# Open Diabetes UAM
# Heuristik Algorithmen

**Qualitätssicherungsdokument**


| | |
|---|---|
| Gruppe 11: | Aino Schwarte <aino.schwarte@stud.tu-darmstadt.de><br>Anna Mees <anna.mees@stud.tu-darmstadt.de><br>Jan Paul Petto <janpaul.petto@stud.tu-darmstadt.de><br>Paul Wolfart <paul.wolfart@stud.tu-darmstadt.de><br>Tom Großmann <tom.grossmann@stud.tu-darmstadt.de> |
| Teamleiter: | Benedikt Schneider <schneider-benedikt@gmx.net> |
| Auftraggeber: | M.Sc. Jens Heuschkel <heuschkel@tk.tu-darmstadt.de><br>Telecooperation<br>Smart Urban Networks |
| Abgabedatum: | 31.03.2019 |


TECHNISCHE
UNIVERSITÄT
DARMSTADT


Bachelor-Praktikum WS 2018/2019
Fachbereich Informatik

## Inhaltsverzeichnis

## 1 Einleitung

Das sogenannte Un-Announced-Meal-Problem (UAM) ist derzeit eine der größten Therapieschwächen bei der Behandlung von Diabetes Mellitus Typ 1[1] und beschreibt die Mahlzeiten, welche vom Patienten nicht berücksichtigt wurden und somit den Blutzuckerspiegel unkontrolliert steigen lassen. Denn bei Diabetespatienten produziert der Körper kaum bis gar kein eigenes Insulin, welches den Blutzuckerpegel reguliert. Das Insulin muss manuell verabreicht werden. Neben der Grundversorgung wird zusätzlich auch eine zu jeder Mahlzeit präzise berechnete Insulindosis benötigt, um den Blutzuckerspiegel innerhalb akzeptabler Grenzwerte zu halten. Extrem hohe oder niedrige Blutzuckerwerte können akute schwerwiegende und sogar tödliche Folgen haben. Befindet sich der Blutzuckerspiegel regelmäßig und über längere Zeiträume außerhalb der Normalwerten, kann dies auch zu schweren Langzeitschäden führen.

Um in Zukunft ein voll automatisches System mit Messgerät und Insulinpumpe zu ermöglichen, welches die Insulinversorgung des Patienten übernimmt, wird eine Möglichkeit benötigt, Mahlzeiten verlässlich zu erkennen. Bisherige Systeme können mit der Problematik von unangekündigten Mahlzeiten nicht umgehen, wenn eine Mahlzeit vom Patienten vergessen oder zu gering eingeschätzt wurde. Das dadurch fehlende Insulin wird bestenfalls allmählich mit dem steigendem Blutzuckerspiegel verabreicht, was allerdings zu spät ist, um hohe Werte zu verhindern. In solchen Situationen kann es zu einer lebensbedrohlichen Überkorrektur kommen.

Das Ziel unseres Projekts ist es anhand des steigenden Blutzuckers mit wenigen Messwerten eine Mahlzeit präzise zu berechnen, damit die adäquate Menge an Insulin schon frühzeitig injiziert werden kann. Dazu bauen wir auf dem Open Source Projekt Nightscout[2] auf, welches der Visualisierung von gemessenen Blutzuckerwerten dient. Die Messungen werden alle fünf Minuten von einem Sensor unter der Haut durchgeführt. Über die Cloud werden sie in einer Datenbank gespeichert und können auf jedem internetfähigen Gerät abgerufen werden. Zusätzlich werden Insulindosierungen und bekannte Mahlzeiten als Events kenntlich gemacht. Unser Programm bezieht diese Messwerte und Events aus der Datenbank und startet einen Algorithmus zur Berechnung von Mahlzeiten. Da es keine bekannte optimale Methode gibt, um Mahlzeiten zu erkennen, implementieren wir unterschiedliche Ansätze, aus denen gewählt werden kann. Diese Ansätze beruhen auf verschiedenen wissenschaftlichen Publikationen, welche die Auswirkungen von Kohlenhydraten und Insulin auf den Blutzuckerspiegel mathematisch beschreiben. Wir benutzen diese Modelle um von den gemessenen Blutzuckerwerten und bekannten Insulinbehandlungen auf Kohlenhydraten, also Mahlzeiten, zurück zu schließen. Die berechneten Mahlzeiten werden dann als neue Events in Nightscout eingetragen und je nach Ansatz auch im weiteren Verlauf berücksichtigt.

Es ist nicht sicher, ob sich das UAM-Problem mit dem aktuellen Sachverstand der Wissenschaft zuverlässig lösen lässt. Unsere Ansätze können höchstens so gut sein, wie die Modelle, die uns zur Verfügung stehen. Unser Projekt dient also als Plausibilitätsprüfung, ob sich das UAM-Problem zurzeit realistisch lösen lässt. Wir werden unsere Algorithmen, wenn sie sich als verlässlich erweisen, der Nightscout Community zur Verfügung stellen.

[1] Diabetes Typ 2 fällt nicht unter das UAM-Problem, da es sich dabei lediglich um eine Insulinresistenz handelt. Eine nicht erkannte Mahlzeit stellt also ein geringeres Risiko dar und die Krankheitsbilder lassen sich auch nicht direkt vergleichen.

[2] http://www.nightscout.info

# 2 Qualitätsziele

## 2.1 Korrektheit

### 2.1.1 Beschreibung

Unser wichtigstes QS-Ziel ist die Korrektheit. Dabei liegt das Hauptaugenmerk neben der Erzeugung von korrekten Daten auch darauf, Fehler und Fehlerquellen korrekt zu erkennen. Wenn die zur Verfügung stehenden Daten zur Berechnung einer Mahlzeit nicht ausreichen oder sonstige Probleme auftreten, muss der Benutzer darüber in Kenntnis gesetzt werden und es dürfen keine potentiell falschen Ergebnisse produziert werden. Diabetespatienten, die unsere Algorithmen verwenden werden, vertrauen darauf, dass unser Ansatz korrekte Werte zurück gibt und Mahlzeiten richtig erkannt werden. Da ein zu hoher oder zu niedriger Insulinwert, wie bereits erläutert, schwere körperliche Langzeitfolgen haben oder sogar akut lebensbedrohlich sein kann, ist es offensichtlich, weshalb hier keine Fehler passieren dürfen.

### 2.1.2 Maßnahmen

Wir wollen die Korrektheit durch die die Nutzung von automatischen Coverity Scans[1] und dem Continuous Integration Tool Travis CI[2] erreichen. Travis CI kann kostenlos genutzt werden, da wir an einem Open-Source-Projekt arbeiten, welches auf GitHub[3] veröffentlicht wird.

Travis CI kompiliert die Software und überprüft automatisch alle implementierten Tests und meldet zurück, ob diese erfolgreich abgeschlossen wurden.

Coverity Scans analysieren den Code auf Race Conditions und Speicherlecks, bei denen zwar Arbeitsspeicher belegt, allerdings weder genutzt, noch frei gegeben wird.

Wir verwenden mehrere Branches für zu entwickelnde Features und einen Master Branch, auf dem immer eine lauffähige Version liegt. Auf dem Master Branch kann nicht direkt gepusht werden, nur über die Feature Branches. Es sind Pull-Request-Reviews und Status Checks nötig, um auf den Master Branch zu schreiben. Diese Status Checks beinhalten das erfolgreiche Kompilieren des Codes durch Travis CI und den erfolgreichen Durchlauf aller Tests. Erst im Anschluss können Pull-Requests akzeptiert werden. Wenn mindestens eine andere Person den Pull-Request überprüft und akzeptiert hat, wird der Branch gemerged. Git führt einen Verlauf, wer den Pull-Request-Review freigegeben hat. Es wird somit immer protokolliert, wer Korrektur gelesen hat. Dieses Protokoll wird im Anhang zur Verfügung gestellt.

### 2.1.3 Prozessbeschreibung

In GitHub gibt es einen Master Branch. Entwickelt wird ausschließlich in sogenannten Feature Branches. Für jedes Feature bzw. jede Feature-Gruppe wird ein eigener Branch erstellt. Nach

[1] https://scan.coverity.com
[2] https://travis-ci.org
[3] http://github.com/TUDa-BP-11/opendiabetes-uam-heuristik

Abschluss des Features und der Iterationszyklen wird ein Pull-Request erstellt. Mindestens eine andere Person im Projekt kontrolliert die Änderungen im Pull-Request, bevor dieser akzeptiert wird. Werden Mängel oder Probleme entdeckt, löst diese die Person, die den Pull-Request erstellt hat. Dies soll innerhalb der nächsten drei Tage geschehen und darf maximal auf sieben Tage verlängert werden. Sollte nach dieser Zeit noch keine Lösung gefunden werden, wir dies beim nächsten Auftraggebertreffen besprochen. Treten dabei Probleme oder Schwierigkeiten auf, wird die Hilfe der Teammitglieder in Anspruch genommen. Danach werden die Änderungen erneut von mindestens einer anderen Person kontrolliert, bis keine Mängel mehr gefunden werden und der Branch in den Master Branch gemerged werden kann.

Travis CI wird automatisch bei jedem Git-Commit ausgeführt. Dafür wurde eine entsprechende .travis.yml Datei angelegt. Travis CI kompiliert die Software und startet alle implementierten Tests. Schlägt einer der Tests fehl, werden wir per E-Mail informiert. Außerdem vergibt Travis CI Badges, anhand derer in GitHub sichtbar ist, welche Branches in ihrem aktuellen Stand ohne Probleme kompiliert und getestet wurden.

Die Tests bestehen aus statischen Unit-Tests, welche zunächst alle grundlegenden Funktionen testen, ohne welche die anderen Funktionen nicht arbeiten können, wie zum Beispiel das korrekte Verbinden auf eine Nightscout Test Instanz und die volle Funktionalität der API-Schnittstelle. Außerdem werden alle verwendeten mathematischen Modelle - zum Beispiel zur Berechnung von Insulinabbau über Zeit im Blutkreislauf - mit festen Testdaten auf Korrektheit überprüft. Diese Testdaten wurden zuvor unabhängig von der Implementierung der Funktionen berechnet, wir testen die Funktionen also nicht mit Werten, die durch die Funktionen selbst generiert wurden. In einem zweiten Schritt werden - abhängig vom Status und Inhalt der in der Nightscout Test Instanz gespeicherten Daten - dynamisch weitere Tests generiert, die die implementierten Algorithmen auf verschiedene Weisen testen. Dies ist stark von den jeweiligen Algorithmen abhängig, wir testen aber mindestens jede offen verfügbare (`public`) Methode. Dafür verwenden wir die in JUnit-5 integrierte dynamische Test-Generation. Methoden die nicht automatisch getestet werden können müssen durch Tests des Erstellers getestet werden. Dies stellt sicher, dass sie fehlerfrei funktionieren. Sollten hierbei Fehler auftreten, werden diese mit sinnvollen Fehlermeldungen ausgegeben, so dass diese behoben werden können und der Nutzer informiert ist. Auf diese Weise können wir Tests mit verschiedenen Beispieldaten problemlos mehrmals ausführen und auch die Möglichkeit anbieten, das Programm mit eigenen Testdaten zu überprüfen.

Sollte die Test Instanz von Nightscout einmal nicht verfügbar sein, schlagen die Tests mit eindeutigen Meldungen fehl und teilen dem Benutzer mit, wie die Fehler zu beheben sind. Wenn auf der Nightscout Test Instanz unzureichende oder falsche Daten gespeichert sind, überprüfen die Tests, ob die Algorithmen korrekt abbrechen und dem Benutzer mitteilen, welche Probleme mit den verfügbaren Daten bestehen. Nicht ausgeführte Tests auf Grund von fehlenden Daten in der Nightscout Test Instanz werden als nicht ausgeführt markiert, lassen den gesamten Test aber nicht fehlschlagen.

Coverity wird nur bei Commits auf den Coverity Branch ausgeführt, da die Nutzung von Coverity beschränkt ist. Dies wird durchgeführt, bevor in den Master Branch gemerged wird. Die Anzahl der Builds pro Woche ist auf 28 mit maximal 4 Tests am Tag beschränkt, wenn weniger als 100.000 Zeilen Code getestet werden. Wurden die maximale Anzahl der Builds pro Tag erreicht, werden weitere Builds an dem entsprechenden Tag abgelehnt. Wenn Travis CI oder Coverity Fehler erzeugen, müssen diese von demjenigen, der gepusht hat. Auch dies soll innerhalb der nächsten drei Tage geschehen und kann ebenfalls maximal auf sieben Tage

verlängert werden. Falls dies nicht gelöst werden kann wird das Problem auf dem nächsten Aufraggebertreffen besprochen.

## 2.2 Modularität

### 2.2.1 Beschreibung

Unser zweites QS-Ziel ist die Modularität und damit verbunden die Erweiterbarkeit. Unser Ziel ist es, ohne viel Aufwand neue Algorithmen zur Berechnung der Mahlzeiten einpflegen zu können, sowie die Möglichkeit zu bieten neue Daten und Datenquellen einbinden zu können.

Dies ist besonders sinnvoll, da unser Code einer Opensource-Community, unter der AGPLv3 Lizenz, auf GitHub zur Verfügung steht und Mitglieder der Nightscout-Community (einschließlich Herrn Heuschkel als Auftraggeber) den Code wieder verwenden möchten.

So ist es möglich, wenn neue Ansätze für eine bessere Berechnung von Mahlzeiten gefunden wurden, diese einfach zu implementieren und auszuführen, ohne dass das Hauptprogramm geändert werden muss. Genauso können neue Datenquellen, wie zum Beispiel eine MySQL Datenbank, oder ein entferntes Dateisystem, eingefügt werden.

### 2.2.2 Maßnahmen

Um dies möglich zu machen, werden zunächst GitHub Wiki Artikel zur Verfügung gestellt[4]. In diesen wird ausführlich erklärt, wie unsere Software zu installieren, verwenden und erweitern ist. Die Projektstruktur ist modular, sodass neue Algorithmen und Datenquellen einfach durch die Implementierung eines Interfaces hinzugefügt werden können. Es werden verschiedene Entwurfsmuster (Design Patterns) verwendet, um die Erweiterbarkeit sicher zu stellen.

Unser Projekt gliedert sich in drei Hauptmodule: Das Hauptprogramm (Modul 1), die Anbindung an Nightscout und der Daten-Parser (Modul 2) und die Repräsentation der Daten (Modul 3). Diese Module können unabhängig voneinander verwendet werden, wobei sie jeweils auf das in der obigen Liste folgende Modul aufbauen. Unser Auftraggeber verwendet zum Beispiel die im dritten Modul definierten Datenklassen auch für andere Projekte, und die Anbindung an Nightscout kann benutzt werden, um beliebige Daten von und zu Nightscout zu übertragen.

Das Hauptprogramm gliedert sich in vier weitere Module: Die Hauptklasse und ihre dazugehörigen Klassen, zum Starten des Programms (Modul 1.1), die Datenquellen, zum Einlesen und Abspeichern der Daten (Modul 1.2) und die Algorithmen, zur Verarbeitung der Daten (Modul 1.3). Häufig benutzte Funktionen und die Implementierungen der verwendeten mathematischen Modelle, welche in verschiedenen Algorithmen benötigt werden, werden in eigene Klassen ausgelagert (Modul 1.4). Hierunter fällt zum Beispiel das Berechnen von Insulinabbau im Blutkreislauf über Zeit oder die Umwandlung von Kohlenhydraten zu Glucose über Zeit, was unabhängig vom später verwendeten Algorithmus immer gleich verläuft. Dies ermöglicht es viele Funktionen in verschiedenen Algorithmen wieder zu verwenden und diese unabhängig von den Algorithmen testen zu können.

Wie bereits beim Ziel der Korrektheit beschrieben, werden Pull-Requests auf den Master Branch von mindestens einer anderen Person kontrolliert. Hier wird auch überprüft, ob die oben beschriebene Modularität eingehalten wurde.

[4] https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/wiki

### 2.2.3 Prozessbeschreibung

Die Wiki Artikel beschreiben auf englisch, Schritt für Schritt, wie ein neuer Algorithmus implementiert werden kann. Dies wird exemplarisch an einem unserer Algorithmen gezeigt. Die Artikel werden von zwei Projektmitgliedern geschrieben. Mindestens zwei andere Mitglieder lesen diese Korrektur und überprüfen anhand einer Checkliste (siehe Anhang), ob diese vollständig sind und vollziehen die Schritte auf einem neuen System nach. Dies ist erreicht, wenn der Algorithmus nach der Ausführung der Anleitung lauffähig ist.

Sowohl für die Algorithmen als auch für die Datenquellen verwenden wir das Strategy Pattern[5], um während der Laufzeit verschiedene Algorithmen und Datenquellen zu laden. Das Datenquellen-Interface (`DataProvider`) definiert abstrakte Methoden zum Laden von Blutzucker Messwerten, Insulinbehandlungen, und bekannten Mahlzeiten jeweils in einem bestimmten Zeitfenster. Das Algorithmus-Interface (`Algorithm`) definiert abstrakte Methoden, um bekannte Blutzucker Messwerte, Insulinbehandlungen und Mahlzeiten zur Verfügung zu stellen, eine Methode um die Ausführung des Algorithmus zu starten und eine Methode um die berechneten Mahlzeiten abzurufen.

Die Nightscout Anbindung beinhaltet einen unabhängig verwendbaren Parser, welcher ebenfalls ein Interface erweitert und mit dem Template Method Pattern das direkte Analysieren von Strings (zum Beispiel die Rückgabewerte der Nightscout API) und die Verwendung von normalen Textdateien als Quelle ermöglicht. Dieser Parser kann die JSON-Repräsentation der Daten von Nightscout übersetzen. Da Nightscout selbst das Hinzufügen von neuen Datentypen erlaubt, ist es auch hier kein Problem einen neuen Parser für noch unbekannte Daten hinzuzufügen.

Zur Repräsentation der Daten im Programm benutzen wir verschiedene vom Auftraggeber vorgegebene Datenklassen, welche auch in anderen Projekten des Auftraggebers verwendet werden. Alle Algorithmen und Datenquellen benutzen ausschließlich diese Klassen, um miteinander zu kommunizieren, und ermöglichen dadurch ebenso unser Programm mit anderen vom Auftraggeber verwendeten Programmen zu kombinieren.

Bei der Kontrolle von Pull-Requests auf den Master Branch achtet die kontrollierende Person[6] darauf, dass die verwendeten Entwurfsmuster korrekt umgesetzt wurden und zum Beispiel ein Algorithmus nicht direkt Daten an das Hauptprogramm weiter gibt. Außerdem wird darauf geachtet neue Funktionen so abstrakt wie möglich zu implementieren, um die Wiederverwendung an anderen Stellen zu erleichtern. Dabei wird kontrolliert, ob diese neuen Funktionen sinnvoll in ihrem Modul platziert sind, oder ob eine Implementation in einem anderen Modul die Wiederverwendung an mehr Stellen ermöglichen würde. Die im Anhang enthaltene Checkliste beinhaltet auch die hierfür wichtigen Punkte. Sollte ein Pull-Request abgelehnt worden sein, hat der Ersteller des Pull-Requests wieder drei Tage Zeit dies zu beheben und kann maximal auf sieben Tage verlängern. Auch hier gilt, dass das Problem bei dem nächsten Auftraggebertreffen besprochen wird, falls es in der Zeit nicht gelöst wird. Wer den Pull-Request angenommen hat wird automatisch in git dokumentiert. Auch hier wird die entsprechende Dokumentation im Anhang zur Verfügung gestellt.

[5] https://en.wikipedia.org/wiki/Strategy_pattern

[6] Die kontrollierende Person ist immer eine andere, als die Person, welche einen Pull-Request erstellt hat. Dies wird von GitHub sicher gestellt, der Ersteller eines Pull-Requests ist automatisch nicht mehr in der Lage, den Pull-Request anzunehmen.

# A Anhang

## A.1 Pull-Request-Review Checkliste

Die aktuellste Version der Checkliste ist auch im GitHub Wiki einsehbar.

### General

- Travis CI Build successfull
- Coverity Scan executed *(Hinzugefügt ab PR #15)*
- Scope is appropriate *(Hinzugefügt ab PR #13)*
  - if more than one feature is merged by this pull request, they have to be logically connected
- Module is appropriate
  - the changes are applied in the lowest possible place on the hierarchy list of modules
- Access is appropriate *(Hinzugefügt ab PR #13)*
  - all fields, methods and classes have the most restrictive access possible (private < protected < package < public)
  - Documentation is complete
  - all methods that are not private have appropriate JavaDoc
  - private methods should be documented if the method name does not immediately explain the methods purpose
  - getters and setters are excluded if the only thing they do is getting or setting an objects field
- Modularity is kept
  - no cyclic references between modules are introduced
  - design patterns are applied (see below)
    - main module: Strategy Pattern for algorithms and dataproviders
    - nsapi module: Template Method Pattern for parsers
- Style and Layout
  - package and class names make sense, method names are self explanatory but short
  - easy to read and understand

**Tests**

- All implemented tests are executed by travis
  - a test that is known to be broken can be excluded from execution only, if another test tests the same feature
  - broken tests should be documented for why they are still needed (fix may be coming later, or they are kept for reference)
- All public methods are tested
  - all methods, that are not getters and setters, have to be tested
  - exceptions to this rule have to be explained in detail
- Methods which have no restrictions on their arguments are tested with all possible arguments *(Hinzugefügt ab PR #13)*
  - if an exhaustive test of all arguments is not possible, the arguments are randomized over the entire range of possible arguments
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments *(Hinzugefügt ab PR #13)*

**Design Patterns**

*(Hinzugefügt ab PR #13)*

**Strategy Pattern:**

- The two strategy interfaces are:
  - `Algorithm`
    - This interface was changed to an abstract class during development. The strategy pattern still applies.
  - `DataProvider`
- The strategy implementations exclusively use their interface methods for communication with other classes
- The main program does not execute methods on conrete implementations of the strategy but only methods of the strategy interfaces (with the exception of the constructors)

**Template Method Pattern:**

- The template interfaces are:
  - `Parser<T>`
    - method: `T parse(String)`

## A.2 Pull-Request-Review Protokoll

Die folgenden Pull-Request-Review Protokolle wurden während dem Projekt erstellt. Es konnten mehrfach Pull-Requests nicht direkt angenommen werden, allerdings wurden die Probleme in allen Fällen direkt während dem Review Prozess gelöst da es sich jedes mal nur um fehlende Tests oder fehlendes Javadoc handelte. „Main" Klassen wurden manuell auf ihre funktionalen Anforderungen getestet, alle anderen („Library"-) Klassen wurden durch JUnit Tests getestet.

**05.01.2019 PR#3**
- Protokoll: Bild A.1

**11.01.2019 PR#4**
- Protokoll: Bild A.2

**04.03.2019 PR#13**
- Protokoll: Bild A.3
- Test Coverage: Bild A.4

**13.03.2019 PR#15**
- Protokoll: Bild A.5
- Test Coverage: Bild A.6
- Manuelle Tests: Bild A.7

**15.03.3019 PR#16**
- Protokoll: Bild A.8
- Test Coverage: Bild A.9
- Manuelle Tests: Bild A.10

**18.03.3019 PR#18**
- Protokoll: Bild A.11
- Test Coverage: Bild A.12

**23.03.2019 PR#45**
- Protokoll: Bild A.13

**23.03.3019 PR#43**
- Protokoll: Bild A.14
- Test Coverage: Bild A.15
- Manuelle Tests: Bild A.16

**23.03.2019 PR#46**
- Protokoll: Bild A.17

**29.03.3019 PR#52**

- Protokoll: Bild A.18

**29.03.3019 PR#47**

- Protokoll: Bild A.19
- Test Coverage: Bild A.20

**30.03.3019 PR#51**

- Protokoll: Bild A.21
- Test Coverage: Bild A.22

**Abbildung A.1.:** Pull Request #3

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/3

created by: Jan Petto
approved by: Anna Mees
merged on: 05.01.2019

## General

- Travis CI Build successfull
  - check
- Module is appropriate
  - check
- Documentation is complete
  - check
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check

**Abbildung A.2.:** Pull Request #4

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/4

created by: Paul Wolfart
approved by: Anna Mees
merged on: 11.01.2019

## General

- Travis CI Build successfull
  - check
- Module is appropriate
  - check
- Documentation is complete
  - check
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check

**Abbildung A.3.:** Pull Request #13

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/13

created by: Jan Petto
approved by: Aino Schwarte
merged on: 04.03.2019

## General

- Travis CI Build successfull
  - check
- Coverity Scan executed
  - Nicht ausgeführt
- Scope is appropriate
  - Implementation von durch Auftraggeber vorgegebener Libraries
    - Import Änderungen wegen geänderter Interfaces
    - Hilfsfunktionen
    - Vault Entry Export
- Module is appropriate
  - Durch Auftraggeber vorgegebene Änderungen in Vault
    - niedrigstes Modul
  - Nightscout Api Änderungen im NSApi Modul
- Access is appropriate
  - check
- Documentation is complete
  - Durch Auftraggeber vorgegebene Änderungen bleiben undokumentiert
  - Debugging Methode nicht dokumentiert (fliegt später raus)
  - Parser Interface
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check
- Methods which have no restrictions on their arguments are tested with all possible arguments
  - check
  - NSApi: Tests sind noch nicht erschöpfend, das ist außerhalb des Scopes dieses PRs, wird in anderem Branch entwickelt
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments
  - check
  - NSApi: Siehe oben, Tests werden in anderem Branch entwickelt

## Design Patterns

### Strategy Pattern

- Keine Änderungen an Komponenten mit Strategy Pattern

### Template Method Pattern

- Parser: check

**Abbildung A.4.:** Test Coverage Pull Request #13

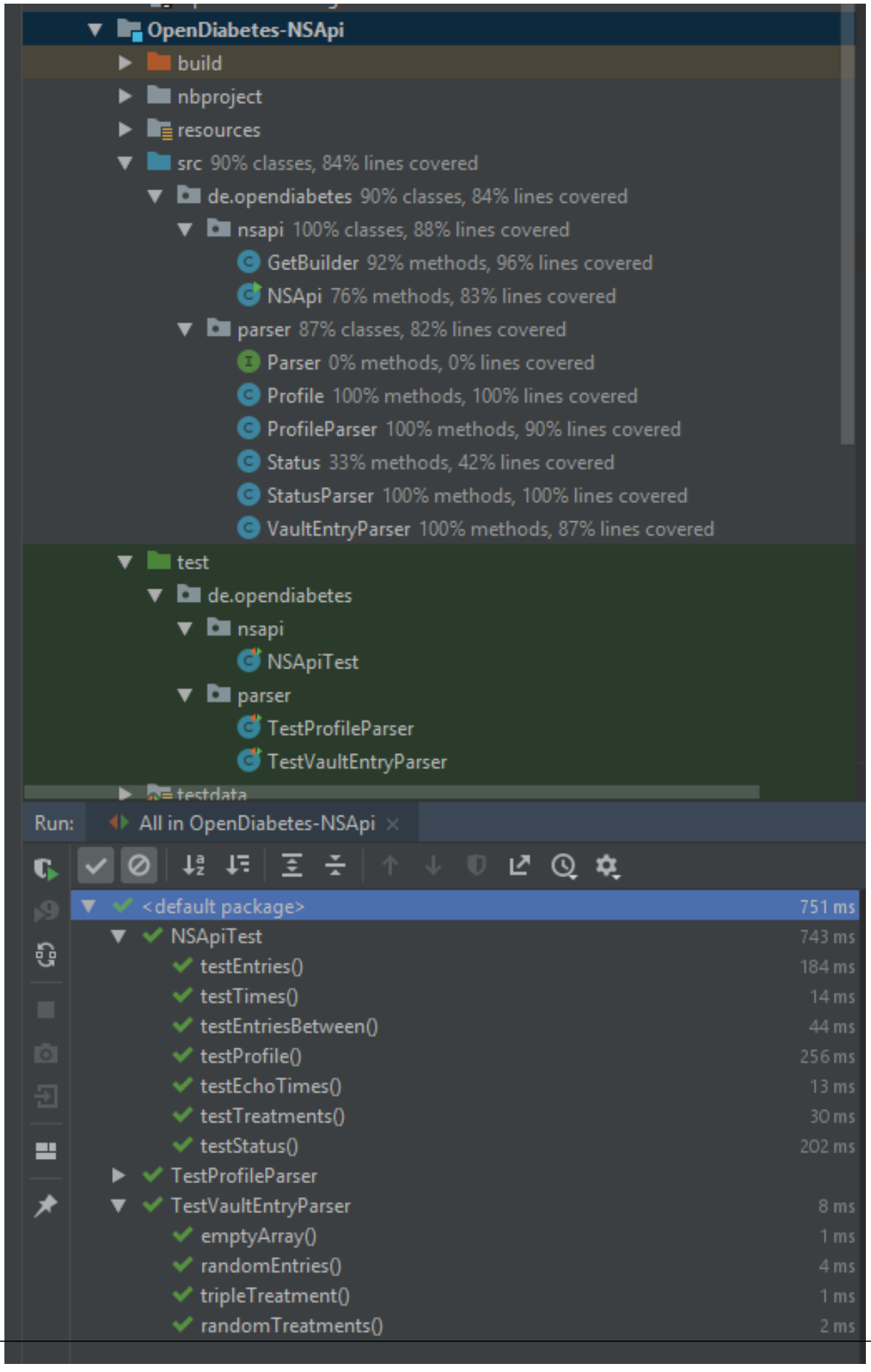


**Abbildung A.5.:** Pull Request #15

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/15

created by: Jan Petto
approved by: Paul Wolfart
merged on: 13.03.2019

## General

- Travis CI Build successfull
  - check
- Coverity Scan executed
  - check, no defects in merged components
- Scope is appropriate
  - check
- Module is appropriate
  - check
- Access is appropriate
  - check
- Documentation is complete
  - check
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check
  - report on the manual test of the main class
- Methods which have no restrictions on their arguments are tested with all possible arguments
  - check
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments
  - check

## Design Patterns

## Strategy Pattern

- Keine Änderungen an Komponenten mit Strategy Pattern

## Template Method Pattern

- Parser: check

**Abbildung A.6.:** Test Coverage Pull Request #15

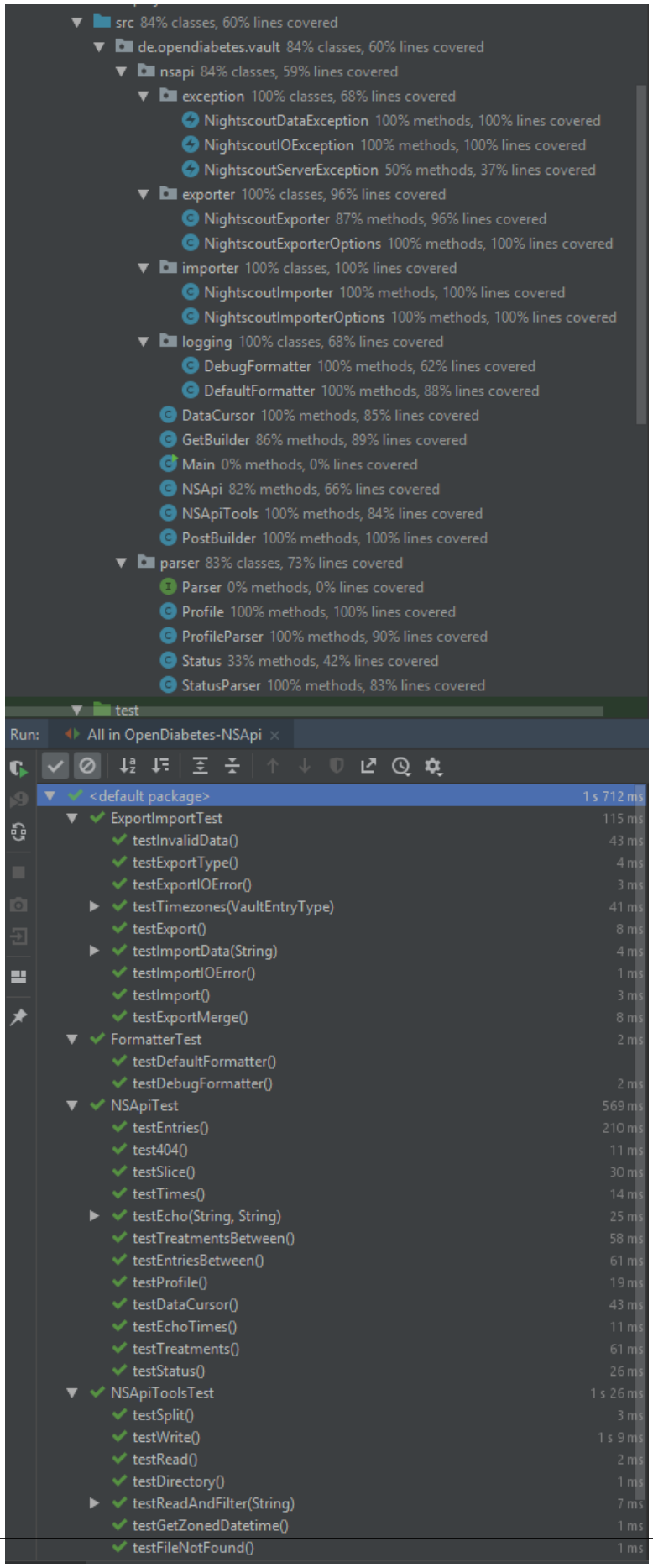


**Abbildung A.7.:** Manual Tests Pull Request #15

Tester: Jan Petto
Tested on: 13.03.2019

## Ausführung Main

| | |
|---|---|
| Ohne Argumente gibt Erklärung aller möglichen Argumente aus | check |
| Ausführung mit nicht allen benötigten Argumenten beendet und gibt Warnung aus | check |
| Ausführung mit falschen Secret gibt Warnung aus<br>(zu beachten: Nightscout Server muss mit Authentifizierung konfiguriert sein) | check |
| Auführung mit --status gibt Status aus | check |
| Debug Flag gibt bei Fehlern Stacktrace mit aus | check |
| Verbose Flag gibt detailliertere Logmessages | check |
| Post File funktioniert | check |
| Get File funktioniert | check |
| Filtern von Daten funktioniert | check |

**Abbildung A.8.:** Pull Request #16

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/16

created by: Jan Petto
approved by: Paul Wolfart
merged on: 15.03.2019

## General

- Travis CI Build successfull
  - check
- Coverity Scan executed
  - check
- Scope is appropriate
  - check
- Module is appropriate
  - check
- Access is appropriate
  - check
- Documentation is complete
  - check
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check
  - report on the manual test of the main class
- Methods which have no restrictions on their arguments are tested with all possible arguments
  - check
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments
  - check

## Design Patterns

~~no changes concering design patterns~~

**Abbildung A.9.:** Test Coverage Pull Request #16

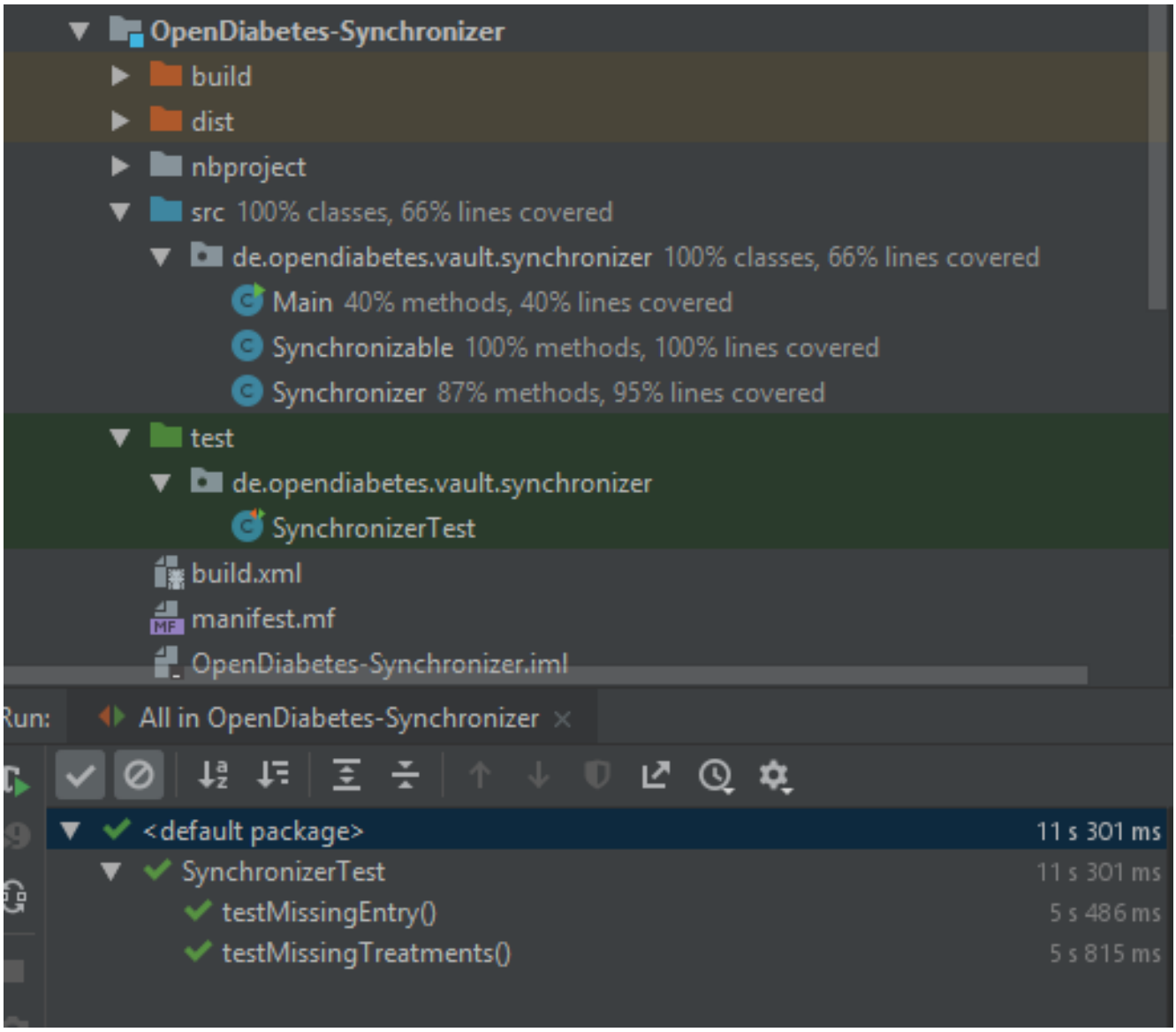


**Abbildung A.10.:** Manual Tests Pull Request #16

Tester: Jan Petto
Tested on: 15.03.2019

## Ausführung Main

| | |
|---|---|
| Ohne Argumente gibt Erklärung aller möglichen Argumente aus | check |
| Ausführung mit nicht allen benötigten Argumenten beendet und gibt Warnung aus | check |
| Ausführung mit falschen Secret gibt Warnung aus<br>(zu beachten: Nightscout Server muss mit Authentifizierung konfiguriert sein) | check |
| Auführung ohne weitere Argumente synchronisiert entries, treatments und device status | check |
| Debug Flag gibt bei Fehlern Stacktrace mit aus | check |
| Verbose Flag gibt detailliertere Logmessages | check |
| --data flag erlaubt andere Datenpunkte (Test mit uam Datenpunkt) | check |
| IDs werden von Daten entfernt außer --with-ids Flag ist gesetzt | check |

**Abbildung A.11.:** Pull Request #18

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/18

created by: Paul Wolfart
approved by: Jan Petto
merged on: 18.03.2019

## General

- Travis CI Build successfull
  - check
- Coverity Scan executed
  - check (from previous build, no new changes)
- Scope is appropriate
  - check
- Module is appropriate
  - check
- Access is appropriate
  - check
- Documentation is complete
  - check
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check
- Methods which have no restrictions on their arguments are tested with all possible arguments
  - check, except timezone tests:
    - tests are for edge cases which would be contradicted by random arguments
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments
  - check

## Design Patterns

No changes concerning any design patterns

**Abbildung A.12.:** Test Coverage Pull Request #18

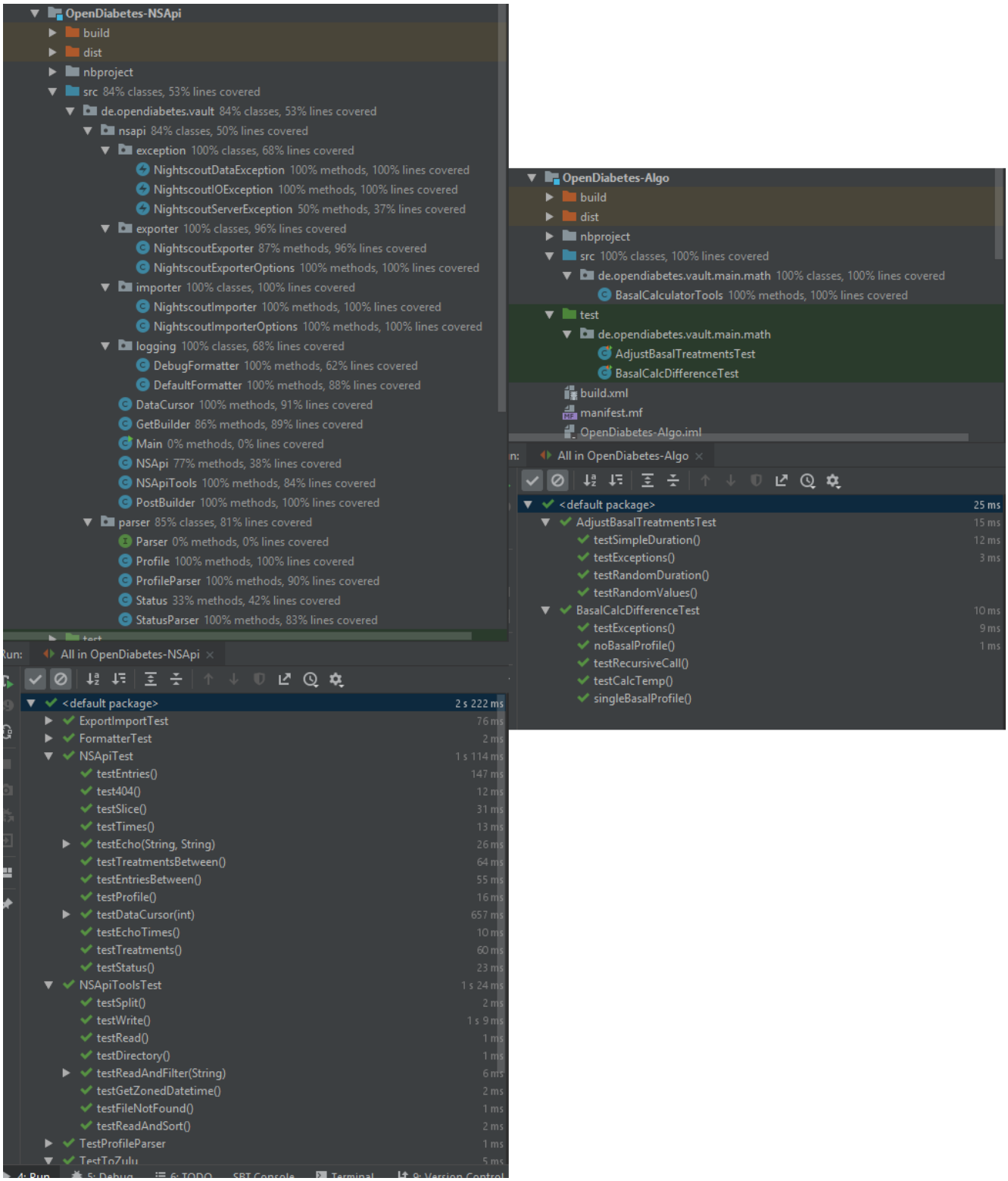


**Abbildung A.13.:** Pull Request #45

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/43

created by: Jan Petto
approved by: Anna Mees
merged on: 23.03.2019

## General

- Travis CI Build successful
  - check
- Coverity Scan executed
  - check
- Scope is appropriate
  - check
- Module is appropriate
  - check
- Access is appropriate
  - check
- Documentation is complete
  - check
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check
  - report on the manual test of the main class
- Methods which have no restrictions on their arguments are tested with all possible arguments
  - check
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments
  - check

## Design Patterns

no changes in this area

**Abbildung A.14.:** Pull Request #43

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/43

created by: Jan Petto
approved by: Anna Mees
merged on: 23.03.2019

## General

- Travis CI Build successful
  - check
- Coverity Scan executed
  - check
- Scope is appropriate
  - check
- Module is appropriate
  - check
- Access is appropriate
  - check
- Documentation is complete
  - check
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check
  - report on the manual test of the main class
- Methods which have no restrictions on their arguments are tested with all possible arguments
  - check
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments
  - check

## Design Patterns

no changes concerning design patterns

**Abbildung A.15.:** Test Coverage Pull Request #43

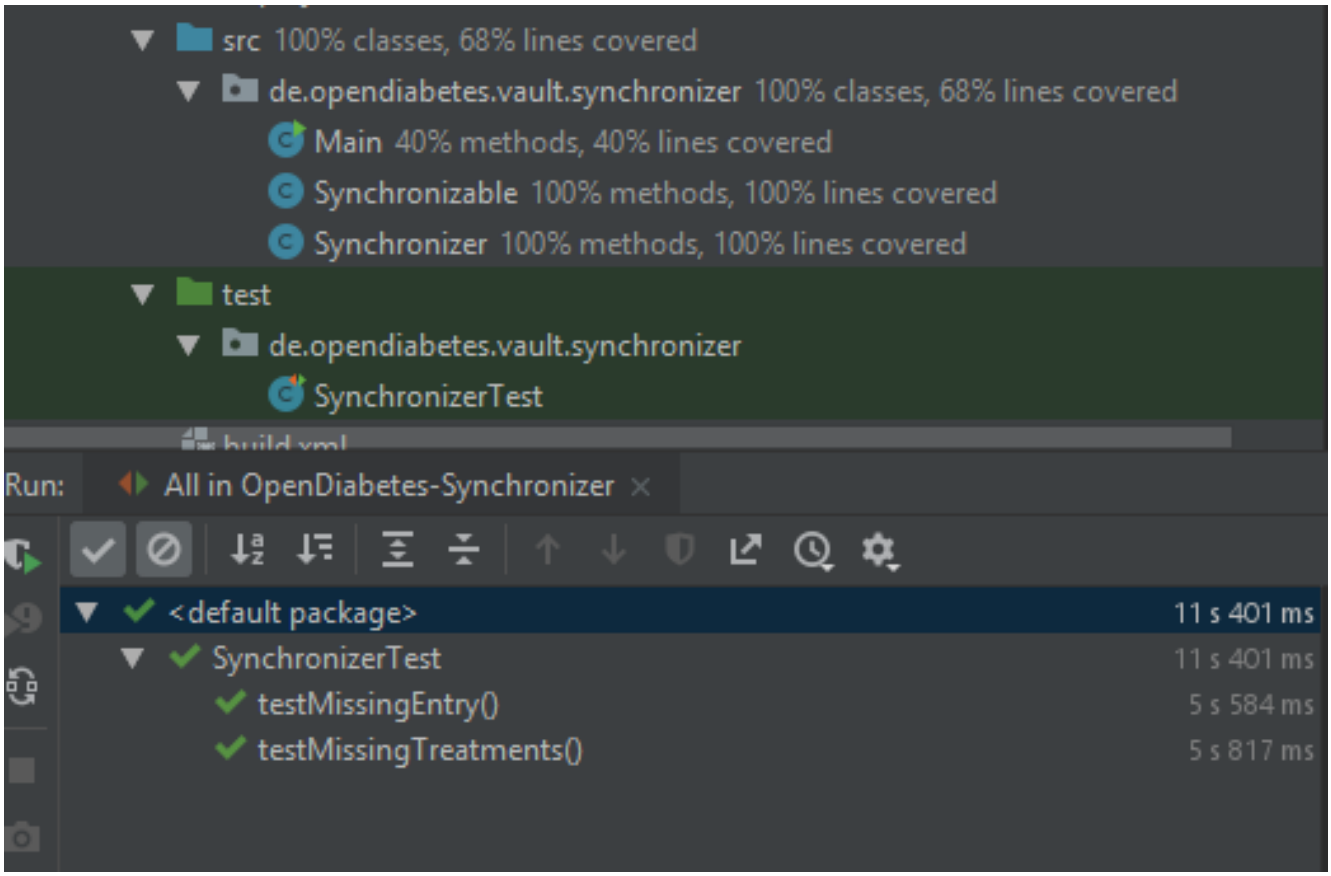


**Abbildung A.16.:** Manual Tests Pull Request #43

Tester: Jan Petto
Tested on: 23.03.2019

## Ausführung Main

| | |
|---|---|
| Ohne Argumente gibt Erklärung aller möglichen Argumente aus | check |
| Ausführung mit nicht allen benötigten Argumenten beendet und gibt Warnung aus | check |
| Ausführung mit falschen Secret gibt Warnung aus<br>(zu beachten: Nightscout Server muss mit Authentifizierung konfiguriert sein) | check |
| Auführung ohne weitere Argumente synchronisiert entries, treatments und device status | check |
| Debug Flag gibt bei Fehlern Stacktrace mit aus | check |
| Verbose Flag gibt detailliertere Logmessages | check |
| --data Flag erlaubt andere Datenpunkte (Test mit uam Datenpunkt) | check |
| IDs werden von Daten entfernt außer --with-ids Flag ist gesetzt | check |
| -diff Flag schreibt Fehlende Daten in Datei anstatt sie auf Zielinstanz hochzuladen | check |

**Abbildung A.17.:** Pull Request #46

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/43

created by: Jan Petto
approved by: Anna Mees
merged on: 23.03.2019

## WIP General

- Travis CI Build successful
  - check
- Coverity Scan executed
  - check
- Scope is appropriate
  - check
- Module is appropriate
  - check
- Access is appropriate
  - check
- Documentation is complete
  - check
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check
  - report on the manual test of the main class
- Methods which have no restrictions on their arguments are tested with all possible arguments
  - check
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments
  - check

## Design Patterns

no changes in this area

**Abbildung A.18.:** Pull Request #52

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/52

Geöffnet von: Jan Petto
Reviewed von: Paul Wolfart
Merged on: 29.03.2019

Mit diesem Pullrequest wurden nur Dateien verändert die den Buildprozess von Travis verändern.

**Abbildung A.19.:** Pull Request #47

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/47
Geöffnet von: Jan Petto
Reviewed von: Paul Wolfart
merged am: 29.03.2019

## General

- Travis CI Build successfull
  - was successfull
- Coverity Scan executed
  - was executed
- Scope is appropriate
  - check
- Module is appropriate
  - check
- Access is appropriate
  - check
- Documentation is complete
  - Alle änderungen wurden dokumentiert
- Modularity is kept
  - keine zyklischen Referenzen
- Style and Layout
  - check

## Tests

- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check
- Methods which have no restrictions on their arguments are tested with all possible arguments
  - check
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments
  - check

## Design Patterns

no changes in this area

**Abbildung A.20.:** Test Coverage Pull Request #47

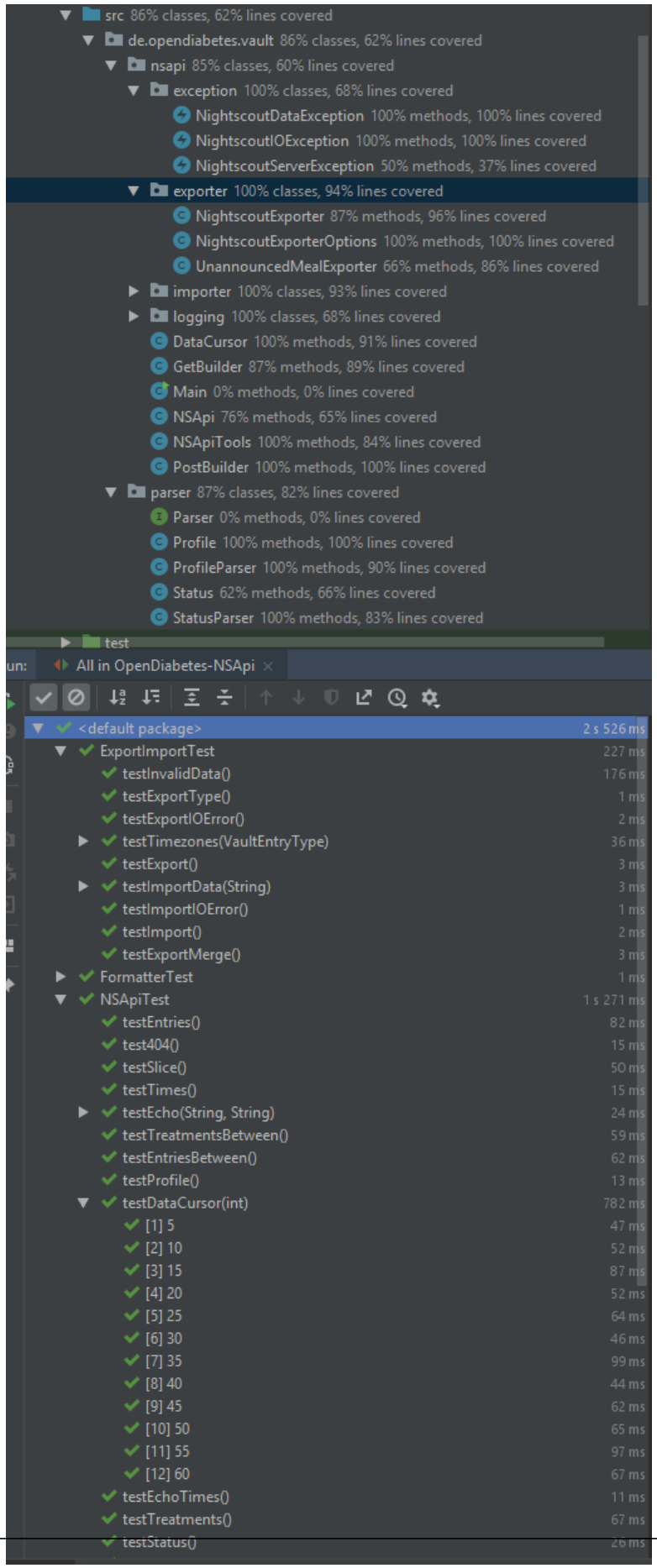


**Abbildung A.21.:** Pull Request #51

https://github.com/TUDa-BP-11/opendiabetes-uam-heuristik/pull/51

Created by: Anna Mees
Approved by: Jan Petto
Merged on: 30.03.2019

Anfangs fehlten einige Tests und die Dokumentation war unvollständig. Dies wurde behoben.

## General

- Travis CI Build successfull
  - check
- Coverity Scan executed
  - check
- Scope is appropriate
  - check
- Module is appropriate
  - check
- Access is appropriate
  - check
- Documentation is complete
  - check
- Modularity is kept
  - check
- Style and Layout
  - check

## Tests

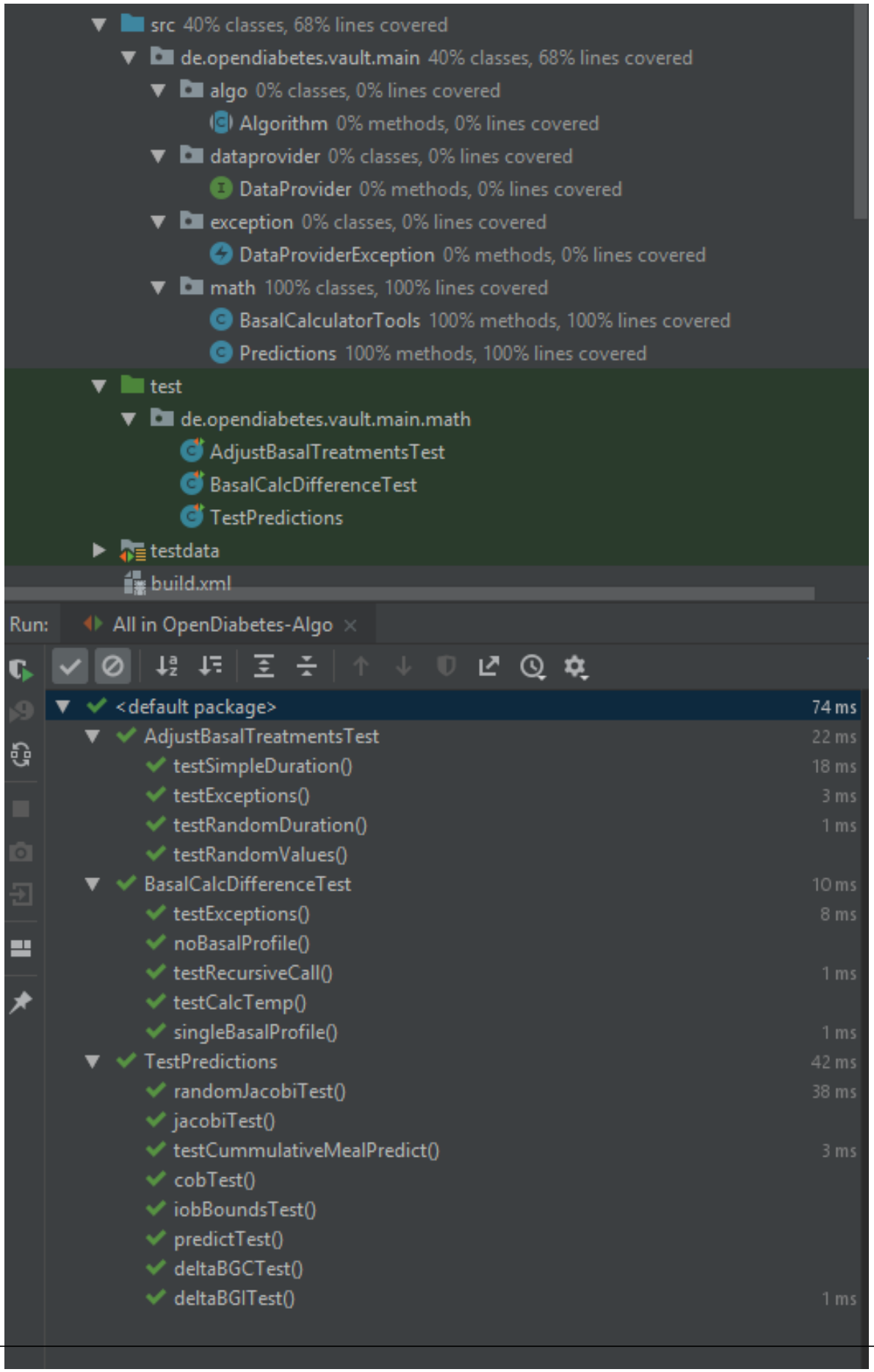
- All implemented tests are executed by travis
  - check
- All public methods are tested
  - check
  - Die abstrakten Algorithm Klasse ist ungetestet. Tests hierfür werden in einem zukünftigen Merge Request mit den Implementationen der Klasse zusammen getestet.
- Methods which have no restrictions on their arguments are tested with all possible arguments
  - check
- Methods which have restrictions on their arguments are tested for appropriate error messages or exceptions if they are executed with invalid arguments
  - check

## Design Patterns

## Strategy Pattern

- The strategy implementations exclusively use their interface methods for communication with other classes
  - check
- The main program does not execute methods on conrete implementations of the strategy but only methods of the strategy interfaces (with the exception of the constructors)
  - check (Main program not yet included)

**Abbildung A.22.:** Test Coverage Pull Request #51



## A.3 GitHub Wiki Artikel Checkliste

TODO

## A.4 Code Beispiele

### A.4.1 DataCursor

Der `DataCursor` kann benutzt werden um durch Nightscout Daten zu iterieren. Der Cursor lädt die Daten dafür in einen internen Puffer, der automatisch aufgefüllt wird. Der Nightscout Server gibt Daten immer in absteigender Reihenfolge sortiert nach Datum zurück, weswegen der `DataCursor` von einem `latest` Zeitpunkt absteigend zu einem `oldest` Zeitpunkt durch die Daten iteriert. Alle Nightscout API Pfade, welche bei einem HTTP GET Request ein Array an Daten zurück geben, können benutzt werden, solange die zurückgegebenen Daten ein Datumsfeld beinhalten.

DataCursor.java

```
1  package de.opendiabetes.vault.nsapi;
2
3  import com.google.gson.JsonArray;
4  import com.google.gson.JsonElement;
5  import com.google.gson.JsonObject;
6  import de.opendiabetes.vault.nsapi.exception.NightscoutIOException;
7  import de.opendiabetes.vault.nsapi.exception.NightscoutServerException;
8
9  import java.time.format.DateTimeFormatter;
10 import java.time.temporal.TemporalAccessor;
11 import java.util.Iterator;
12 import java.util.concurrent.LinkedBlockingQueue;
13 import java.util.logging.Level;
14
15 import static de.opendiabetes.vault.nsapi.NSApi.LOGGER;
16
17 /**
18  * A buffer for Nightscout data. Lazily loads objects from the server as needed.
      The internal buffer is refreshed
19  * if it runs out of objects until the server does not return any more data.
20  */
21 public class DataCursor implements Iterator<JsonElement> {
22     private final NSApi api;
23     private final String path;
24     private final String dateField;
25     private final String oldest;
26     private final int batchSize;
27
28     private final LinkedBlockingQueue<JsonObject> buffer;
29     private String current;
30     private boolean first;
31     private boolean finished;
32
33     /**
```

```
34          * Creates a new cursor. Latest and oldest point in time are formatted using
                the {@link NSApi#DATETIME_PATTERN_ENTRY} pattern.
35          *
36          * @param api        Nightscout API instance
37          * @param path       API path used with {@link NSApi#createGet(String)}. Has
                to return an array of json objects.
38          * @param dateField the name of the field which holds information about the
                date and time of your data object
39          * @param latest     latest point in time to load data for
40          * @param oldest     oldest point in time to load data for
41          * @param batchSize amount of entries which will be loaded at once
42          */
43         public DataCursor(NSApi api, String path, String dateField, TemporalAccessor
                latest, TemporalAccessor oldest, int batchSize) {
44             this(api, path, dateField, latest, oldest, batchSize,
                    NSApi.DATETIME_FORMATTER_ENTRY);
45         }
46
47         /**
48          * Creates a new cursor.
49          *
50          * @param api        Nightscout API instance
51          * @param path       API path used with {@link NSApi#createGet(String)}. Has
                to return an array of json objects.
52          * @param dateField the name of the field which holds information about the
                date and time of your data object
53          * @param latest     latest point in time to load data for
54          * @param oldest     oldest point in time to load data for
55          * @param batchSize amount of entries which will be loaded at once
56          * @param formatter DateTimeFormatter used to format latest and oldest point
                in time
57          */
58         public DataCursor(NSApi api, String path, String dateField, TemporalAccessor
                latest, TemporalAccessor oldest, int batchSize, DateTimeFormatter
                formatter) {
59             this.api = api;
60             this.path = path;
61             this.dateField = dateField;
62             this.oldest = formatter.format(oldest);
63             this.batchSize = batchSize;
64
65             this.buffer = new LinkedBlockingQueue<>(batchSize);
66             this.current = formatter.format(latest);
67             this.first = true;
68             this.finished = false;
69         }
70
71         /**
72          * Checks if there is more data. May block the thread to request new data
                from the Nightscout server.
73          * Logs exceptions using {@link NSApi#LOGGER}.
74          *
75          * @return true if there is more data
76          */
77         @Override
```

```
78      public boolean hasNext() {
79          if (buffer.size() > 0)
80              return true;
81          if (finished)
82              return false;
83          try {
84              GetBuilder builder = api.createGet(path);
85              // use lte on first fetch, lt for all remaining requests.
86              if (first) {
87                  builder.find(dateField).lte(current);
88                  first = false;
89              } else builder.find(dateField).lt(current);
90              // finish request
91              JsonArray array = builder
92                      .find(dateField).gte(oldest)
93                      .count(batchSize)
94                      .getRaw().getAsJsonArray();
95              array.forEach(e -> buffer.offer(e.getAsJsonObject()));
96              // we are done if the returned array has less data then the batch size
97              if (array.size() < batchSize)
98                  finished = true;
99              else current = array.get(array.size() -
                    1).getAsJsonObject().get(dateField).getAsString();
100
101             return array.size() > 0;
102         } catch (NightscoutIOException | NightscoutServerException |
                IllegalStateException e) {
103             LOGGER.log(Level.SEVERE, e, e::getMessage);
104             return false;
105         }
106     }
107
108     /**
109      * Gets the next object of the current buffer. Note that this method will
             never block, but may return null if not
110      * used in conjunction with {@link this#hasNext()} as the buffer will only be
             refreshed by that method.
111      *
112      * @return the next entry parsed as some kind of json element.
113      */
114     @Override
115     public JsonObject next() {
116         return buffer.poll();
117     }
118 }
```

### A.4.2 NightscoutImporter

Der `NightscoutImporter` importiert Nightscout Daten und erzeugt eine Liste von `VaultEntry` Objekten. Er implementiert dabei die vom Auftraggeber vorgegebene `Importer` Klasse.

NightscoutImporter.java

```
1 package de.opendiabetes.vault.nsapi.importer;
```

```
2
3  import com.google.gson.JsonElement;
4  import com.google.gson.JsonObject;
5  import com.google.gson.JsonParseException;
6  import com.google.gson.JsonParser;
7  import de.opendiabetes.vault.container.VaultEntry;
8  import de.opendiabetes.vault.container.VaultEntryType;
9  import de.opendiabetes.vault.importer.Importer;
10 import de.opendiabetes.vault.nsapi.NSApi;
11 import de.opendiabetes.vault.nsapi.exception.NightscoutDataException;
12 import de.opendiabetes.vault.util.TimestampUtils;
13
14 import java.io.InputStream;
15 import java.io.InputStreamReader;
16 import java.io.Reader;
17 import java.time.ZonedDateTime;
18 import java.time.format.DateTimeFormatter;
19 import java.time.format.DateTimeParseException;
20 import java.time.temporal.TemporalAccessor;
21 import java.util.ArrayList;
22 import java.util.Date;
23 import java.util.List;
24 import java.util.logging.Level;
25
26 /**
27  * Imports JSON objects from a Nightscout server as {@link VaultEntry}s.
28  */
29 public class NightscoutImporter extends Importer {
30     private final NightscoutImporterOptions options;
31     private final JsonParser json;
32
33     public NightscoutImporter() {
34         this(new NightscoutImporterOptions());
35     }
36
37     public NightscoutImporter(NightscoutImporterOptions options) {
38         super(options);
39         this.options = options;
40         this.json = new JsonParser();
41     }
42
43     /**
44      * Parses the source as a nightscout json representation of vault entries.
          Expects an array of json objects.
45      * If an object in the array contains information for multiple vault entry
          types, it will be split up into individual entries.
46      * Currently supports the following entries: {@link
          VaultEntryType#GLUCOSE_CGM}, {@link VaultEntryType#BOLUS_NORMAL},
47      * {@link VaultEntryType#MEAL_MANUAL} and {@link VaultEntryType#BASAL_MANUAL}.
48      *
49      * @param source Data source.
50      * @return list of generated vault entries. May contain more entries than the
          source did.
51      * @throws NightscoutDataException if the given source is not formatted
          correctly
```

```
52      */
53     @Override
54     public List<VaultEntry> importData(InputStream source) {
55         Reader reader = new InputStreamReader(source);
56         JsonElement element;
57         try {
58             element = json.parse(reader);
59         } catch (JsonParseException e) {
60             throw new NightscoutDataException("exception while reading data", e);
61         }
62 
63         if (!element.isJsonArray())
64             throw new NightscoutDataException("source is not an array");
65 
66         List<VaultEntry> entries = new ArrayList<>();
67         try {
68             for (JsonElement e : element.getAsJsonArray()) {
69                 JsonObject o = e.getAsJsonObject();
70                 boolean valid = false;
71 
72                 // BG measurements
73                 if (o.has("type") && o.get("type").getAsString().equals("sgv")) {
74                     Date date = TimestampUtils.createCleanTimestamp(new
                            Date(o.get("date").getAsLong()));
75                     entries.add(new VaultEntry(VaultEntryType.GLUCOSE_CGM, date,
                            o.get("sgv").getAsDouble()));
76                     valid = true;
77                 }
78 
79                 // insulin bolus
80                 if (o.has("insulin") && !o.get("insulin").isJsonNull()) {
81                     Date date = makeDate(o.get("timestamp").getAsString());
82                     entries.add(new VaultEntry(VaultEntryType.BOLUS_NORMAL, date,
                            o.get("insulin").getAsDouble()));
83                     valid = true;
84                 }
85 
86                 // meals
87                 if (o.has("carbs") && !o.get("carbs").isJsonNull()) {
88                     Date date = makeDate(o.get("timestamp").getAsString());
89                     entries.add(new VaultEntry(VaultEntryType.MEAL_MANUAL, date,
                            o.get("carbs").getAsDouble()));
90                     valid = true;
91                 }
92 
93                 // basal
94                 if (o.has("eventType") &&
                        o.get("eventType").getAsString().equals("Temp Basal")) {
95                     Date date = makeDate(o.get("timestamp").getAsString());
96                     entries.add(new VaultEntry(VaultEntryType.BASAL_MANUAL, date,
                            o.get("rate").getAsDouble(),
                            o.get("duration").getAsDouble()));
97                     valid = true;
98                 }
99                 if (!valid) {
```

```
100                     if (options.requireValidData())
101                         throw new NightscoutDataException("invalid source data,
                               could not identify vault entry type");
102                     else NSApi.LOGGER.log(Level.WARNING, "Could not parse JSON
                           Object: " + o.toString());
103                 }
104             }
105         } catch (NumberFormatException | IllegalStateException |
                NullPointerException e) {
106             throw new NightscoutDataException("invalid source data", e);
107         }
108         return entries;
109     }
110
111
112     /**
113      * Converts a date and time string to a {@link Date} object in local time.
             Strips seconds and milliseconds
114      *
115      * @param dateString ISO 8601 compliant date time string
116      * @return local Date object representing the input with seconds and
             milliseconds set to zero
117      */
118     private Date makeDate(String dateString) {
119         TemporalAccessor t;
120         try {
121             t = DateTimeFormatter.ISO_DATE_TIME.parse(dateString);
122         } catch (DateTimeParseException e) {
123             throw new NightscoutDataException("invalid date string: " +
                    dateString, e);
124         }
125         // dates are automatically converted to local time by the toInstant()
                method of ZonedDateTime
126         Date date = Date.from(ZonedDateTime.from(t).toInstant());
127         return TimestampUtils.createCleanTimestamp(date);
128     }
129 }
```

## A.5 Coverity Scan

Die Konfiguration von Coverity Scan erwies sich als schwierig, denn es können zwar Pfade angeben werden, welche nicht getestet werden sollen, Änderungen an den Einstellungen lassen sich aber nicht speichern. In diversen Internetforen lässt sich beobachten, dass auch andere Benutzer das gleiche Problem haben, weswegen wir dazu übergegangen sind die Defekte aus dem Code unseres Auftragebers als „absichtlich" zu markieren. Probleme in den Tests konnten in allen Fällen als „falsch-positiv" oder „absichtlich" markiert werden.
Die häufigsten „falsch-positiven" Ereignisse waren String to Byte bzw. Byte to String Konvertierungen. Die meisten Probleme die gefunden wurden waren Variablen und Felder die nicht benutzt oder nur aktualisiert wurden, danach waren es Felder und Methoden die nicht als statisch markiert wurden obwohl sie so benutzt wurden. Diese Probleme wurden behoben.

### A.5.1 Beispiel gefundener Bug

In Bild A.23 gibt die Funktion `getRawBasalTreatments()` null zurück und löst damit eine NullPointerException aus.


**Abbildung A.23.:** NullPointerException Beispiel A.5.1

### A.5.2 Beispiel False Positive

In Bild A.24 wird `!allFilesSet(config)` zu false ausgewertet.
In Bild A.25 wird das Inverse, also `allFilesSet(config)` auch zu false ausgewertet.

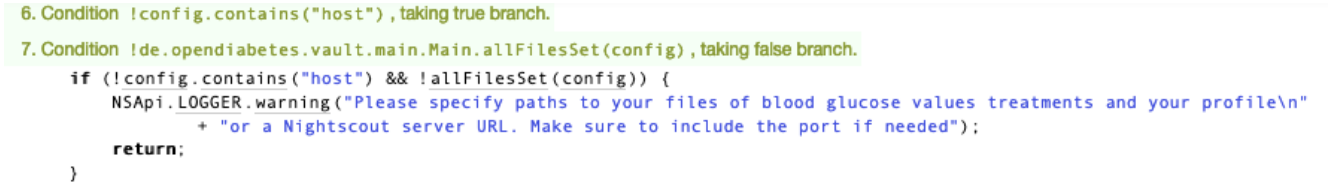



**Abbildung A.24.:** False Positive Beispiel A.5.2

```
private static AlgorithmDataProvider chooseDataProvider(JSAPResult config) {
1. Condition config.contains("host"), taking false branch.
    if (config.contains("host")) {
        return new NightscoutDataProvider(config.getString("host"), config.getString("secret"),
                config.getInt("batchsize"), (ZonedDateTime) config.getObject("latest"),
                (ZonedDateTime) config.getObject("oldest"));
    }
2. Condition de.opendiabetes.vault.main.Main.allFilesSet(config), taking false branch.
    if (allFilesSet(config)) {
        return new FileDataProvider(null, config.getString("entries"), config.getString("treatments"),
                config.getString("profile"), (ZonedDateTime) config.getObject("latest"),
                (ZonedDateTime) config.getObject("oldest"));
    }
3. return_null: Explicitly returning null.
    return null;
```

**Abbildung A.25.:** False Positive Beispiel A.5.2