitätsschlüssel gefunden werden kann. Wieder kommt eine Guard-Assertion zum Einsatz; dieses Mal soll die manuell geänderte Flugverbindungsentität in der Änderungstabelle ihres Entitätstyps gefunden werden.

Überprüfen der Änderungen dieser Entität

Mit Listing 3.10 behauptet die Testmethode, dass für die betrachtete Flugverbindung mit dem gegebenen Änderungsantrag mindestens eine gesicherte Änderung durchgeführt wurde.

```
cl_abap_unit_assert=>assert_true(
    <s_entity_changes>-saved_change
).
```

Listing 3.10 Zentrale Assertion der Testmethode

3.1.3 ABAP-Unit-Protokoll

Die kompakte Schreibweise der Guard-Assertions in Listing 3.8 und Listing 3.9 geht zulasten der Lesbarkeit des Protokolls, falls einmal eine von ihnen nicht erfüllt sein sollte. Eine Alternative könnte wie in Listing 3.11 aussehen.

```
IF sy-subrc = 0.
  LOOP AT <s_entity_type_changes>-changed_entities
  ASSIGNING <s_entity_changes>.
    ASSIGN <s_entity_changes>-entity->* TO <s_key>.
    IF <s_key> = <s_pfli_key>.
      lv_entity_found = abap_true.
      EXIT.
    ENDIF.
  ENDLOOP.
ENDIF.
IF lv_entity_found = abap_false.
  lv_msg_string =
    'Change request 3, flight connection AIR 0001: no changes'.
  cl_abap_unit_assert=>fail( lv_msg_string ).
ENDIF.
```

Listing 3.11 Ausführliche Fehlermeldung für das ABAP-Unit-Protokoll

Falls es Änderungen zum Entitätstyp `PFLI` gibt, wird die Entität mit dem Schlüssel `<S_PFLI_KEY>` gesucht und der Wert der Variablen `LV_ENTITY_FOUND` entsprechend gesetzt. Andernfalls ist der Wert der Variablen `LV_ENTITY_FOUND` in jedem Fall `ABAP_FALSE`.

Die Fehlermeldung gibt die verwendeten Testdaten und den Grund für den vorzeitigen Ausstieg vollständig wieder. Meistens ist ein solcher Aufwand für Fehlermeldungen nicht nötig. Bei kleinen Testmethoden mit nur einer Assertion genügt fast immer der Name der Testmethode für eine erste Problemeingrenzung. Und kurze Testmethoden lassen sich sehr schnell debuggen. Auch für die *Fehlerlokalisierung* gilt die Regel, dass man keine vorzeitigen Optimierungen durchführen sollte (*Avoid-Premature-Optimizations-Prinzip*).

Diese originale Version des Testcodes bezeichne ich im Folgenden als *Version 0*. Ihre Nachfolgerversion wird mittels Refactoring aus der Version 0 hervorgehen: lesbarer und strukturell besser, aber funktional gleich.

3.1.4 Diagramme

Abbildung 3.1 visualisiert diesen Testablauf in einem Blockdiagramm. Links oben sehen Sie einen Tester, der über die Benutzeroberfläche der Einzelobjektpflege (SOM-UI) und die Convenience API die für den Testfall benötigten Entitätsdaten ins Backend schreibt. Der Änderungsantrag, mit dem die aktiven Daten der Flugverbindung zuerst gelesen, dann als ungesicherte Daten im Hauptspeicher abgelegt und schließlich als gesicherte Daten persistiert werden, wird dabei implizit über die Methode `SET_ENVIRONMENT` in das Attribut `MV_CREQUEST_ID` der Convenience API geschrieben (Aufruf von links).

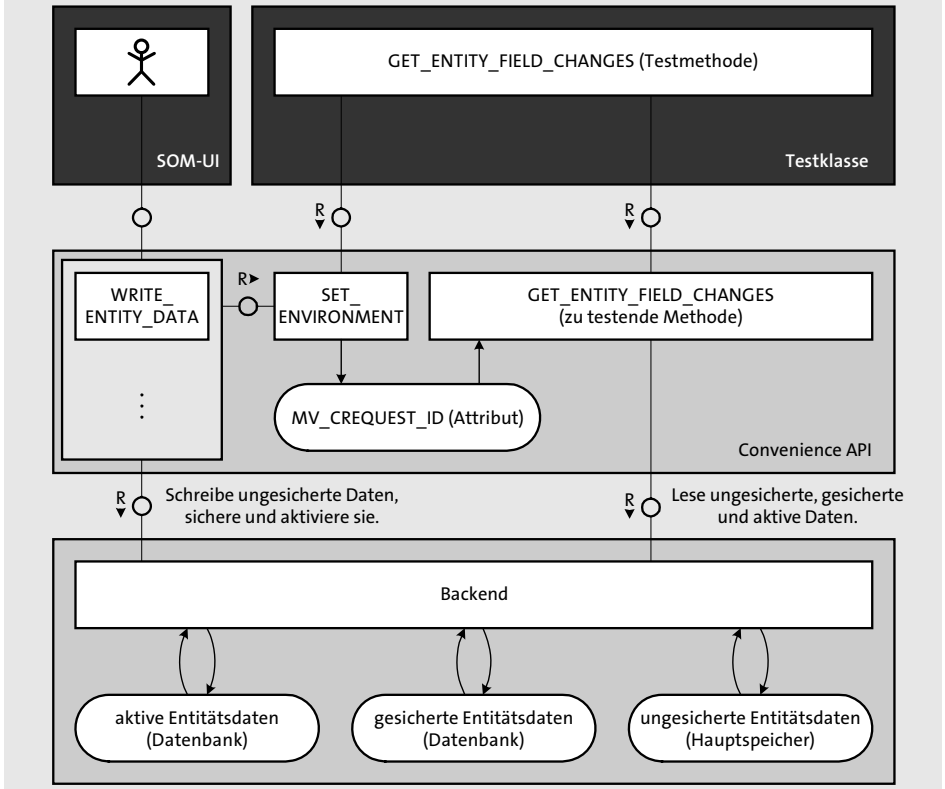

Abbildung 3.1 Blockdiagramm für die originale Testmethode

Wie Abbildung 3.1 und Listing 3.3 zeigen, muss die Testmethode `GET_ENTITY_FIELD_CHANGES` diesen Aufruf der Methode `SET_ENVIRONMENT` explizit für das Objekt `MO_CONV_API` ausführen (Aufruf von oben). Nur so kann die zu testende Methode `GET_ENTITY_FIELD_CHANGES` nach ihrem Aufruf durch die Testmethode den relevanten Änderungsantrag im Attribut `MV_CREQUEST_ID` vorfinden. Für ihre Berechnungen ruft die zu tes-

tende Methode (wiederholt) ins Backend, damit das Backend für sie aktive, gesicherte und ungesicherte Daten der genannten Entitäten liest.

Ein Blockdiagramm zeigt den Verbund und das Zusammenspiel von Objekten. In Abbildung 3.1 sind dies das Testobjekt der Testklasse als Block oben rechts, das Objekt der Convenience API als Block in der Mitte und der als Backend bezeichnete Objektverbund als Block unten. Dagegen zeigt ein Klassendiagramm, wie diese Objekte designt, also softwaretechnisch definiert, sind.

Das Klassendiagramm in Abbildung 3.2 zeigt auf der linken Seite die Testklasse `LTC_HIGHLIGHT_CHANGES_0` mit einem Attribut für die zu testende Convenience API und mit der bislang einzigen Testmethode `GET_ENTITY_FIELD_CHANGES`. Der kursive Zusatz `FOR TESTING` ist als Kommentar zu verstehen, denn dieses ABAP-Sprachelement existiert in der *Unified Modeling Language* (UML), in der das Klassenmodell modelliert ist, nicht. Auf der rechten Seite steht das für diesen Test wesentliche Interface `IF_USMD_CONV_SOM_GOV_API`, das die zu testende Klasse `CL_USMD_CONV_SOM_GOV_API` implementiert.

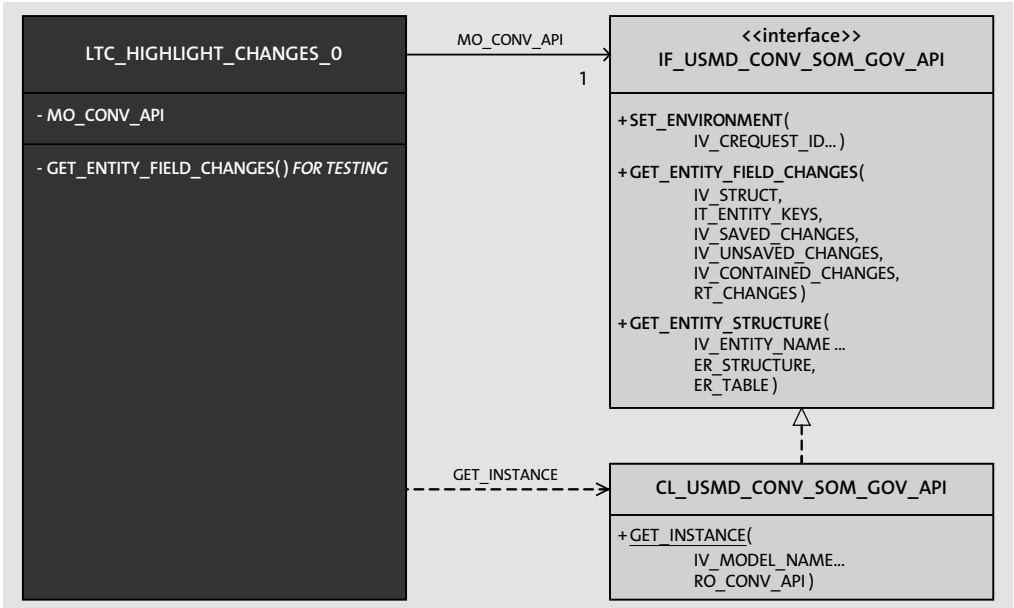

Abbildung 3.2 Klassendiagramm für die originale Testklasse

Obwohl ein Klassendiagramm nicht auf die Darstellung eines Ablaufs ausgelegt ist, lässt sich aus diesem Diagramm trotzdem ablesen, dass sich die Testmethode zuerst von der produktiven Klasse ein Objekt geben lässt und dann auf die Funktionalität des Objekts ausschließlich über dessen Interface zugreift. Dabei kommen zwei unterschiedliche Beziehungen zur Anwendung:

- Zum einen eine *Assoziation* (engl. *Association*), die über das Attribut `MO_CONV_API` aufrechterhalten wird. Sie wird durch einen Pfeil mit durchgezogener Linie symbolisiert.
- Zum anderen eine *Verwendung* (engl. *Usage*), bei der kein Attribut im Spiel ist. Sie wird durch einen Pfeil mit gestrichelter Linie symbolisiert.

In beiden Fällen zeigt die Pfeilspitze die Richtung der Beziehung an.

Testaufbau und Assertions

In diesem Teil des Buches gehe ich am Ende jedes Abschnitts, der eine neue Version des Testcodes eingeführt hat, kurz auf die zugrunde liegende Theorie ein. Dazu gehören neben Designprinzipien und Entwurfsmustern (engl. *Patterns*) auch anerkannte Praktiken der agilen Softwareentwicklung (z. B. Refactoring). Bei den verwendeten Fachbegriffen habe ich mich, soweit dies möglich und dienlich war, an die Standardwerke von Gerard Meszaros (XUnit Test Patterns, 2009) und Robert C. Martin (Clean Code, 2009) gehalten.

Die Testmethode verlässt sich auf einen im Vorfeld erstellten Testaufbau (*Prebuild-Fixture-Entwurfsmuster*). Da auf ihn nur lesend zugegriffen wird, kann er auch für weitere Testmethoden wiederverwendet werden (*Shared-Fixture-Entwurfsmuster*).

Die Testmethode selbst vervollständigt den Testaufbau (*Inline-Setup-Entwurfsmuster*). Es scheint, als ob sie mit einem eigenen Objekt der zu testenden Klasse arbeitet (*Fresh-Fixture-Entwurfsmuster*). Anstatt eine `IF`-Anweisung zu verwenden, vermeidet die Testmethode mithilfe einer Assertion, dass der Testfall ungeordnet weiterläuft (*Guard-Assertion-Entwurfsmuster*). Die Testmethode überprüft ihr Ergebnis mit der Klasse `CL_ABAP_UNIT_ASSERT` selbst (*Self-Validating-Prinzip*).

3.2 Allgemeine Clean-Code-Prinzipien

Wie Robert C. Martin gezeigt hat, kann man über sauberen Code (engl. *Clean Code*) ein ganzes Buch schreiben. Ich konzentriere mich hier auf die Verbesserung der Lesbarkeit und die Vermeidung von Dopplungen des Testcodes. Was sich nach wenig anhört, erweist sich bei konsequenter Verfolgung als hochgesteckter Anspruch. Diese Ziele dominieren nicht nur diesen Abschnitt, sondern auch die meisten anderen Kapitel dieses ersten Teils.

3.2.1 Verbesserungsprozess

Um den Prozess zur Verbesserung des Testcodes zu strukturieren, leite ich jede neue Version wie folgt ein:

1. **Stand der Vorgängerversion**
   Wo stehe ich? Was habe ich schon erreicht?
2. **Probleme der Vorgängerversion**
   Was fällt mir auf? Was sollte ich als Nächstes verbessern?
3. **Lösungswege zu einer Nachfolgerversion**
   Wohin möchte ich? Wie erreiche ich dieses Ziel?

Schauen wir die im vorangegangenen Abschnitt entwickelte Version unter diesen Gesichtspunkten an.

Stand der Vorgängerversion

Bei der Testmethode der Version 0 (siehe Listing 3.2 und Abbildung 3.1) handelt es sich um einen selbstüberprüfenden Integrationstest mit manuellem Testaufbau. Sie stellt einen ersten, sinnvollen Testfall dar.

Probleme der Vorgängerversion

Folgende Aspekte sind an der Testmethode problematisch:

- **Lesbarkeit**
  Die Testmethode ist schwer lesbar. Das liegt zum einen an der dynamischen Programmierung, mit der die Anwendung die unterschiedlichsten Stammdatenmodelle bearbeiten kann. Zum anderen wird diese technische Überfrachtung durch die mehrstufig verschachtelten Tabellen verursacht.
- **Viele Parameter**
  Wegen der Vielzahl an Parametern ist der Aufruf der Testmethode schwierig zu verstehen.
- **Dopplungen**
  Schon mit einer zweiten Testmethode müssten ganze Anweisungsblöcke verdoppelt werden.
- **Festwerte**
  Die Bedeutung von Festwerten wie `PFLI` ist vom Kontext abhängig. Einmal bezeichnet er einen Entitätstyp und ein anderes Mal ein Attribut. Bei einer Wertänderung ist eine fehleranfällige Einzelbearbeitung nötig.

Lösungswege zu einer Nachfolgeversion

Für diese Probleme gibt es folgende Lösungen:

- **Lesbarkeit**
  Bei schlechter Lesbarkeit wird schnell der Ruf nach einer ausführlichen Kommentierung laut. Allerdings sollten Sie bei Kommentaren stets beachten, dass diese ebenfalls gewartet werden müssen. Die Motivation für Kommentare ist häufig

eine Verbesserung des Überblicks und der Lesegeschwindigkeit, besonders in langen Methoden mit vielen technisch anspruchsvollen Anweisungen. Doch anstatt zusätzliche Kommentare einzufügen, sollten Sie den Code mit geeigneten (privaten) Methoden strukturieren. In vielen Fällen entsprechen die Namen dieser Methoden gerade dem Kurzkommentar, den Sie ansonsten über die betreffenden Anweisungen geschrieben hätten. Dies trifft auch auf die betrachtete Testmethode zu.

- **Viele Parameter**
  Eine *Verschalung* der zu testenden Methode erlaubt die Fokussierung auf die für den Test wichtigen Parameter und deren Argumente.
- **Dopplungen**
  Die Auslagerung oder *Extraktion* eines Anweisungsblocks in eine private Methode ist ebenso ein probates Mittel, um die Wiederverwendung zu ermöglichen und so Dopplungen innerhalb einer Klasse zu vermeiden.
- **Festwerte**
  Konstanten ermöglichen eine semantische Beschreibung ihrer Festwerte allein über ihren Namen. Bei Verwendung von Konstanten benötigen Sie folglich keine Kommentare mehr. Ferner verhindern Konstanten Buchstaben- bzw. Zahlendreher beim Schreiben und erlauben die zentrale und sichere Änderung eines Festwertes beim Überarbeiten. Da sie von der Codevervollständigung (engl. *Code Completion*) unterstützt werden, erhöhen sie zusätzlich die Schreibeffizienz.

3.2.2 Verbesserung der Definition der Testklasse

Listing 3.12 zeigt die Definition der Testklasse nach Umsetzung der im vorangegangenen Abschnitt beschriebenen Lösungen.

```
CLASS ltc_highlight_air_updates_1 DEFINITION FINAL
FOR TESTING DURATION SHORT RISK LEVEL HARMLESS.
  PRIVATE SECTION.
    METHODS setup.
    METHODS get_saved_pfli_changes FOR TESTING.

    METHODS create_pfli_key
      IMPORTING
        iv_carr_id TYPE mdg_s_carr_id
        iv_conn_id TYPE mdg_s_conn_id
      RETURNING
        VALUE(rsr_key) TYPE REF TO data.
    METHODS add_pfli_key
      IMPORTING
```

```
      isr_key TYPE REF TO data
    CHANGING
      ct_entity_type_keys TYPE usmd_gov_api_ts_ent_tabl.
  METHODS get_nodes_changes
    IMPORTING
      is_changes_scope    TYPE s_changes_scope
      it_entity_type_keys TYPE usmd_gov_api_ts_ent_tabl
    RETURNING
      VALUE(rt_entity_type_changes) TYPE usmd_t_changed_entities.
  METHODS find_changed_pfli
    IMPORTING
      isr_key               TYPE REF TO data
      it_entity_type_changes TYPE usmd_t_changed_entities
    RETURNING
      VALUE(rs_entity_changes) TYPE usmd_s_changed_entity.

  DATA mo_conv_api TYPE REF TO if_usmd_conv_som_gov_api.
ENDCLASS.
```

Listing 3.12 Definition der Testklasse (Version 1)

Der Name der Testklasse lautet nun `LTC_HIGHLIGHT_AIR_UPDATES_1`. Er drückt nun die Abhängigkeit der Testklasse von den Testdaten und dem Änderungsantrag ansatzweise aus. Es ist ja die Änderung der Flugverbindung der Fluggesellschaft `AIR`, die wir hier testen. Ebenso gibt der Name der Testmethode `GET_SAVED_PFLI_CHANGES` nun den Testfall präziser wieder. Die zusätzlichen Methoden repräsentieren wichtige logische Schritte der Testmethode. Auf sie gehe ich im folgenden Abschnitt ein.

3.2.3 Verbesserung der Implementierung der Testklasse

Die Implementierung der Testklasse der Version 1 ist auf mehrere Methoden verteilt: eine Setup-Methode, eine Testmethode und mehrere Hilfsmethoden.

Setup-Methode

Die Methode `SETUP` einer Testklasse wird vom ABAP-Unit-Testframework vor jeder Testmethode aufgerufen. Ein expliziter Aufruf am Anfang der Testmethode ist nicht notwendig.

```
METHOD setup.
  "Gegeben
  mo_conv_api =
    cl_usmd_conv_som_gov_api=>get_instance( gc_flight_model ).
```



```
  mo_conv_api->set_environment( iv_crequest_id = gc_crequest_id ).
ENDMETHOD.
```

Listing 3.13 Setup-Methode der Testklasse

Wie Listing 3.13 zeigt, enthält die *Setup-Methode* genau die zwei Anweisungen, mit denen auch die Testmethode der Version 0 startet. Der einzige Unterschied liegt in der Ersetzung der Festwerte durch Konstanten. Da die Testklasse auf diesen Änderungsantrag festgelegt ist, beginnt auch jede weitere Testmethode mit diesen Anweisungen. Damit dient die Setup-Methode unmittelbar dem *DRY-Prinzip* (Don't Repeat Yourself).

Dem *Gegeben-Wenn-Dann-Entwurfsmuster* (engl. *Given-When-Then-Pattern*) zufolge sollte für eine Testmethode klar erkennbar sein, welche Ausgangssituation nötig ist (Gegeben-Teil), damit ein Aufruf der zu testenden Methode (Wenn-Teil) eine bestimmte Endsituation hervorbringt (Dann-Teil). In unserer Testmethode gehört allerdings nicht nur die Setup-Methode zum Gegeben-Teil, sondern auch die manuelle Aktivität, die mithilfe des Änderungsantrags `GC_REQUEST_ID` die Flugverbindung `GC_CONNECTION_ID` der Fluggesellschaft `GC_CARRIER_ID` geändert hat.

Testmethode

Wie Listing 3.14 zeigt, startet diese Testmethode gleich mit dem Wenn-Teil des Testfalls. Dieser Teil umfasst neben dem Aufruf der zu testenden Methode auch alle Anweisungen, die zur Bereitstellung der Argumente dieses Methodenaufrufs (hier `LT_ENTITY_TYPE_KEYS`) benötigt werden.

```
METHOD get_saved_pfli_changes.
  DATA lt_entity_type_keys TYPE usmd_gov_api_ts_ent_tabl.

  "Wenn
  DATA(lsr_pfli_key) = create_pfli_key(
      iv_carr_id = gc_carrier_id
      iv_conn_id = gc_connection_id
  ).
  add_pfli_key(
    EXPORTING
      isr_key             = lsr_pfli_key
    CHANGING
      ct_entity_type_keys = lt_entity_type_keys
  ).
  data(lt_entity_type_changes) = get_nodes_changes(
      is_changes_scope    = gcs_only_saved_changes
      it_entity_type_keys = lt_entity_type_keys
```

```
  ).

  "Dann
  data(ls_entity_changes) = find_changed_pfli(
      isr_key                = lsr_pfli_key
      it_entity_type_changes = lt_entity_type_changes
  ).
  cl_abap_unit_assert=>assert_true(
      ls_entity_changes-saved_change
  ).
ENDMETHOD.
```

Listing 3.14 Implementierung der Testmethode (Version 1)

Inline-Deklarationen

Mithilfe von *Inline-Deklarationen* kann die Anzahl der am Anfang der Testmethode zu deklarierenden Variablen stark reduziert werden. Inline-deklarierte Variablen sind auch im weiteren Verlauf der Methode ohne Einschränkungen wiederverwendbar.

In dieser Testmethode bilden die Rückgabeparameter der Methoden `CREATE_PFLI_KEY`, `GET_NODES_CHANGES` und `FIND_CHANGED_PFLI` die Grundlage für die Inline-Deklaration der Variablen `LSR_PFLI_KEY`, `LT_ENTITY_TYPE_CHANGES` und `LS_ENTITY_CHANGES`. Das System erlaubt Inline-Deklarationen allerdings nur für Ausgabe- und Rückgabeparameter. Die Variable `LT_ENTITY_TYPE_KEYS`, die zuerst das Argument eines `CHANGING`-Parameters ist, muss deshalb noch klassisch am Anfang der Testmethode deklariert werden.

Verschalung der zu testenden Methode

Neben der Verwendung von Konstanten konzentriert sich diese Version 1 auf die Einführung von privaten Hilfsmethoden. Diese Hilfsmethoden kapseln technische Details und machen mit ihren bewusst gewählten Bezeichnern Kommentare überflüssig. Die Methoden `CREATE_PFLI_KEY`, `ADD_PFLI_KEY`, `GET_NODES_CHANGES` und `FIND_CHANGED_PFLI` sind dafür allesamt treffende Beispiele.

Wie Listing 3.15 verdeutlicht, handelt es sich bei `GET_NODES_CHANGES` um eine spezialisierte Verschalung der zu testenden Methode.

```
METHOD get_nodes_changes.
  rt_entity_type_changes = mo_conv_api->get_entity_field_changes(
      iv_struct            = zif_usmd=>c_struct_key_attr
      it_entity_keys       = it_entity_type_keys
      iv_saved_changes     = is_changes_scope-saved
      iv_unsaved_changes   = is_changes_scope-unsaved
```



```
      iv_contained_changes = abap_false
  ).
ENDMETHOD.
```

Listing 3.15 Verschalung der zu testenden Methode

Auffällig ist zuallererst die Reduzierung der Zahl der Eingabeparameter von fünf auf zwei. Weniger Parameter bedeuten in vielen Fällen eine wesentlich geringere Komplexität. Weniger Parameter haben deshalb für die schnelle und breite Verständlichkeit eine große Bedeutung. Wie ich an der Methode `GET_NODES_CHANGES` beispielhaft zeige, können Sie die Anzahl der Parameter auf verschiedene Arten reduzieren:

- **Spezifisches Argument in den Methodennamen integrieren**
  Ihrem Namen `GET_NODES_CHANGES` zufolge konzentriert sich die Methode auf Änderungen einzelner Knoten des Entitätenbaumes. Es muss also für jede in die Berechnung einzubeziehende Entität ein eigener Schlüssel eingegeben werden. Das entspricht genau dem Wert, den die Methode an `IV_CONTAINED_CHANGES` übergibt.

  Für den komplementären Fall, dass neben den gegebenen Entitäten auch für alle abhängigen (enthaltenen) Entitäten die Änderungen berechnet werden sollen, ist dann eine zweite Methode `GET_TREES_CHANGES` nötig. Da dieser Ansatz die Lesbarkeit maßgeblich verbessert, ist diese Maßnahme für Testcode in jedem Fall eine lohnende Investition. Auch für Produktcode lohnt sie sich oftmals, wozu Sie in Abschnitt 9.2, »Regeln für die Signatur einer Methode«, mehr erfahren.
- **Aggregation von Argumenten**
  Für die Aggregation von Argumenten sollten Sie in den meisten Fällen Objekte verwenden und für diese Objekte kleine lokale oder globale Klassen definieren. In unserem Beispiel bietet sich allerdings auch eine Struktur mit Konstanten wie in Listing 3.16 an.

```
TYPES:
  BEGIN OF s_changes_scope,
    saved   TYPE abap_bool,
    unsaved TYPE abap_bool,
  END OF s_changes_scope.

CONSTANTS:
  BEGIN OF gcs_only_saved_changes,
    saved   TYPE abap_bool VALUE abap_true,
    unsaved TYPE abap_bool VALUE abap_false,
  END OF gcs_only_saved_changes.
```

Listing 3.16 Definition einer konstanten Struktur

Technisch lässt sich diese Strukturkonstante der Basisklasse auch wie folgt definieren, aber sie dann eben nicht mehr so gut lesbar:

```
CONSTANTS cs_only_saved_changes TYPE s_changes_scope VALUE 'X '.
```

Mit dem Namensteil `ONLY` erreiche ich hier eine recht genaue Beschreibung der zwei spezifischen, booleschen Einzelwerte. Für eine vollständige Testabdeckung werden im Laufe der Zeit alle vier möglichen Strukturkonstanten benötigt. Es ist aber eine gute Praxis, diese erst bei tatsächlichem Bedarf zu ergänzen, zum einen weil ungenützter Code den aktuellen Stand der Implementierung verschleiert, zum anderen weil zusätzlicher Code auch zusätzlichen Aufwand beim Refactoring bedeutet.

- **Kontextabhängige Umwandlung eines Arguments in eine Konstante**

  Da innerhalb dieser Testklasse der Parameter `IV_STRUCT` stets das Argument `ZIF_USMD=>C_STRUCT_KEY_ATTR` erhält, kann dies in der Hilfsmethode auch hart codiert werden. Gleiches gilt natürlich für obsolete Parameter, also für Parameter, die nicht mehr gebraucht werden, aber aus Gründen der Abwärtskompatibilität nicht entfernt werden können.

3.2.4 Diagramme

Abbildung 3.3 zeigt das Blockdiagramm der vor allem mithilfe von privaten Methoden verbesserten Version 1 der Testklasse.

Um eine Überladung mit Details zu vermeiden, ist dieses Blockdiagramm trotz seiner Detailtiefe nicht vollständig. Das muss es aber auch nicht sein, weil etwa die Methode `CREATE_PFLI_KEY` für das Verständnis der Testmethode von untergeordneter Bedeutung ist. Der Fokus liegt auf der Einbeziehung der `PFLI`-Entität in die Berechnung. Um das Gegeben-Wenn-Dann-Entwurfsmuster zu visualisieren, habe ich den rechten Testblock um gestrichelte Trennlinien ergänzt.

Das Klassendiagramm in Abbildung 3.4 hat sich gegenüber der Vorgängerversion ebenfalls nur im schwarzen Testbereich verändert. Die Testklasse hat immer noch nur eine Testmethode, weil sowohl die `SETUP`-Methode als auch die privaten Hilfsmethoden keinen Zusatz `FOR TESTING` haben.

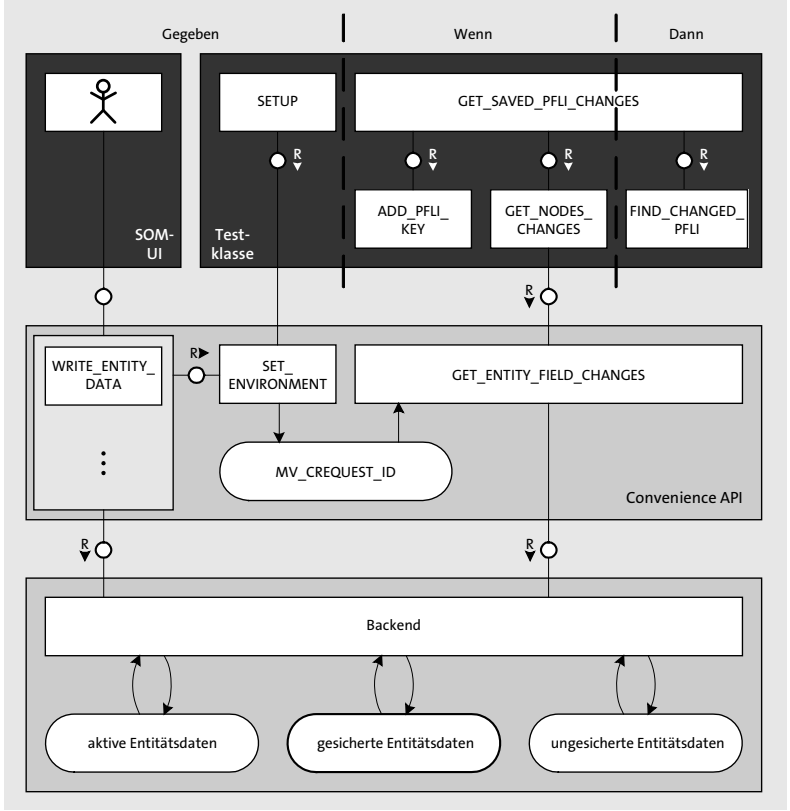

Abbildung 3.3 Zwei Schichten von Methoden innerhalb der Testklasse

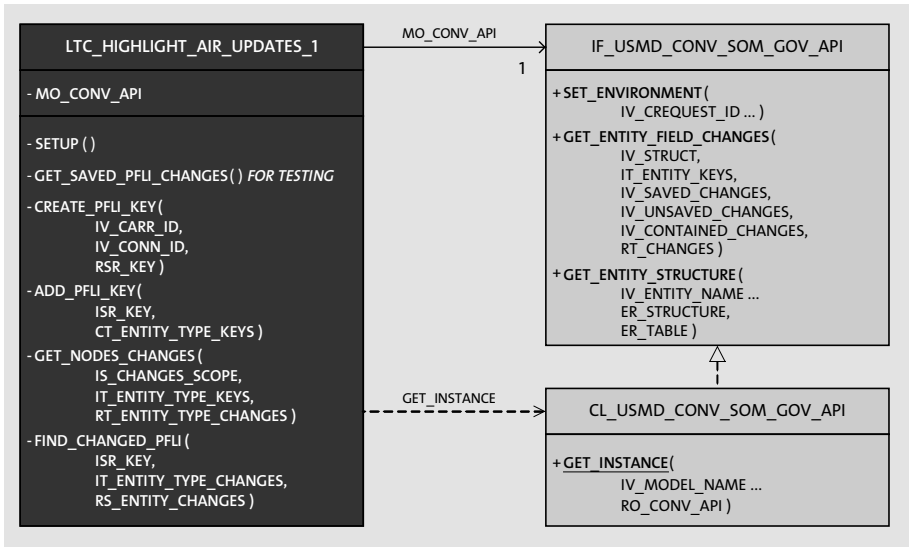

Abbildung 3.4 Hilfsmethoden als Bausteine für weitere Testmethoden

Setup-Methode und allgemeine Clean-Code-Prinzipien

Es gibt nun eine Setup-Methode, die auch von weiteren Testmethoden verwendet wird (*Implicit-Setup-Entwurfsmuster*). Die Hilfsmethoden der Testklasse geben den wiederverwendbaren Schritten der Testfälle einen sprechenden Namen (*Test-Utility-Method-Entwurfsmuster*). Konstanten geben mit ihren Namen die Semantik von Festwerten wieder (*Literal-Values-Entwurfsmuster*).

Version 1 der Testklasse hat mit einer guten Lesbarkeit (durch die Verbergung technischer Details in Methoden mit aussagekräftigen Namen) und der Vermeidung von Dopplungen (durch die Auslagerung von Anweisungsblöcken zur Wiederverwendung) schon viel zu einem sauberem Code beigetragen (Clean-Code-Prinzipien). Diesen Qualitätsstand sollten Sie in jedem Fall erreichen, denn dann könnte sogar ein weniger erfahrener Kollege die Testabdeckung mit weiteren Methoden vervollständigen. Ebenso haben Sie den Aufwand für die Entwicklung der nächsten Testmethode maßgeblich reduziert, sodass einer testgetriebenen Wartung nichts mehr im Wege stehen sollte.

3.3 Testorientierte Clean-Code-Prinzipien

Teilnehmer an der SAP-Schulung »ABAP OO in der Praxis« vermuten regelmäßig, dass die Version 1 der Testklasse nicht mehr maßgeblich verbessert werden kann. Doch mit Version 2 erhöhe ich die Lesbarkeit der Testmethode noch einmal deutlich. Grundlage dafür ist das von mir entwickelte *Testklassen-Entwurfsmuster*. Doch bevor ich Ihnen dieses Entwurfsmuster vorstelle, möchte ich eine Bestandsaufnahme des Verbesserungsprozesses nach dem bereits bekannten Muster durchführen.

3.3.1 Verbesserungsprozess

Version 2 der Testklasse löst einige der Probleme, die für Version 1 noch bestehen.

Stand der Vorgängerversion

Die Testmethode der Version 1 ist immer noch ein selbstüberprüfender Integrationstest mit manuellem Testaufbau. Der Name der Testklasse beschreibt ihre *Zuständigkeit* (engl. *Responsibility*), der Name der Testmethode den Testfall. Des Weiteren stehen in Form von privaten Methoden logische Blöcke zur Wiederverwendung in weiteren Testmethoden zur Verfügung.

Probleme der Vorgängerversion

Folgende Aspekte sind an der Testmethode noch problematisch:

- **Technische Details**
  Die Namen der Ein- und Ausgabetabellen der Hilfsmethoden sind ebenfalls technische Details. Entscheidend ist, welche Entitäten eingegeben und welche Änderungen damit gefunden werden, und nicht, in welchen konkreten Tabellen diese Daten stecken.
- **Zwischenschritt**
  Das Suchen der geänderten Entität ist ein notwendiger Zwischenschritt, der aber für das Verständnis des Tests nicht hilfreich ist.

Lösungswege zu einer Nachfolgeversion

Für diese Probleme gibt es folgende Lösungen:

- **Technische Details**
  Wenn sich eine Testklasse auf Testfälle für eine zu testende Methode konzentriert, dann darf die Testklasse auch Attribute für die Ein- und Ausgabeparameter dieser Methode besitzen.
- **Zwischenschritt**
  Mit einer *Custom-Assertion*, also einer applikationsspezifischen Assertion-Methode, lässt sich die Suche nach den gesicherten Änderungen und deren Überprüfung zusammenfassen.

3.3.2 Verbesserung der Implementierung der Testklasse

In Listing 3.17 wurde die Setup-Methode um die Erzeugung des Flugverbindungsschlüssels erweitert. Das ist nur konsequent, weil sich die Testklasse ja auf Testfälle zur Flugverbindung `AIR 0001` beschränkt und damit dieser Schlüssel grundlegend für alle Testmethoden ist. Für die Bereitstellung dieses Entitätsschlüssels in der Testmethode ist das Attribut `MSR_PFLI_KEY` notwendig. Des Weiteren sind die Konstanten nun in der Testklasse definiert, die sie verwendet. Ihre Namen benötigen deshalb das Präfix `G` für global nicht mehr.

```
METHOD setup.
  mo_conv_api =
    cl_usmd_conv_som_gov_api=>get_instance( c_flight_model ).
  mo_conv_api->set_environment( iv_crequest_id = c_crequest_id ).

  msr_pfli_key = create_pfli_key(
      iv_carr_id = c_carrier_id
      iv_conn_id = c_connection_id
  ).
```

```
ENDMETHOD.

METHOD get_saved_pfli_changes.
  import_pfli_key( msr_pfli_key ).
  get_nodes_changes( cs_only_saved_changes ).
  assert_saved_pfli_change( msr_pfli_key ).
ENDMETHOD.
```

Listing 3.17 Implementierung von Setup- und Testmethode (Version 2)

Die Testmethode GET_SAVED_PFLI_CHANGES benötigt lediglich noch drei Methodenaufrufe, um den Testfall zu beschreiben. Jeder dieser drei Methodenaufrufe benötigt nur noch einen Parameter. Bevor ich Ihnen zeige, wie das möglich ist, möchte ich bewusst die Frage voranstellen, ob das denn überhaupt sinnvoll ist. Denn wie Sie sicher aus eigener Erfahrung wissen, ist eine kompaktere Programmierung nicht immer auch besser verständlich. Manchmal ist sogar das Gegenteil der Fall, vor allem für Entwickler, die neu im Team sind oder sich seltener mit diesem Code beschäftigen. Das *KISS-Prinzip* (*keep it simple and stupid*), das generell eine möglichst einfache Lösung eines Problems einfordert, ist in jedem Fall zu berücksichtigen.

Um die Verständlichkeit zu prüfen, lesen Sie sich die Testmethode einmal vor. Die erste Anweisung besagt, dass der Flugverbindungsschlüssel MSR_PFLI_KEY von der zu testenden Methode importiert wird. Wie das genau geschieht, ist für das erste Verständnis des Testfalls von untergeordneter Bedeutung. Mit der zweiten Anweisung werden ausschließlich gesicherte Änderungen zu den Entitäten berechnet, deren Schlüssel vorher importiert wurden. Die dritte Anweisung behauptet, dass es für die betrachtete Flugverbindung (mindestens) eine gesicherte Änderung gibt.

Wahrscheinlich werden Sie sich beim Lesen nicht bei jeder Anweisung *ganz* sicher sein, ob Sie mit Ihrem intuitiven Verständnis richtig liegen. Doch ich nehme an, dass Sie den so formulierten Testfall auf Anhieb einordnen können. Jetzt liegt es an Ihnen, in welche Hilfsmethode Sie beim Lesen oder mit dem Debugger navigieren wollen.

Wenn-Teil

Listing 3.18 führt zwei Navigationsschritte für die erste Anweisung der Testmethode aus. Im ersten Schritt zeigt sich, dass das Importieren eines Flugverbindungsschlüssels ein Spezialfall des Hinzufügens dieses Schlüssels mit dem Entitätstyp PFLI ist. Im zweiten Schritt zeigt sich, wie dieser Schlüssel der geschachtelten Tabelle MT_IMP_ENTITY_TYPE_KEYS genau hinzugefügt wird.

```
METHOD import_pfli_key.
  add_key(
      iv_entity_type = c_pfli_type
      isr_key        = isr_key
```



```
  ).
ENDMETHOD.

METHOD add_key.
  DATA ls_entity_type_keys TYPE usmd_gov_api_s_ent_tabl.
  FIELD-SYMBOLS <t_key> TYPE INDEX TABLE.

  ASSIGN isr_key->* TO FIELD-SYMBOL(<s_key>).

  ls_entity_type_keys-entity = iv_entity_type.
  mo_conv_api->get_entity_structure(
    EXPORTING
      iv_entity_name = iv_entity_type
    IMPORTING
      er_table       = ls_entity_type_keys-tabl
  ).
  ASSIGN ls_entity_type_keys-tabl->* TO <t_key>.
  INSERT <s_key> INTO TABLE <t_key>.
  INSERT ls_entity_type_keys INTO TABLE mt_imp_entity_type_keys.
ENDMETHOD.
```

Listing 3.18 Datenbereitstellung für die zu testende Methode

Die zweistufige Implementierung von IMPORT_PFLI_KEY ist darin begründet, dass weitere Testmethoden auch weitere Entitätstypen benötigen werden. Deren importierende Hilfsmethoden sollten auf keinen Fall den Code der Methode ADD_KEY verdoppeln.

Natürlich wäre auch eine generische Methode IMPORT_KEY mit einem zusätzlichen Parameter IV_ENTITY_TYPE möglich, aber da diese Hilfsmethode in der Testmethode selbst aufgerufen wird, würde dieser zweite Parameter auf Kosten des Leseflusses gehen. Genau dieses Sichtbarkeitsproblem hat die untergeordnete Hilfsmethode ADD_KEY nicht, und so ergänzen sich die beiden ideal.

Die Implementierung der Methode ADD_KEY zeigt mit der ersten Anweisung, dass auch Feldsymbole inline deklariert werden können. Am Ende dieser Methode sehen Sie, dass der gegebene Entitätsschlüssel zusammen mit seinem Entitätstyp in das Attribut MT_IMP_ENTITY_TYPE_KEYS der Testklasse eingefügt wird.

Wie Listing 3.19 zeigt, wird diese Schlüsseltabelle, ihrem Namen entsprechend, in der Hilfsmethode GET_NODES_CHANGES von der zu testenden Methode importiert.

```
METHOD get_nodes_changes.
  mt_act_entity_type_changes =
    mo_conv_api->get_entity_field_changes(
      iv_struct            = zif_usmd=>c_struct_key_attr
```

```
      it_entity_keys       = mt_imp_entity_type_keys
      iv_saved_changes     = is_changes_scope-saved
      iv_unsaved_changes   = is_changes_scope-unsaved
      iv_contained_changes = abap_false
  ).
ENDMETHOD.
```

Listing 3.19 Datenaustausch mit der zu testenden Methode

Im gleichen Zuge setzt diese Hilfsmethode mit der Rückgabe der zu testenden Methode das Attribut `MT_ACT_ENTITY_TYPE_CHANGES` der Testklasse.

Dann-Teil

Die Hilfsmethoden in Listing 3.20 durchsuchen diese Tabelle der tatsächlichen (engl. *actual*) Entitätsänderungen dann mit der Erwartung, darin auch gesicherte Änderungen für die gegebene Flugverbindungsentität zu finden.

```
METHOD assert_saved_pfli_change.
  DATA(ls_entity_changes) = find_changed_entity(
      iv_entity_type = c_pfli_type
      isr_key        = isr_key
  ).
  cl_abap_unit_assert=>assert_true(
      ls_entity_changes-saved_change
  ).
ENDMETHOD.

METHOD find_changed_entity.
  FIELD-SYMBOLS <s_changes_key> TYPE any.

  READ TABLE mt_act_entity_type_changes
    ASSIGNING FIELD-SYMBOL(<s_entity_type_changes>)
    WITH KEY entity_type = iv_entity_type
             struct      = zif_usmd=>c_struct_key_attr.
  IF sy-subrc <> 0.
    RETURN.
  ENDIF.

  ASSIGN isr_key->* TO FIELD-SYMBOL(<s_key>).
  LOOP AT <s_entity_type_changes>-changed_entities
  INTO rs_entity_changes.
    ASSIGN rs_entity_changes-entity->* TO <s_changes_key>.
    IF <s_changes_key> = <s_key>.
      RETURN.
    ENDIF.
  ENDLOOP.

  CLEAR rs_entity_changes.
ENDMETHOD.
```

Listing 3.20 Datenüberprüfung für die zu testende Methode

Dabei führt wieder die Verschalung einer generischen Methode mit mehreren Parametern durch eine entitätstypspezifische Methode mit weniger Parametern zu guter Lesbarkeit bei geringer Dopplung.

3.3.3 Diagramme

Das Blockdiagramm in Abbildung 3.5 visualisiert zum einen die Schichtung der Methoden in der Testklasse. Zum anderen zeigt es das Testklassen-Entwurfsmuster in einprägsamer Weise.

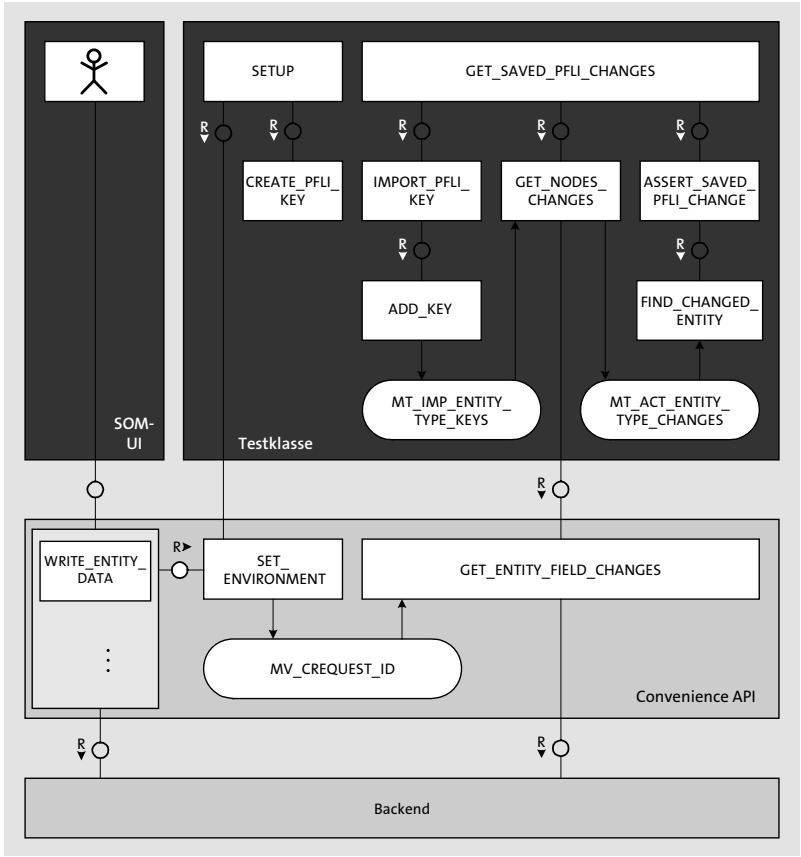

Abbildung 3.5 Drei Schichten von Methoden innerhalb der Testklasse

Die Eingabedaten der zu testenden Methode bestehen nicht nur aus Parametern der sie verschalenden Hilfsmethode, sondern auch aus Attributen der Testklasse. Diese Eingabeattribute (meistens Tabellen) sollten zur schnelleren Einordnung das Kürzel `IMP` bzw. `IMPORT` in ihrem Namen tragen. Die Testmethode sollte für das Setzen oder Befüllen dieser Eingabeattribute Hilfsmethoden verwenden, deren Namen allesamt mit `IMPORT` beginnen.

Auf der anderen Seite übergibt die zu testende Methode ihre Ausgabedaten nicht nur an die Parameter der sie verschalenden Hilfsmethode, sondern auch an weitere Attribute der Testklasse, die ihrerseits das Kürzel `ACT` bzw. `ACTUAL` in ihrem Namen tragen sollten. Zuletzt sollte die Testmethode maßgeschneiderte Überprüfungsmethoden verwenden, deren Namen allesamt mit `ASSERT` beginnen.

Dieses Entwurfsmuster sollte aber nur auf eine Testklasse angewendet werden, die sich auf das Testen einer einzigen Produktmethode konzentriert. Ansonsten geht ihr innerer Zusammenhalt nach und nach verloren, denn die Attribute und Testmethoden zu einer Produktmethode sind in der Regel disjunkt zu denjenigen einer anderen.

Das Klassendiagramm in Abbildung 3.6 soll Ihnen als Ersatz für die Definition der Testklasse dienen, für die ich in diesem Abschnitt kein Listing vorgesehen habe. Es fehlen zwar die Typen, aber diese ändern sich über die Versionen hinweg sowieso nicht mehr.

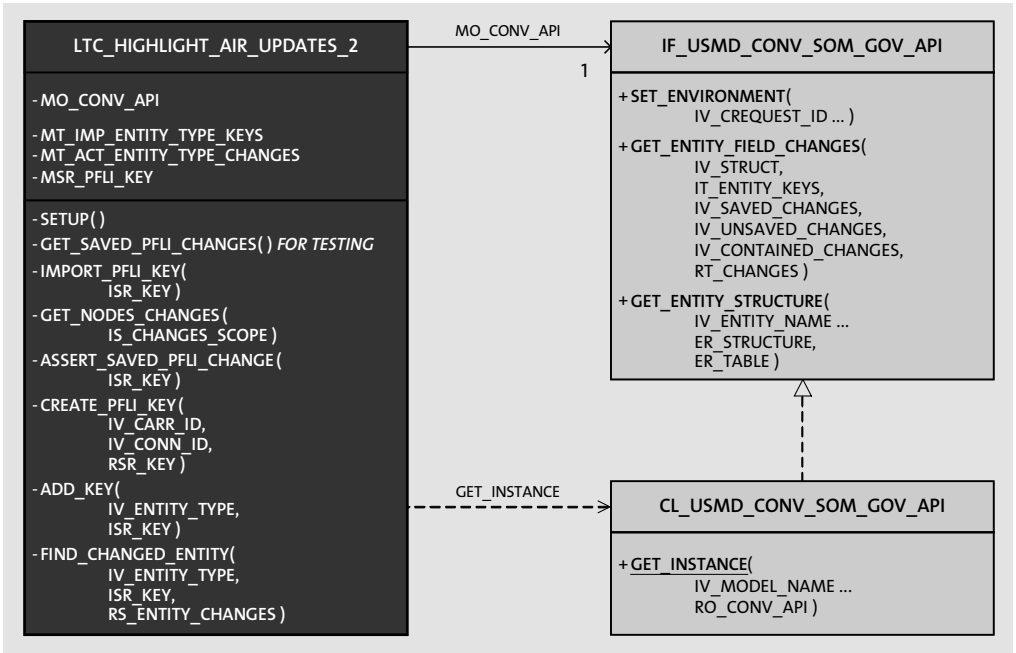

Abbildung 3.6 Attribute für den Datenfluss unterhalb der Testmethode

Custom-Assertions und Testklassen-Entwurfsmuster

Sie können die Lesbarkeit einer Testmethode deutlich erhöhen, wenn Sie ihr Ergebnis von Custom-Assertions überprüfen lassen.

Sie sollten die Testmethoden nicht nur anhand der von diesen Methoden getesteten Funktionalität (hier das Hervorheben von Änderungen) auf Testklassen verteilen, wie es das *Test-Class-Per-Feature-Entwurfsmuster* vorsieht. Vielmehr sollten Sie für jede hinreichend komplexe Produktmethode eine eigene Testklasse vorsehen. Damit schaffen Sie die Voraussetzungen für das Testklassen-Entwurfsmuster, das mit weiteren Attributen und Hilfsmethoden eine nicht technische Beschreibung des Testfalls erlaubt, und das, ohne dass der Testfall undurchsichtig wird (*Obscure-Test-Antimuster*). Dass sich eine Testklasse auf eine Produktmethode konzentriert, ist ganz im Sinne des *Single-Responsibility-Prinzips*.