Dokumentation

für die DHBW Software Systems gGmbH

Entwicklung eines SWE-CASE-TOOLs

Entwicklungs-Team:	David Eckel (2327134)
	Florian Schiffel (9073402)
	Stefan Maier (7514857)
Kunde:	DHBW Software Systems gGmbH
Projektleiter Kunde:	Bohl, Peter, Dipl-Ing.
Datum der Abgabe:	17.06.2020

Inhalt

Einleitung

Aufgaben Beschreibung/ einleitende Worte (max. 1 Seite)

Im Rahmen der Vorlesung Software Engineering I des 4. Semesters soll ein CASE (Computer-aided Software Engineering) Tool für die Erstellung eines Lastenhefts/Anforderungsanalyse entwickelt werden. Dazu wurden bereits in separaten Projekten die Anforderungen durch den Kunden evaluiert und Teilbereiche für die erste Implementierung spezifiziert. Anhand der Anforderungen für den ersten Release wird nun im Rahmen des 3. Projektes ein CASE Tool implementiert.

Da es sich durch die Anforderungen um eine Desktop Applikation handelt, wurde für die Implementierung die Programmiersprache Java ausgewählt. Mit der Programmiersprache Java ist es möglich Plattform unabhängige Programme zu erstellen. Des Weiteren ist die Verarbeitung von XML Dateien, das die zentrale Speicherung der Softwaredaten darstellt, sehr einfach in Java zu implementieren.

Als Build Tool wird Maven eingesetzt und als Testframework JUnit.

Die Softwarearchitektur wurde mittels Visual Paradigm erstellt, da dieses Programm alle gängigen UML Diagrammtypen für eine Softwarearchitektur und Spezifikation besitzt.

Im nachfolgenden Kapitel sind die Tools aufgeführt, um die Software eigenständig zu Kompilieren und die Software Architektur mittels Visual Paradigm einzusehen.

Installations-/ Debugging-Hinweise

Installation Java IDE

Für die Kompilierung der Software wird **Intellij IDEA 2019.3.5** als Integrated Development Environment (IDE) empfohlen.

Wenn nicht bereits installiert, bitte das Tool über folgenden Link herunterladen und, den entsprechenden Anweisung folgend, für das gegebene Betriebssystem installieren.

https://www.jetbrains.com/idea/download/other.html

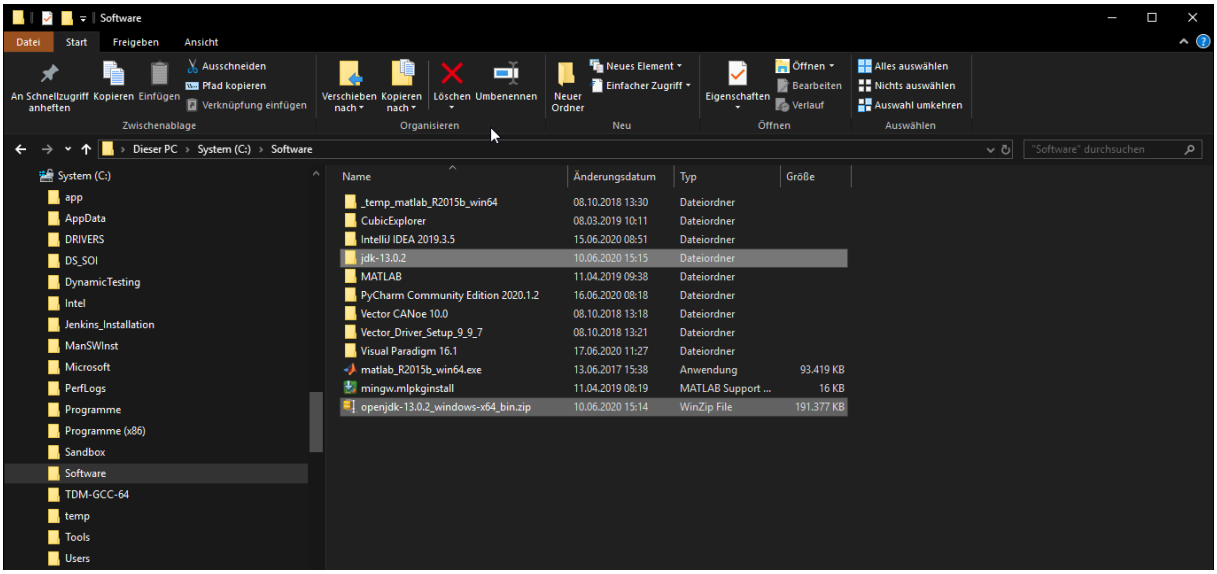
Installation des SDK

Um das Projekt korrekt kompilieren zu lassen wird das Java SDK benötigt. Es wird empfohlen OpenJDK 13.0.2 als SDK zu verwenden. Das SDK lässt sich über folgenden Link herunterladen

https://jdk.java.net/archive/

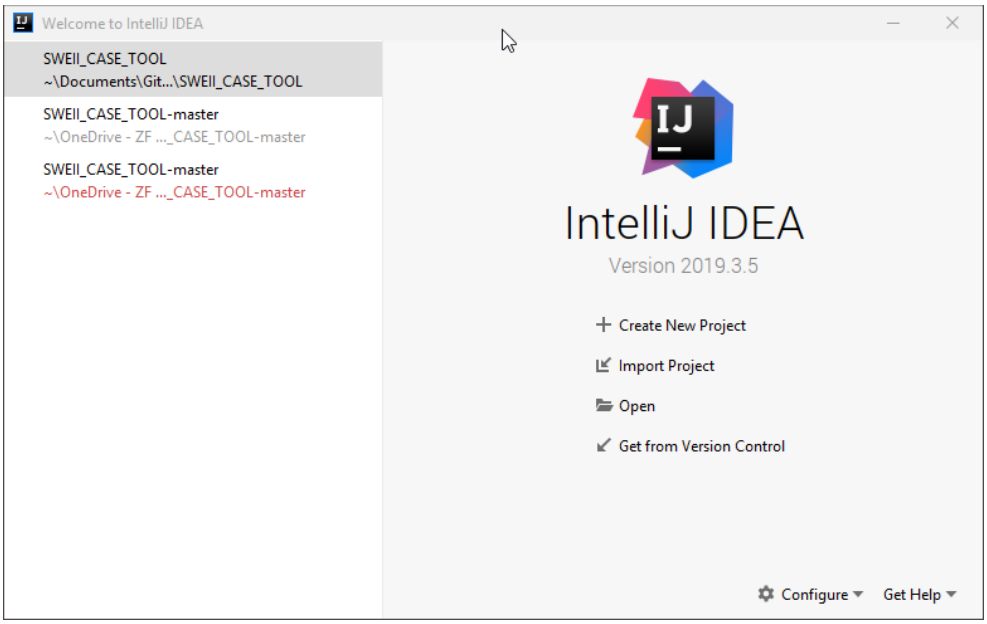
Für die Installation von OpenJDK bitte die entsprechende Datei für das Betriebssystem aus dem Link herunterladen. Der Download erfolgt entweder als ZIP-Archiv oder als tar.gz Archiv.

Anschließend den Download mit einem ZIP Tool ihrer Wahl (z.B. WinZip, 7Zip, etc.) in einem Verzeichnis ihrer Wahl extrahieren. Hier gezeigt in C:\Software\jdk-13.02



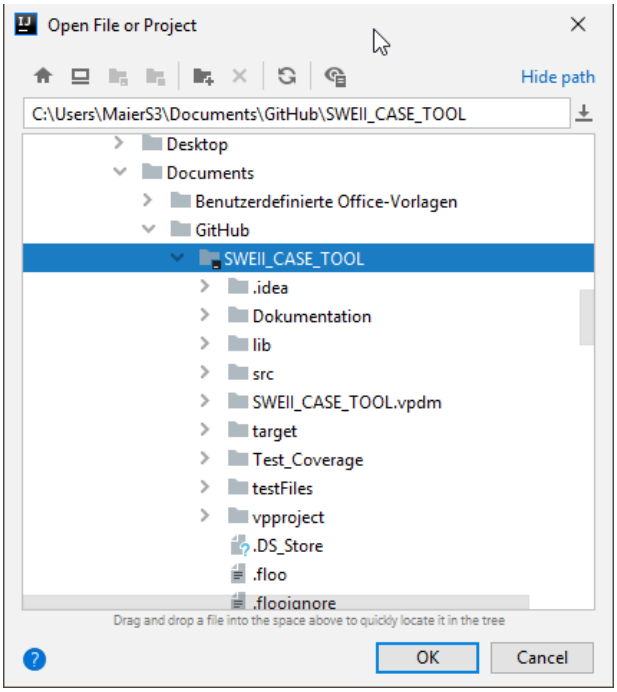
Um die SDK für das Projekt korrekt zu konfigurieren bitte folgende Schritte in IntelliJ IDEA durchführen.

1.) Öffnen Sie IntelliJ IDEA, folgendes Fenster erscheint:



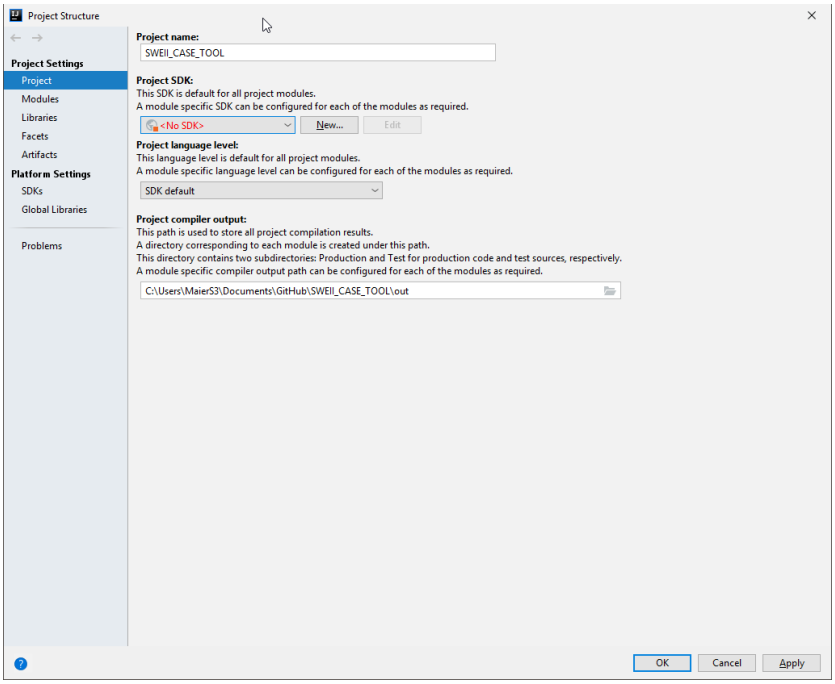
2.) Wählen Sie Öffnen (Open) um eine Projektstruktur zu laden.

3.) Wählen Sie nun den Pfad der Quelldateien aus. Ein mit IntelliJ kompatibles Projekt wird mit einem kleinen Symbol am Ordner gekennzeichnet. Selektieren Sie dieses Projekt und wählen Sie OK.



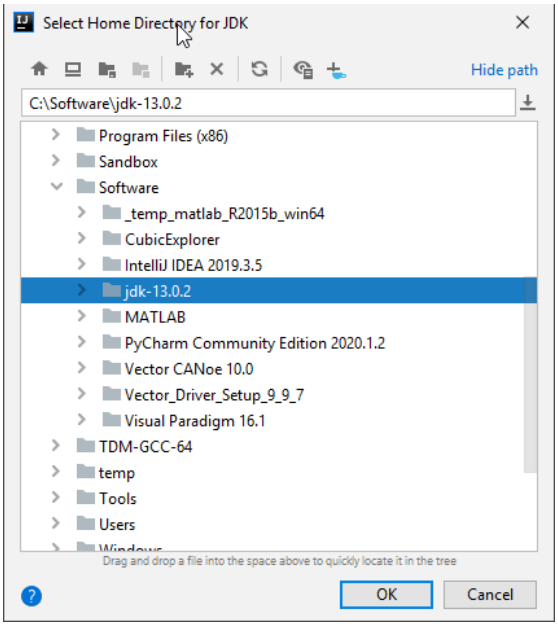
4.) Das Projekt wird in IntelliJ IDEA geladen.

5.) Gehen Sie nun im Menü auf File/Project Structure. Folgendes Fenster sollte erscheinen:

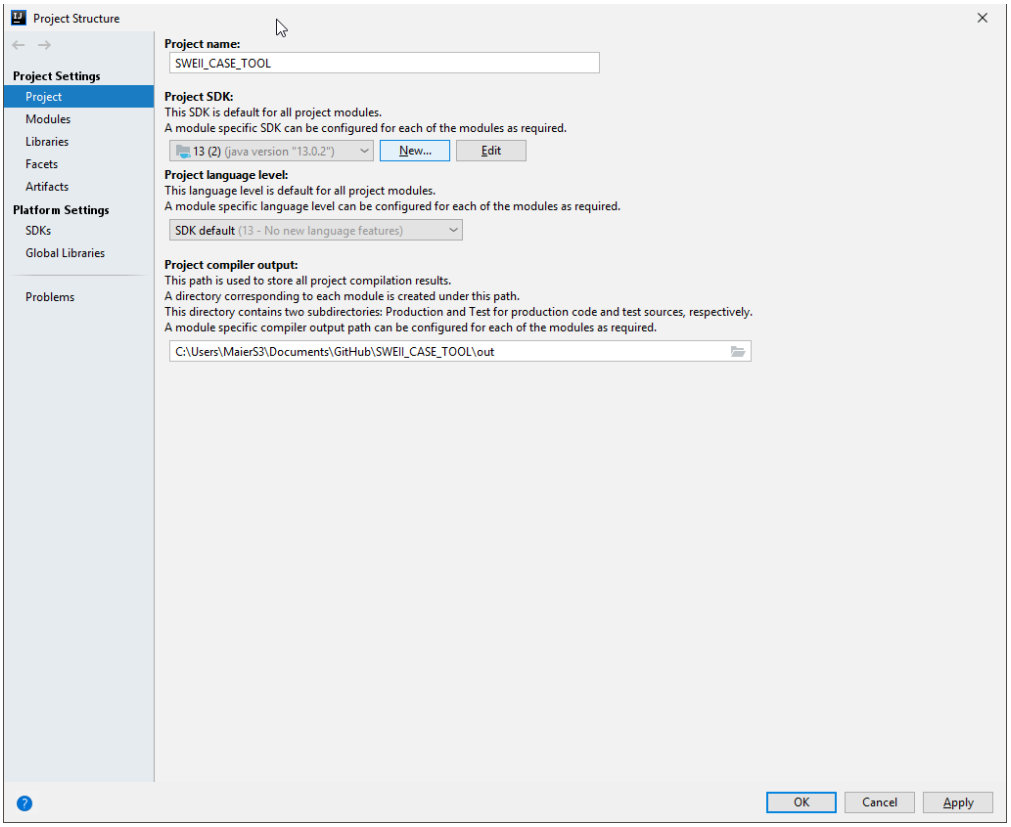


6.) Klicken Sie unter dem Punkt Project SDK auf „New“ und wählen Sie den Punkt „JDK“ aus.

7.) Wählen Sie nun den Ordner aus in dem Sie OpenJDK extrahiert haben. In diesem Beispiel C:\Software\jdk-13.0.2 und bestätigen Sie mit OK.



8.) Nachdem IntelliJ die JDK Installation erkannt hat, sollte unter Project SDK die Version 13 wie folgt dargestellt werden. Bestätigen Sie auch dieses Fenster mit OK.



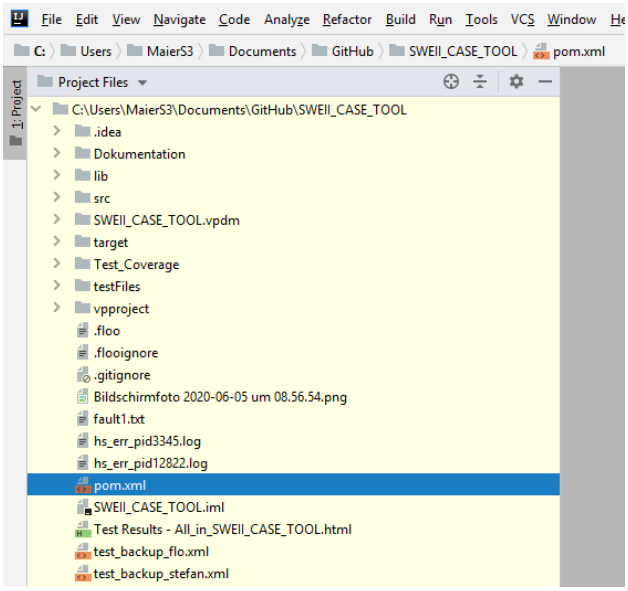
Die Java SDK ist nun für das Projekt installiert und konfiguriert.

INSTALLATION UND KONFIGURATION ALS MAVEN PROJEKT

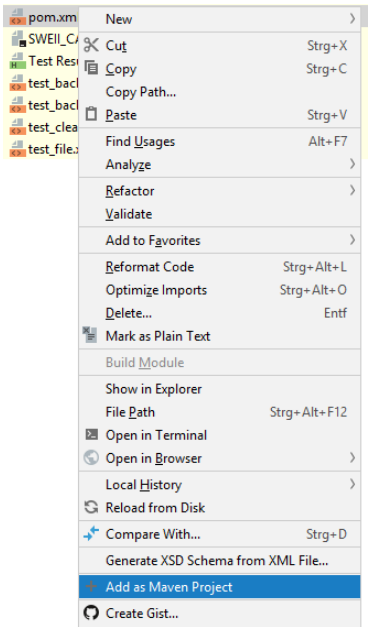
Unter Umständen kann es vorkommen, dass das Projekt noch nicht als Maven Projekt konfiguriert ist. Maven ist in der Regel bereits in IntelliJ IDEA als Plugin installiert, wenn die empfohlene Version verwendet wird. *Andernfalls kann es über File/Settings/Plugins „Maven“ installiert werden.*

Um das Projekt als Maven Projekt zu konfigurieren, bitte folgende Schritte durchführen:

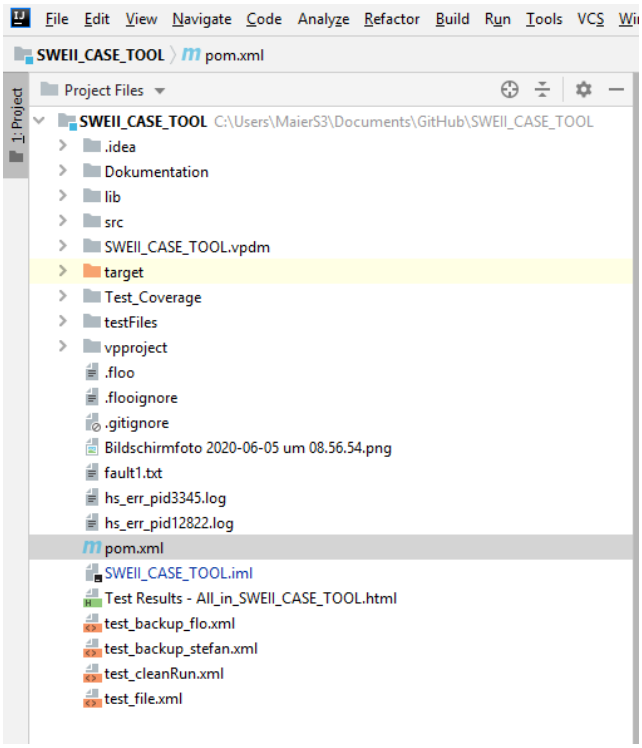
1.) In der Projektstruktur in IntelliJ IDEA das **pom.xml File selektieren**



2.) Über die rechte Maustaste im Kontextmenü den Punkt „Add as Maven Project“ auswählen.



Ist das Projekt korrekt konfiguriert, erscheint ein blaues M (für Maven) an dem pom.xml File

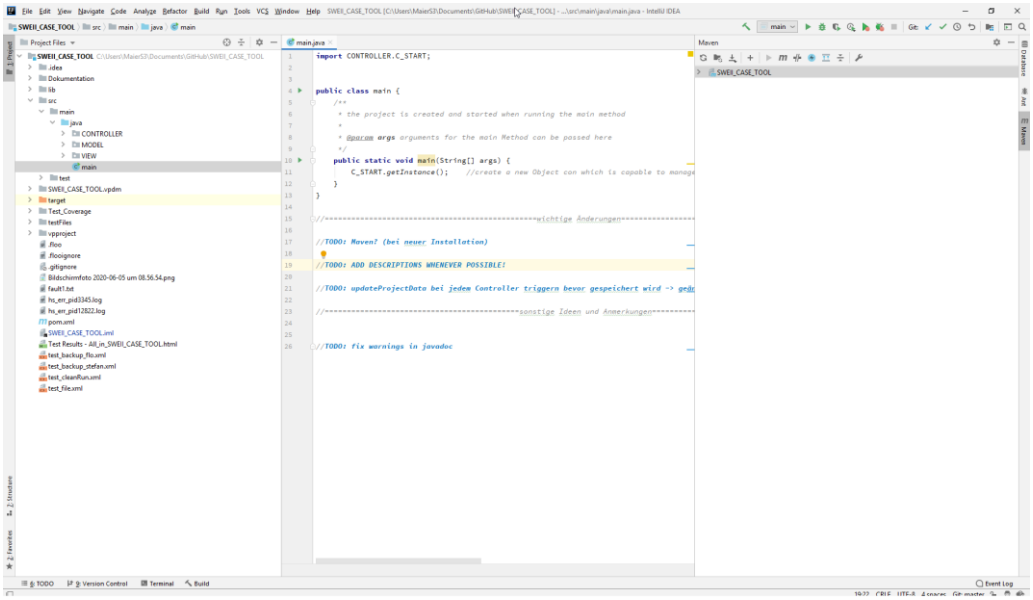


Hinweis: Falls die Auswahlmöglichkeit „Add as Maven Project“ nicht erscheint ist das Maven Projekt bereits korrekt angelegt.

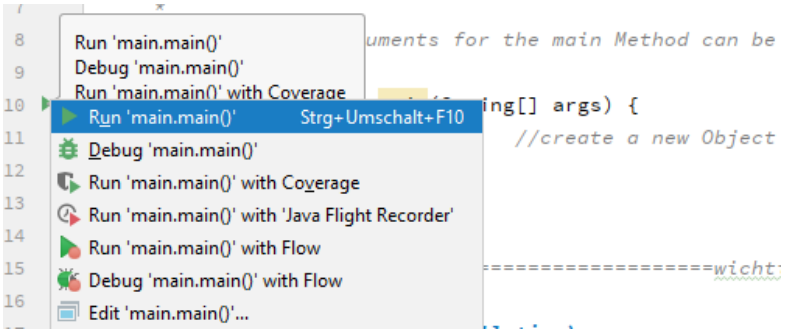
KOMPILIEREN DER APPLIKATION

Nachfolgend soll gezeigt werden, wie das Projekt mit installierter JDK und Maven korrekt in IntelliJ IDEA kompiliert werden kann.

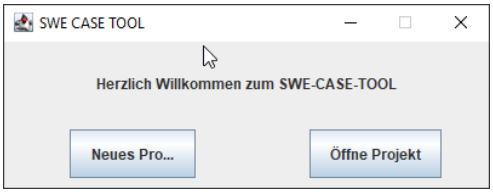
1.) Wählen Sie in IntelliJ in der Projektstruktur folgenden Pfad an. SWEII_CASE_TOOL/src/main/java/main.java



2.) Wenn alle vorherigen Schritte korrekt durchgeführt wurden sollte neben der main Methode ein grüner Pfeil erscheinen. Den Pfeil anwählen und „Run ‘main.main()‘“ anwählen



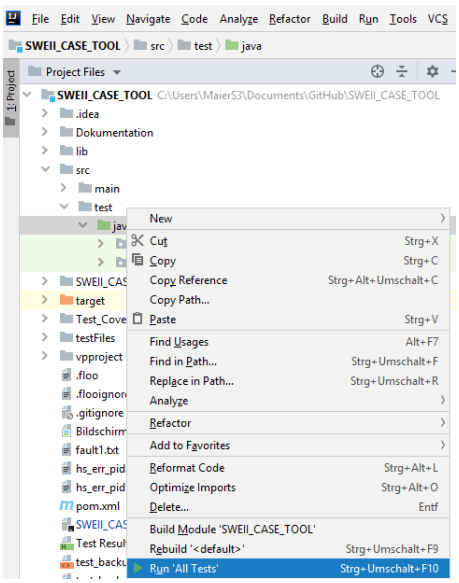
Das Projekt kompiliert und es öffnet sich das Startfenster.



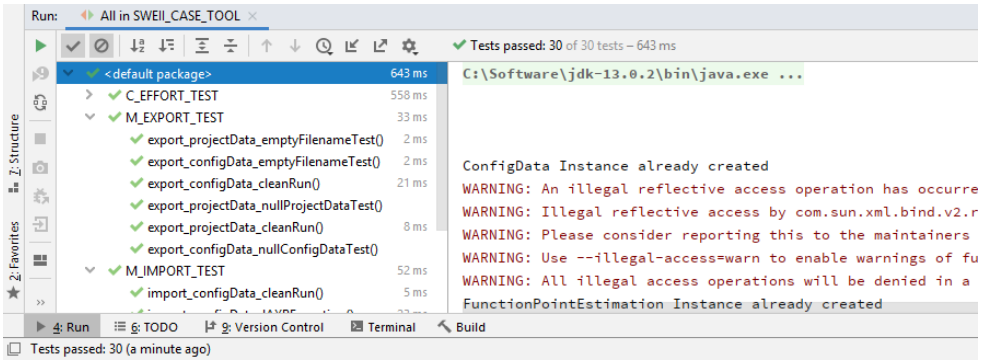
Kompilation der Unit Tests

Um die geforderten Tests durchführen zu lassen, folgende Anweisungen befolgen:

1.) Folgenden Pfad im Projekt Browser von IntelliJ selektieren: SWEII_CASE_TOOL/src/test/java
2.) Mit einem rechten Mausklick im Kontextmenü „Run ‘All Tests‘“ auswählen.



Das Projekt wird kompiliert und mittels des Testframeworks JUnit werden die Tests durchgeführt. Unter „Run“ werden die Testergebnisse angezeigt.



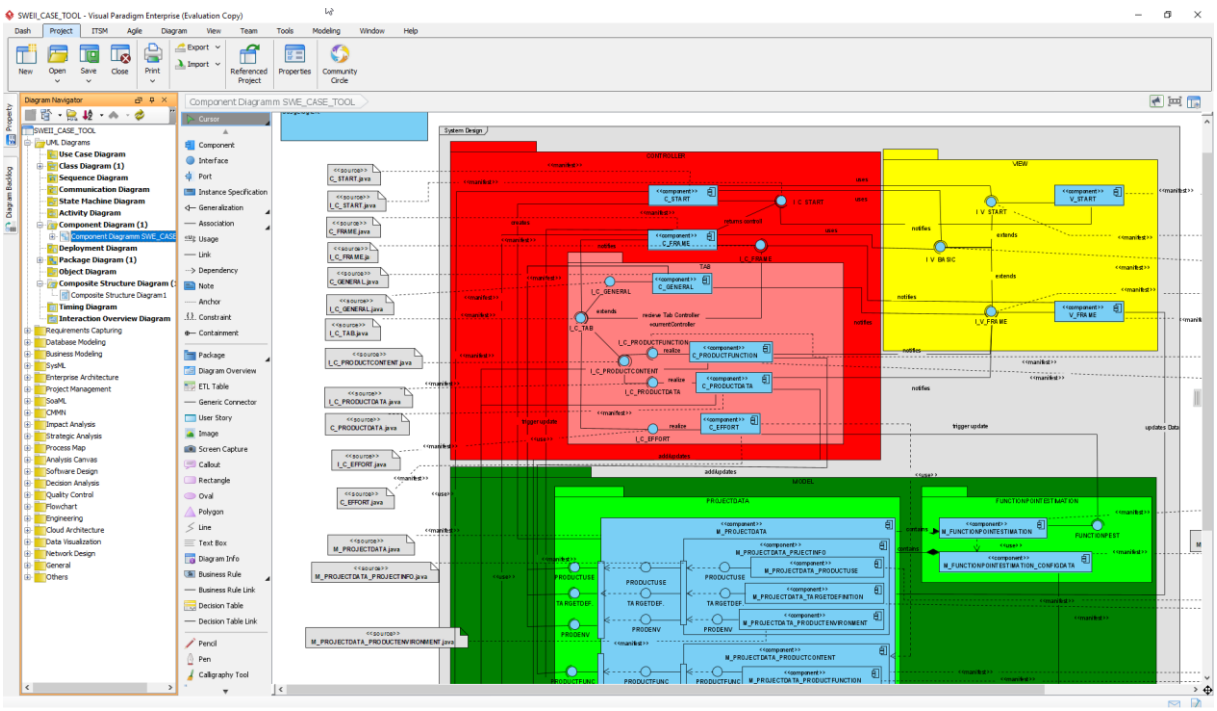
Installation von Visual Paradigm

Um einige Diagramme anzeigen zu lassen, muss Visual Paradigm installiert werden. Die Community Edition steht unter folgendem Link frei zur Verfügung:

https://www.visual-paradigm.com/download/community.jsp

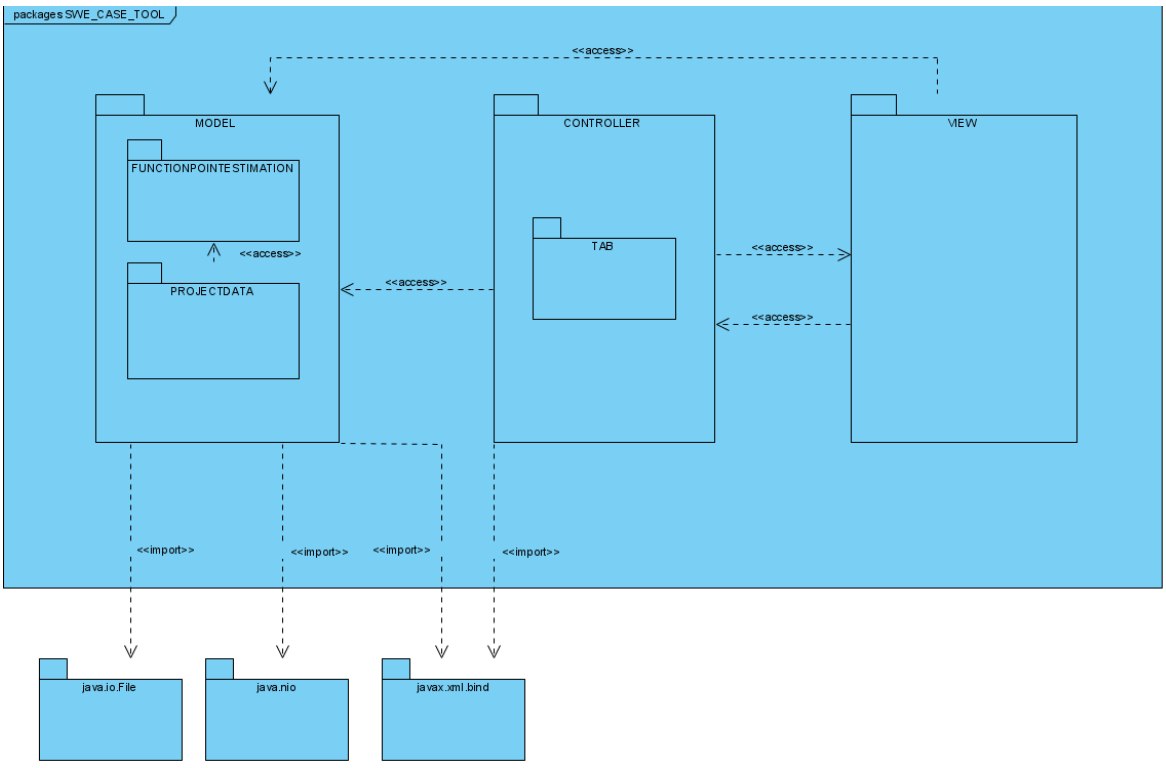
Um die Diagramme zu öffnen bitte folgende Datei unter **SWEII_CASE_TOOL/Dokumentation/Diagramme/VPPDesign/CASE_TOOL_Diagrams.vpp** öffnen.

Wenn sich Visual Paradigm geöffnet hat sind alle Diagramme im Diagram Navigator unter UML Diagrams zu finden. Der Diagram Navigator befindet sich in der linken Seitenleiste.



Programmstruktur

Um die leider nach wie vor oft zum Einsatz kommende, für die professionelle Durchführung eines Software-Projektes jedoch unzureichende Code and Fix Vorgehensweise direkt von Anfang an zu unterbinden, wurden bereits zu Beginn des Projekts die Anforderungen genauestens analysiert und die Planung im Team besprochen. Hierbei wurde schnell klar, dass sich die gestellten Anforderungen an Datenhaltung, Ein- und Ausgabe sowie Steuerung des zu erstellenden Programms am besten durch den Einsatz des Model-View-Controller Prinzips realisieren lassen. Zusätzlich wurden bei der Analyse der Anforderungen mehrere fast vollständig unabhängige Teilbereiche des komplexen Endprodukts erkannt. So ist die Erfassung von Produktfunktionen zwar sehr ähnlich zur Erfassung der Produktdaten, bei der Umsetzung der Produktfunktionen bewährte Methoden können somit bei der Umsetzung der Produktdaten angewendet werden und erfordern einen geringeren Aufwand in der Entwicklung. Eine wirkliche Zusammenarbeit der beiden Module ist jedoch nicht gegeben, die einzigen Schnittstellen sind die Datenhaltung im Modell sowie die Abfrage zur Aufwandsabschätzung. Auch sind die drei Module der Daten, Funktionen sowie Abschätzung logisch gesehen komplett unabhängig von den allgemeinen Produktinformationen wie Zielbestimmung, Produkteinsatz sowie Umgebung. Zusätzlich zur Aufgliederung in Model, View und Controller konnte eine vertikale Dekomposition in vier Bereiche erreicht werden. In allen drei Bereichen des MVC-Musters konnte somit eine Aufteilung in unabhängige Modelteile, Controller und Views gewährleistet werden.



Nachdem das grobe Softwaregerüst somit festgelegt war, haben wir uns dazu entschieden

einen zusätzlichen Start-Controller sowie eine Start-View zu implementieren, welche als alleinigen Zweck die Anzeige und das Aufrufen der entsprechenden Methoden im Modell zum Öffnen oder Erstellen von Dateien steuern müssen. Das hat den Vorteil, dass spätere Änderungen am Dateihandling ohne Probleme implementiert werden und sogar die ganze Darstellung sowie Steuerung problemlos getauscht werden könnte. Der Hauptcontroller des Programms (C_FRAME) ist somit unabhängig vom Dateihandling und muss nur die korrekte Benachrichtigung und Zuweisung von Funktionen der einzelnen Tabs zu den entsprechenden Controllern sicherstellen. Das entsprechende Interface I_C_FRAME ermöglicht auch hier das Tauschen einzelner Funktionen oder sogar des ganzen Controllers, solange gewährleistet ist, dass ein neuer Frame-Controller das Interface implementiert und die Tab-Controller entsprechend ihrer Funktionen benachrichtigt.

Die oben beschriebene Software Architektur als MVC mit vertikaler Dekomposition lässt sich anhand des Komponentendiagramms gut erkennen. Auch die sehr hohe Modularität zwischen den einzelnen Komponenten ist zu erkennen, so sind bis auf die Projektdatenstruktur alle Elemente austauschbar und anpassbar. (s. Anhang und **SWEII_CASE_TOOL/Dokumentation/Diagramme/VPPDesign/CASE_Tool_Diagrams.vpp** in Visual Paradigm unter Components Diagramm (s. Anleitung aus Kapitel 2))

MODEL

Projektdaten

Nach den Anforderungen soll das System so strukturiert werden, dass jedem angelegtem Projekt folgende Daten zugrunde liegen:

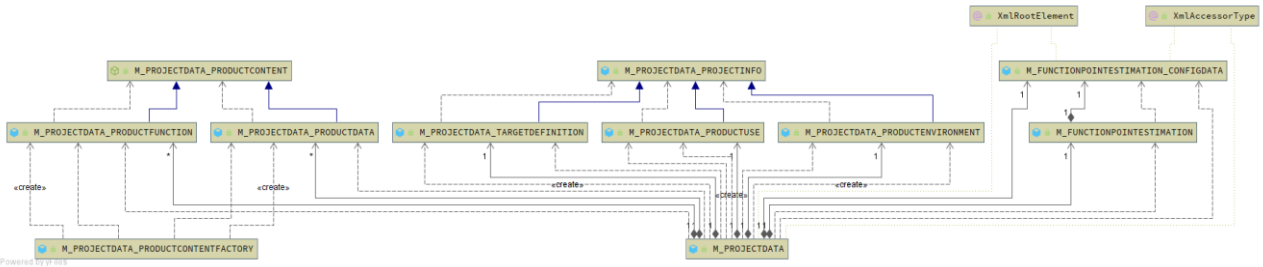
- Zielbestimmung
- Produkteinsatz
- Produktumgebung
- Produktfunktionen
- Produktdaten
- Aufwandschätzdaten
- Schätzkonfiguration

Die Daten sind Projektbezogen und unabhängig von anderen Projekten, somit lassen sich die oben angeführten Informationen in einer Projektdaten-Struktur kapseln. Es ist zudem zu erkennen, dass bestimmte Informationen ähnlich strukturiert sind. Hierfür lässt sich auf einer höheren Abstraktionsebene durch abstrakte Klassendefinitionen eine allgemeine Beschreibung für Projektinformationen und Produktinhalte finden. Dabei besteht eine Projektinformation aus den Bestandteilen „Titel“ und „Inhalt“ über die den Titel und Inhalt lässt sich Zielbe-

stimmung, Produkteinsatz und die Produktumgebung eindeutig Beschreibung und voneinander differenzieren. Gleiches gilt ebenso für die Produktinhalte. Diese lassen sich durch abstrahieren in dem sie alle eine ID besitzen eine DET Gewicht eine Function Point Gewicht sowie eine Function Point Kategorie. Detailinformationen, in denen sich Produktdaten und Produktfunktionen unterscheiden, werden in den separaten Klassen definiert. Jede der beiden Klassen erbt ebenso ein *calculateWeight* Methode, die die Berechnung des FP Gewichts anhand einer vorspezifizierten Tabelle durchführen soll. Die Daten der Aufwandsschätzung beinhalten alle Daten für Analyse entsprechend der Function Point Methode. Des Weiteren beinhaltet die Schätzkonfiguration Faktoren, die die Aufwandsschätzung beeinflussen. Aus den Anforderungen geht hervor, dass Schätzkonfigurationen exportiert und importiert werden sollen, daher wird die Schätzkonfiguration in eine separate Klasse ausgelagert (s. 3.1.2).

Die Variablen der Subklassen von *M_PROJECTDATA* sind öffentlich sichtbar (*public*), da diese Variablen vom User jederzeit einsehbar und veränderbar sind.

Aus nachfolgender Abbildung lässt sich die Struktur der Projektdaten erkennen. Alle Daten des Modells sind in *M_PROJECTDATA* zusammengefasst.

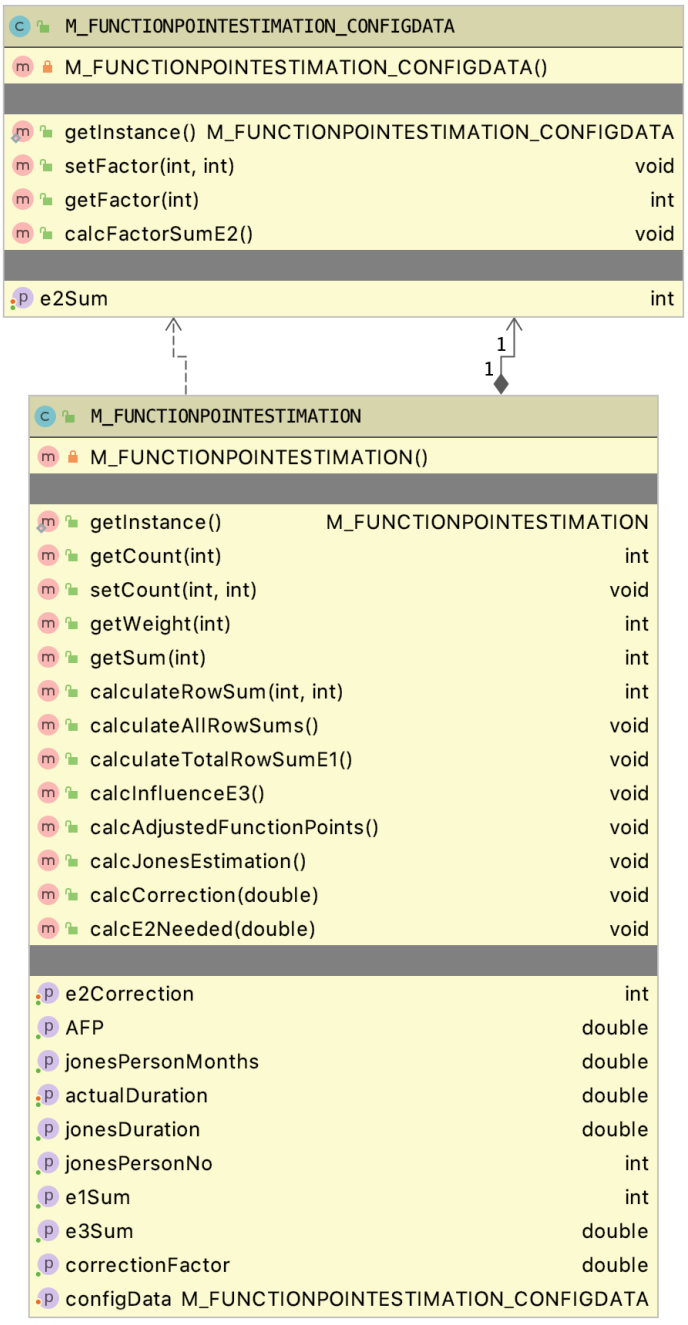


Da die Daten als XML exportiert werden sollen, wird für die Projektdaten sowie die Konfigurationsdaten der Aufwandsschätzung ein XML Root Element hinzugefügt, welches es ermöglicht die Daten mit Hilfe der Export Funktion in einer zusammenhängenden, korrekten Struktur exportieren und importieren zu können.

Factory Methode für Produktinhalte

Für Produktinhalte werden in der Laufzeit des Programms mehrere Objekte instanziiert, dabei entscheidet der Benutzer wann welches Objekt angelegt wird. Hierfür wird für das vereinfachte Anlegen eines Produktinhalts das Entwicklungsmuster der Factory Methode angewandt, das zur Laufzeit und unter Angabe von Parametern ein entsprechendes Objekt anlegt.

Aufwandsabschätzung



Die Daten der Aufwandsabschätzung gehören ebenfalls zum Model. Sie werden als Attribute in Objekten der beiden Klassen *M_FUNCTIONPOINTESTIMATION* sowie *M_FUNCTIONPOINTESTIMATION_CONFIGDATA* abgelegt. (Zur besseren Übersicht sind in dem Klassendiagram nur Konstruktoren und Methoden, inklusive der Getter und Setter aufgeführt. Die komplette Version inklusive aller einzelnen Variablen sind in einem Diagramm in dem Diagramme Ordner abgelegt.)

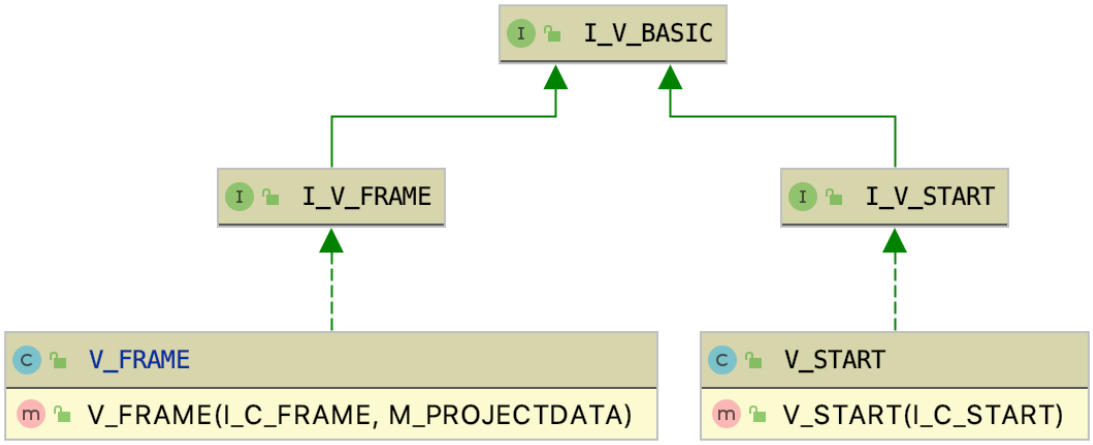
FUNCTIONPOINTESTIMATION

Die Klasse der *FUNCTIONPOINTESTIMATION* umfasst alle dafür nötigen Variablen wie das vorgegebene Gewicht (Variablenname: *weight…*), die Anzahl der verschiedenen Function Points (*count…*) sowie die gewichtete Summe (*sum…*). Anzahl, Gewicht und Summe können über entsprechende Getter-Methoden ausgelesen werden. Für die Anzahlen gibt es zusätzlich eine Setter-Methode um die durch neu eingegebene Function Points geänderte Anzahl zu aktualisieren. Durch die Referenz auf *configData* und somit den Zugriff auf die Einflussfaktoren kann die gewichtete Summe e3Sum sowie die Anzahl der Adjusted Function Points (*afp*) berechnet werden. Zusätzlich kann ein Korrekturfaktor sowie die eigentlich benötigte Summe der Einflussfaktoren (e2Correction) berechnet werden, weitere Informationen sind unter 0 Selbstoptimierung zu finden.

CONFIGDATA

In dem Singleton-Objekt der Klasse CONFIGDATA, werden alle 10 Einflussfaktoren (*factor…*), sowie die Summe aller Einflussfaktoren (*e2Sum*) gespeichert. Neben dem privaten Konstruktor und damit der Anwendung des Singleton Prinzips, gibt es eine Getter- und eine Setter-Methode für die Faktoren, denen als Integer die Nummer des gewünschten Faktors übergeben wird. Die Getter-Methode gibt daraufhin den Wert des gewählten Faktors zurück, der Setter-Methode wird zusätzlich der zu setzende Wert als Integer übergeben, welcher dann in der entsprechenden Variable abgespeichert wird. Zusätzlich gibt es eine Getter- und Setter-Methode für die Summe aller Faktoren *e2Sum*.

VIEW



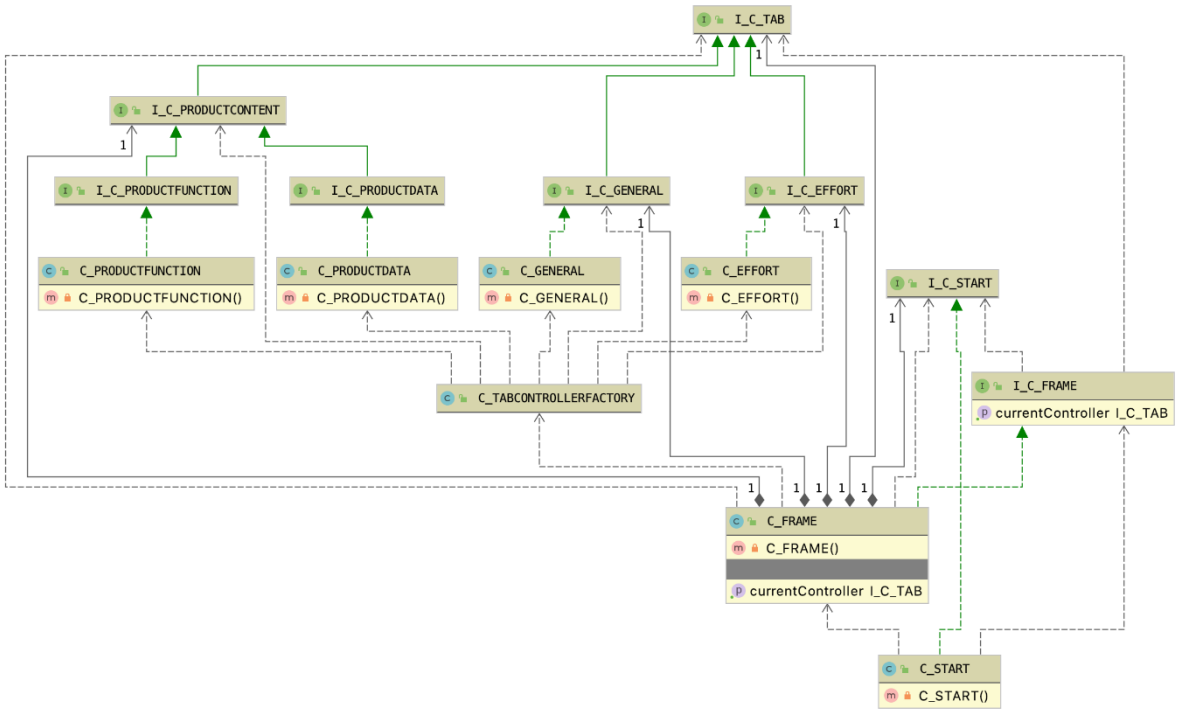
Jede der beiden Views *V_FRAME* und *V_START* sind jeweils von einem Interface abgeleitet, in dem die zu implementierenden Methoden definiert sind. Dadurch wird die genaue Implementierung verborgen (Information Hiding) und die einzelnen Methoden in den View-Klassen könne ohne zusätzliche Vorkehrungen treffen zu müssen ausgetauscht werden. Beide Interfaces ihrerseits erweitern das I_*V_BASIC* Interface welches jene Funktionen definiert, die jede View zwangsweise unterstützen muss. Darunter fallen die *show()* und *hide()* Funktionen. (Eine erweiterte Ansicht des Klassendiagramms ist der Übersicht halber im Anhang mit angefügt.)

Nach dem Start des Programms erzeugt der Kontroller *C_START* ein Objekt der Klasse *V_START* in welchem der Nutzer die Wahl zwischen dem Öffnen und Neu Erstellen eines Projekts hat. Sollte bei der Dateiauswahl ein Fehler auftreten erhält der Nutzer ein Pop-Up in dem der Fehler kurz beschrieben wird und die *V_START* wird erneut sichtbar. Wurde das Projekt erfolgreich erstellt beziehungsweise geöffnet, öffnet sich die zweite und damit auch hauptsächliche View *V_FRAME*. Diese View besteht wiederum aus 4 Tabs welche jeweils zusammenhängende Informationen anzeigen und somit jeweils durch einen eigenen Kontroller gesteuert werden können. Der Nutzer kann in die einzelnen Paneele seine Daten eingeben und sieht jeweils nur die für ihn in diesem Moment hilfreichen Elemente der View. Mit dem Button „Projekt schließen“ kann er ohne zusätzliches Speichern zurück zum Hauptmenü (*V_START*) gelangen. Mit dem Button „Projekt speichern (XML)“ kann er alle in dem aktuellen Projekt erfassten Daten in den beim Start gewählten Pfad abspeichern um sie zu einem späteren Zeitpunkt anpassen oder den Datensatz weitergeben zu können.

Im Gegensatz zu den ersten drei Tabs ist der vierte Tab „Aufwandsschätzung“ seinerseits erneut in drei Tabs unterteilt. So sind die verschiedenen Phasen der Abschätzung besser unterteilt und der Nutzer behält den Überblick über die eingegebenen Daten. Über anklicken der Tabs oder durch auswählen der „Schritt zurück“ und „Nächster Schritt“ Buttons kann der gewünschte Schritt ausgewählt werden. In „Function Points“ wird dem Nutzer eine tabellarische Übersicht der bisher erfassten Function Points dargestellt. Sollten die erfassten Daten nicht seinen Erwartungen entsprechen kann er so jederzeit in den ersten beiden Tabs „Produktfunktionen“ sowie „Produktdaten“ Anpassungen vornehmen, welche dann automatisch aktualisiert in der Übersicht erscheinen. Entsprechen die dargestellten Daten allen zu erfassenden Function Points, kann der nächste Schritt gewählt werden. Hier kann der Nutzer über Schieberegler die einzelnen Anpassungsfaktoren für sein individuelles Projekt einstellen. Die aktuelle Anzahl aller möglichen Faktoren wird dem Nutzer in einem Balken inklusive der dadurch verursachten prozentualen Änderung des Projektaufwands angezeigt.

Wurde mit dem SWE-CASE-TOOL bereits ein Projekt geplant und es stehen somit verifizierte Einflussfaktoren zur Verfügung können diese über Auswählen des „Faktoren importieren“ Buttons importiert werden. Die zu importierenden Faktoren müssen im gleichen Dateipfad wie das Projekt liegen und der Dateiname muss der Konvention „Projektname_config.xml“ entsprechen. Der „Faktoren exportieren“ Button exportiert die Faktoren automatisch in den Projektpfad unter dem Dateinamen des Projekts, versehen mit der Endung „_config.xml“.
Sind die Faktoren entsprechend eingestellt oder importiert, kann der nächste Tab „Berechnung“ gewählt werden. In diesem Tab werden zuerst alle bisher erfassten und daraus berechneten Daten, wie die Summe der Function Points oder die Anzahl an Adjusted Function Points aufgelistet. Aus diesen Daten wurde beim Öffnen des Tabs bereits der Aufwand des erfassten Projektes anhand der Jones-Schätzung ermittelt. Ist der tatsächliche Aufwand des Projekts bekannt, kann dieser in der Einheit Personenmonate in das entsprechende Feld eingetragen werden. Über den Button „Neuen Korrekturfaktor berechnen“ kann aus dem erfassten Aufwand in Zusammenhang mit dem tatsächlichen Aufwand ein Korrekturfaktor sowie die dafür benötigte Summe der Einflussfaktoren berechnet werden. Der Nutzer kann jetzt wahlweise selbstständig die Einflussfaktoren anpassen bis sie dem ermittelten Wert entsprechen oder aber er nutzt die automatische Anpassung. Neben der Anzeige der benötigten Einflussfaktorensumme erkennt der Nutzer dank eines Farbcodes sofort ob und wie weit er noch von dem zu erreichenden Wert entfernt ist. Sollte die Aufwandsdifferenz nicht alleine durch die Anpassung der Einflussfaktoren ausgeglichen werden können, wird dies dem Nutzer ebenfalls angezeigt und der Button zur automatischen Anpassung lässt sich nicht anwählen.

CONTROLLER



Jede der unter 0 dargestellten Views hat einen entsprechenden Controller *C_VIEWNAME* die von der jeweiligen View über Nutzereingaben informiert werden und diese entsprechend verarbeiten. Zusätzlich zu diesem C_FRAME Controller haben die einzelnen 4 Tabs der Frame View einen entsprechenden Controller der für spezifische Eingaben in diesem Tab zuständig ist. Jeder der TAB-Controller sowie der C_FRAME Controller werden vom Interface I_C_TAB abgeleitet und müssen somit eine Funktion *setLinks()* implementieren, welche die nötigen Referenzen zur Frame View sowie den Projektdaten halten. In der ebenfalls zu implementierenden Funktion *updateProjectData()* aktualisiert seinen speziellen Teil der Projektdaten.

Zum Start des Programms erzeugt *C_FRAME* mit Hilfe des Factory Prinzips in der Klasse *C_TABCONTROLLERFACTORY* die Instanz eines jeden TAB-Controllers. Durch Verwendung des Singleton-Prinzips ist sichergestellt, dass zur Laufzeit des Programms jeweils nur genau ein Controller pro Tab existieren kann. Während das Programm läuft hält *C_FRAME* eine ständig aktualisierte Referenz auf den aktuell aktiven TAB-Controller in der Variable *currentController*. Wird ein bestimmter Button der *V_FRAME* betätigt, benachrichtigt die View ihren Controller *C_FRAME* darüber. Dieser leitet diese Benachrichtigung abhängig vom aktuell aktiven Controller an den jeweiligen TAB-Controller weiter. Jeder der TAB-Controller implementiert ein spezielles Interface, gekennzeichnet durch *I_C_CONTROLLERNAME*. Dies ermöglicht das Prinzip des Information Hidings und vereinfacht das Austauschen einzelner Funktionen wesentlich.

Prinzipien und Muster

Interfaces

Da ein Hauptaugenmerk bei der Entwicklung des SWE-CASE-Tools auf den stetigen Veränderungen im Projektplanungszyklus und den damit teilweise notwendig werdenden Anpassungen an einem entsprechenden Planungstool lag, wurden, wann immer möglich, Interfaces eingesetzt. Die mit „I_“ eindeutig gekennzeichneten Interfaces beschreiben alle möglichen Funktionen der implementierenden Klasse, der Methodenaufruf in einer anderen Klasse ist somit jedoch nicht von der endgültigen Implementierung abhängig. Dadurch wird es dem Programmierer ermöglicht einzelne Methoden anzupassen ohne Abhängigkeiten beachten zu müssen. Die einzige Bedingung, die im gestellt sein, sind die Eingabeparameter, die Ausgabeparameter sowie die Funktion einer Methode. Die genaue Implementierung ist dadurch jedoch komplett variabel.

Durch die Nutzung von Interfaces ist jedoch nicht nur die gewünschte Modularität gewährleistet, sondern auch das Entwurfsprinzip des Information Hiding ist gewährleistet. Die eine Methode über ein Interface aufrufende Klasse kennt nur die von diesem Interface zur Verfügung gestellten Funktionen, die genaue Implementierung bleibt ihr jedoch verborgen.

Für die Klassen des Models, also der Projektdaten, konnten keine Interfaces erstellt werden, da die Erweiterung JAXB dadurch keine Daten aus einer XML-Datei importieren kann. Da die Projektdaten jedoch auch für den Nutzer einsehbar sind, wurde auf diese Interfaces verzichtet. Die Modularität ist durch die Aufteilung der relevanten Informationen in verschiedene Unterklassen nach wie vor gegeben.

Singleton

Da sowohl die Controller, als auch die Views und Projektdaten für jede Instanz des Programms nur einmal existieren sollen und um mögliche Komplikationen durch mehrere sich gegenseitig beeinflussende Controller oder gar mehrere gleichzeitig geöffnete Projektdaten ausschließen zu können wurde, wann immer möglich das Entwicklungsmuster Singleton eingesetzt.

Hierfür haben die Klassen eine *private static* Referenz auf ihr Objekt, einen privaten Konstruktor sowie eine *public* Methode *getInstance*, welche eine Referenz zu dem Objekt der aufgerufenen Klasse zurückliefert. Das Entwicklungsmuster ist somit vergleichsweise einfach zu implementieren, es muss jedoch darauf geachtet werden, dass trotzdem bei der Erzeugung alle notwendigen Referenzen gesetzt werden. Dies kann entweder durch das Aufrufen der *getInstance* Methode mit den entsprechenden Übergabeparametern oder mit einer entsprechend zu implementierenden Funktion *setLinks()* geschehen. Im Rahmen des SWE-CASE-TOOLs kommt die zweite Methode zum Einsatz, der Programmierer sollte deswegen nach der erstmaligen Erstellung eines Objekts, welches die *setLinks()* Funktion implementiert auf alle Fälle diese Funktion aufrufen. Dies hat zwar den Nachteil, dass durch Nichtbeachten dieser Regel ein Objekt ohne Referenzen erstellt werden kann, es kann jedoch nicht dazu kommen, dass ein Entwickler Referenzen übergibt, dementsprechend davon ausgeht, dass die Referenzen gesetzt wurden, das Objekt jedoch noch auf die alten Referenzen zeigt, da der Konstruktor gar nicht aufgerufen wurde. Missachtet ein Entwickler die Regel, dass *setLinks()* aufgerufen werden muss, wird das Programm eine *NullPointerException* ausgeben, die die Fehlersuche im Gegensatz zu nicht gesetzten Referenzen deutlich vereinfacht.

Factory

Das Entwurfsmuster der Factory ermöglicht es während der Laufzeit neue Objekte verschiedener Klassen zu erzeugen. In der Klasse *M_PROJECTDATA_PRODUCTCONTENTFACTORY* ist die Methode *createProductContent(String id, String contentType)* implementiert, welche anhand des *contentTypes* ein neues ProduktContent Item erzeugt. Dies ist entweder eine Produktfunktion oder ein Produktdatum. Die Verwendung des Typbezeichners Any ist hierbei mit einer Template Methode vergleichbar, da erst beim return festgelegt wird, welchen Typ die Funktion zurückgibt. Das gleiche Muster wurde ebenfalls für die Erzeugung der TAB-Controller in der Klasse C_TABCONTROLLERFACTORY angewendet.

Programmtests

Die Testergebnisse lassen sich über folgenden Link einsehen:

SWEII_CASE_TOOL\Dokumentation\Test Results - All_in_SWEII_CASE_TOOL.html

Für die Testabdeckung gibt es einen Coverage Report der über folgenden Pfad eingesehen werden kann **SWEII_CASE_TOOL\Dokumentation\Coverage\index.html**

Import/Export

Für den Export und Import wurde die JAXB API verwendet. Diese ermöglicht es JAVA Objekte in XML-Dateien und XML-Dateien in JAVA Objekte zu konvertieren.

Es ist möglich sowohl das gesamte Projekt als auch nur die Konfiguration der Einflussfaktoren zu importieren und/oder exportieren.

Import

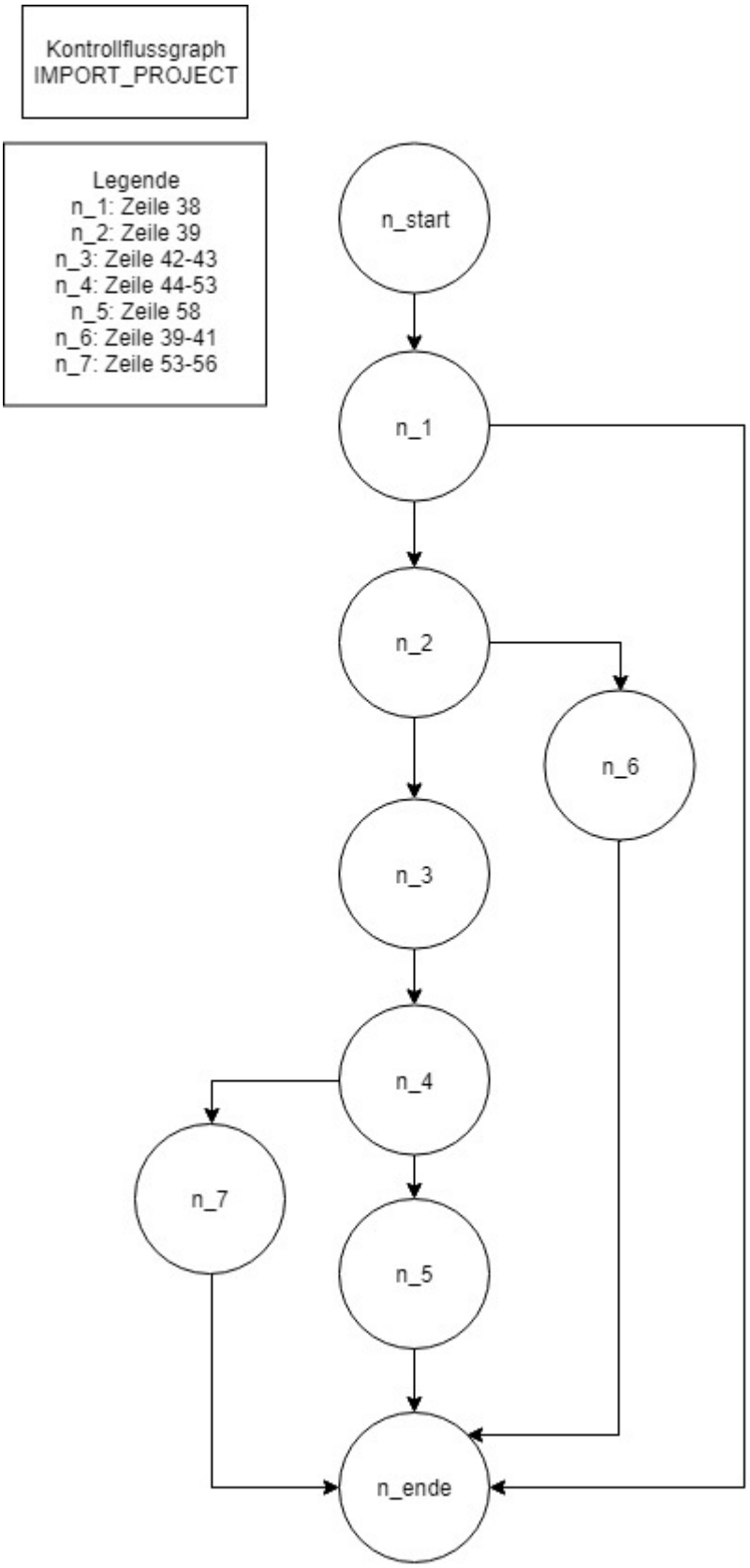
In M_IMPORT gibt es die beiden überladenden Methoden *importProject*.

Die Erste importiert ein ganzes Projekt und gibt die Projektdaten zurück. Hierbei wird zuerst überprüft, ob die Datei im angegebenen Pfad existiert und dann die Konvertierung mit JAXB versucht.

Bei der Ausführung dieser Methode kann es zu 4 Fällen kommen, welche auch die Äquivalenzklassen darstellen:

- **Die Datei, welche durch den Dateipfad gegeben ist, existiert nicht**
  → falls dies eintritt wird eine *InvalidPathException* geworfen, auf welche beim Testen geprüft wird
- **Der Dateipfad ist „null“**
  → falls dies eintritt wird eine *NullPointerException* geworfen, auf welche beim Testen geprüft wird
  Bei der Konvertierung mit JAXB tritt ein Problem auf
  → falls dies eintritt wird eine *JAXBException* von JAXB geworfen, welche abgefangen wird. Dann wird eine *IllegalStateException* geworfen, auf welche beim Testen geprüft wird
- **Die Methode wirft keinen Fehler**
  → falls dies eintritt wird erwartet, dass keine Exception geworfen wird und darauf wird getestet

Es wurden also 3 Tests konstruiert (der erste prüft die ersten beiden Äquivalenzklassen), wobei jeder jeweils einen bestimmten Zweig durchläuft und entweder eine Exception erwartet oder nicht. Damit wurde Zweigabdeckung erreicht.


Kontrollflussgraph
IMPORT_PROJECT
Legende
n_1: Zeile 38
n_2: Zeile 39
n_3: Zeile 42-43
n_4: Zeile 44-53
n_5: Zeile 58
n_6: Zeile 39-41
n_7: Zeile 53-56
n_start
n_1
n_2
n_6
n_3
n_4
n_7
n_5
n_ende

Die zweite Methode importiert die Konfiguration der Einflussfaktoren und überschreibt mit diesen die Einflussfaktoren des derzeitigen Projektes. Hierbei wird überprüft, ob die Datei im angegebenen Pfad existiert, das Projekt, welches die Konfiguration importiert, existiert und dann die Konvertierung mit JAXB versucht.

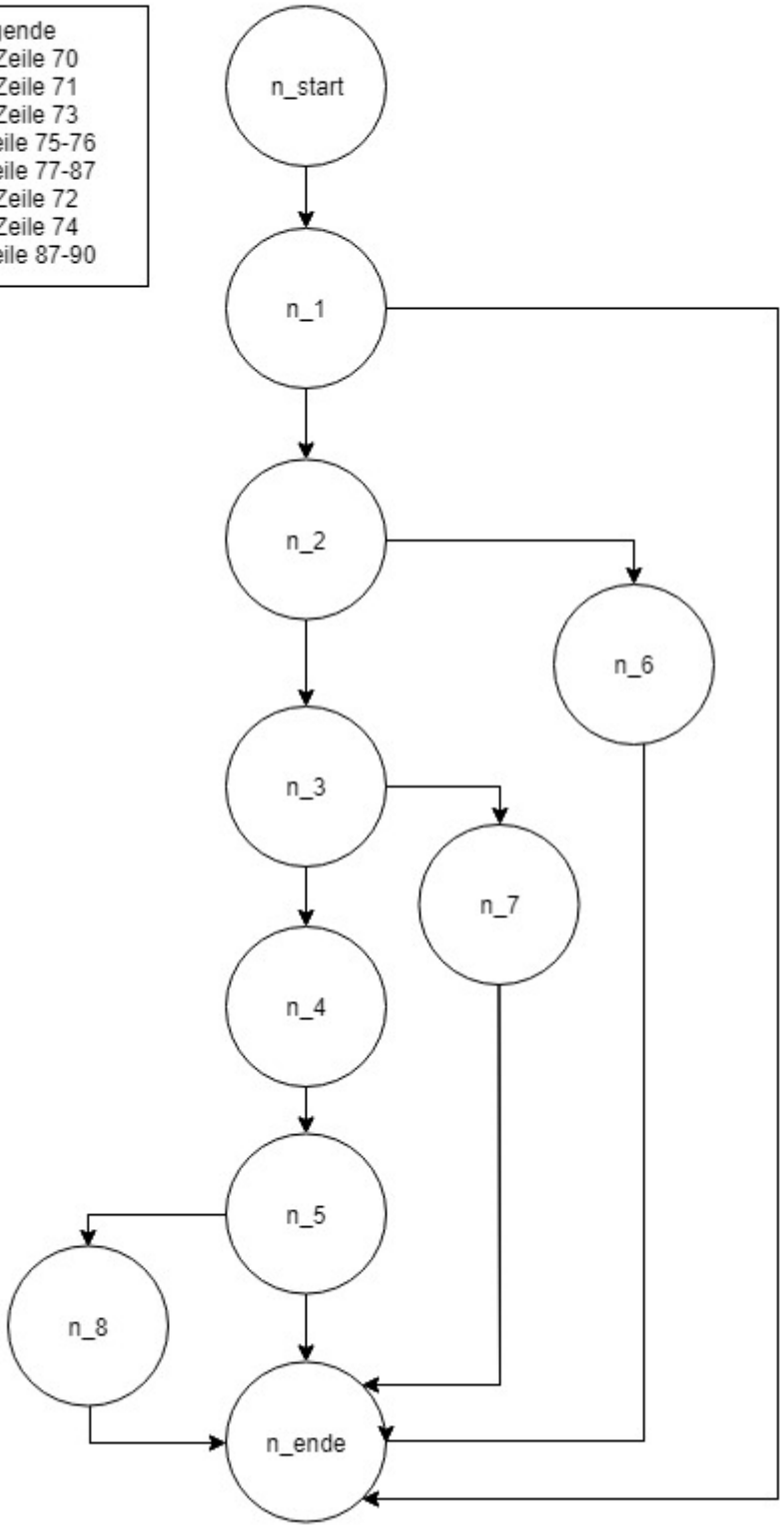
Bei der Ausführung dieser Methode kann es zu 5 Fällen kommen, welche auch die Äquivalenzklassen darstellen:

- **Die Datei, welche durch den Dateipfad gegeben ist, existiert nicht**
  → falls dies eintritt wird eine InvalidPathException geworfen, auf welche beim Testen geprüft wird
- **Der Dateipfad ist „null“**
  → falls dies eintritt wird eine NullPointerException geworfen, auf welche beim Testen geprüft wird
- **Das Projekt, welches die Konfiguration der Einflussfaktoren importiert ist „null“**
  → falls dies eintritt wird eine NullPointerException geworfen, auf welche beim Testen geprüft wird
- **Bei der Konvertierung mit JAXB tritt ein Problem auf**
  → falls dies eintritt wird eine JAXBException von JAXB geworfen, welche abgefangen wird. Dann wird eine IllegalStateException geworfen, auf welche beim Testen geprüft wird
- **Die Methode wirft keinen Fehler**
  → falls dies eintritt wird erwartet, dass keine exception geworfen wird und darauf wird getestet

Es wurden also 4 Tests konstruiert (der erste prüft die ersten beiden Äquivalenzklassen), wobei jeder jeweils einen bestimmten Zweig durchläuft und entweder eine Exception erwartet oder nicht. Damit wurde Zweigabdeckung erreicht.

Kontrollflussgraph
IMPORT_CONFIGDATA

Legende
n_1: Zeile 70
n_2: Zeile 71
n_3: Zeile 73
n_4: Zeile 75-76
n_5: Zeile 77-87
n_6: Zeile 72
n_7: Zeile 74
n_8: Zeile 87-90



Export

In M_EXPORT gibt es die beiden überladenden Methoden export.

Die Erste exportiert ein ganzes Projekt. Hierbei wird überprüft, ob der angegebene Pfad null oder leer ist oder nur aus Leerzeichen besteht. Außerdem wird überprüft, ob das zu exportierende Projekt null ist und die Konvertierung mit JAXB versucht.

Bei der Ausführung dieser Methode kann es zu 4 Fällen kommen, welche auch die Äquivalenzklassen darstellen:

- **Der Dateipfad ist leer oder besteht nur aus Leerzeichen**
  → falls dies eintritt wird eine *InvalidPathException* geworfen, auf welche beim Testen geprüft wird
- **Der Dateipfad ist „null“**
  → falls dies eintritt wird eine *NullPointerException* geworfen, auf welche beim Testen geprüft wird
- **Das zu exportierende Projekt ist „null“**
  → falls dies eintritt wird eine *NullPointerException* geworfen, auf welche beim Testen geprüft wird
- **Die Methode wirft keinen Fehler**
  → falls dies eintritt wird erwartet, dass keine exception geworfen wird und darauf wird getestet

Es wurden also 3 Tests konstruiert (ein Test prüft die ersten beiden Äquivalenzklassen), wobei jeder jeweils einen bestimmten Zweig durchläuft und entweder eine Exception erwartet oder nicht.

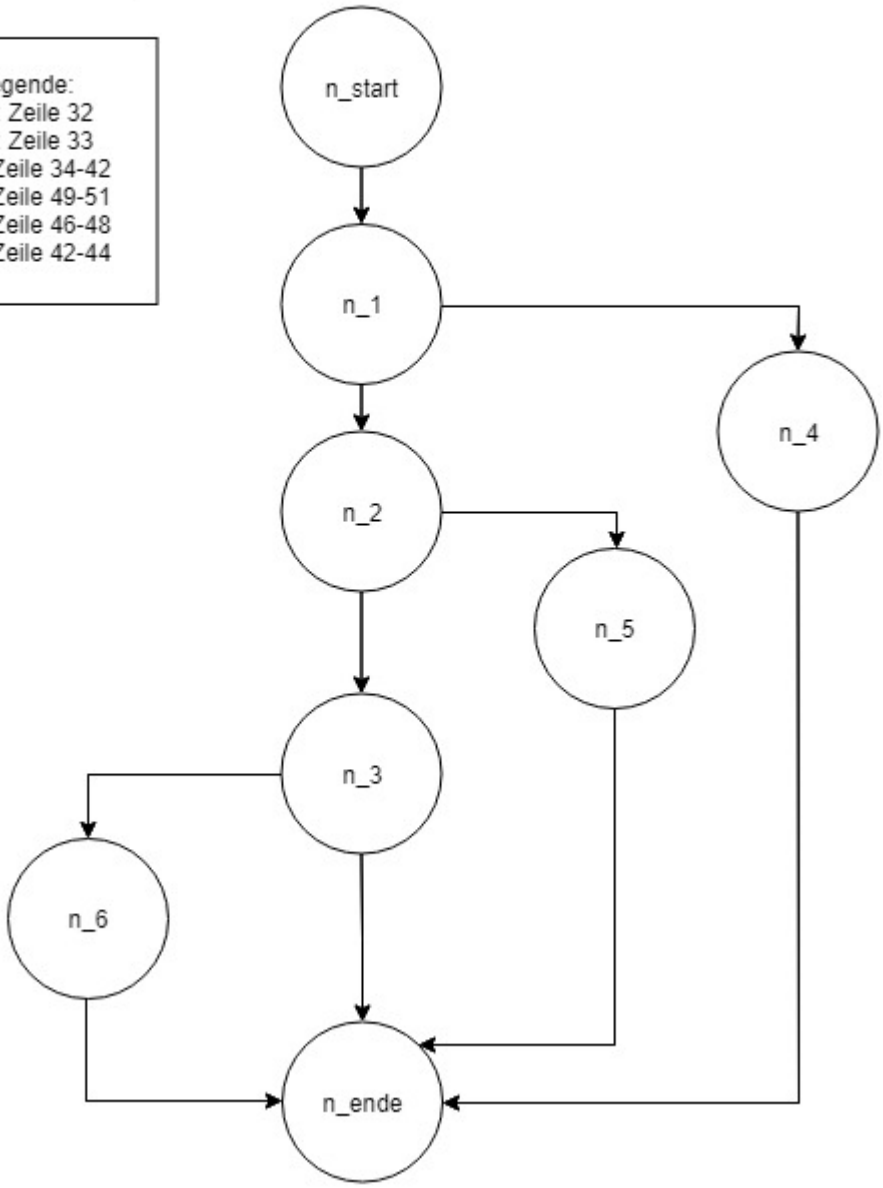
Auf eine *JAXBExeption* kann hier nicht getestet werden, da eine *JAXBException* bei der Konvertierung von Objekt nach XML nur in zwei Fällen geworfen wird:

- Ein Invalides Objekt soll konvertiert werden → dies ist nicht möglich, da hierfür eine falsche Klasse an die export Methode übergeben werden müsste
- Wenn der Output der Konvertierung an einen geschlossenen Outputstream übergeben wird → ebenfalls nicht möglich, da die Methode keinen Output hat

Damit kann hier keine vollständige Zweigabdeckung erreicht werden.

Kontrollflussgraph
EXPORT_PROJECT

Legende:
n_1: Zeile 32
n_2: Zeile 33
n_3: Zeile 34-42
n_4: Zeile 49-51
n_5: Zeile 46-48
n_6: Zeile 42-44



Die zweite Methode exportiert ein die Konfiguration der Einflussfaktoren eines Projektes. Hierbei wird überprüft, ob der angegebene Pfad null oder leer ist oder nur aus Leerzeichen besteht. Außerdem wird überprüft, ob das Objekt der zu exportierenden Einflussfaktoren null ist und die Konvertierung mit JAXB versucht.

Bei der Ausführung dieser Methode kann es zu 4 Fällen kommen, welche auch die Äquivalenzklassen darstellen:

- **Der Dateipfad ist leer oder besteht nur aus Leerzeichen**
  → falls dies eintritt wird eine *InvalidPathException* geworfen, auf welche beim Testen geprüft wird
- **Der Dateipfad ist „null“**
  → falls dies eintritt wird eine *NullPointerException* geworfen, auf welche beim Testen geprüft wird
- **Das zu exportierende Objekt der Einflussfaktoren ist „null“**
  → falls dies eintritt wird eine *NullPointerException* geworfen, auf welche beim Testen geprüft wird
- **Die Methode wirft keinen Fehler**
  → falls dies eintritt wird erwartet, dass keine exception geworfen wird und darauf wird getestet

Es wurden also 3 Tests konstruiert (ein Test prüft die ersten beiden Äquivalenzklassen), wobei jeder jeweils einen bestimmten Zweig durchläuft und entweder eine Exception erwartet oder nicht.

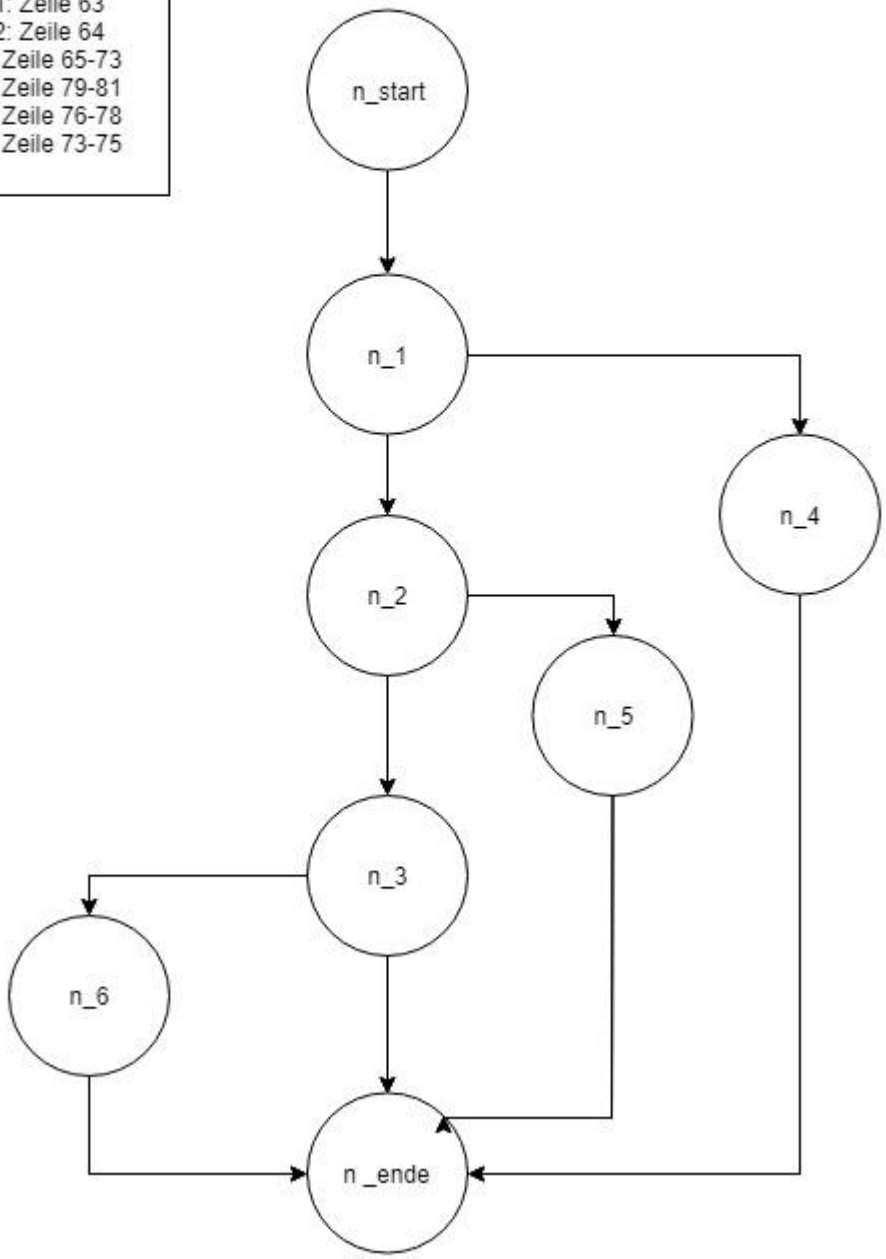
Auf eine *JAXBExeption* kann hier nicht getestet werden, da eine *JAXBException* bei der Konvertierung von Objekt nach XML nur in zwei Fällen geworfen wird:

- Ein Invalides Objekt soll konvertiert werden → dies ist nicht möglich, da hierfür eine falsche Klasse an die export Methode übergeben werden müsste
- Wenn der Output der Konvertierung an einen geschlossenen Outputstream übergeben wird → ebenfalls nicht möglich, da die Methode keinen Output hat

Damit kann hier keine vollständige Zweigabdeckung erreicht werden.

Kontrollflussgraph
EXPORT_CONFIGDATA

Legende:
n_1: Zeile 63
n_2: Zeile 64
n_3: Zeile 65-73
n_4: Zeile 79-81
n_5: Zeile 76-78
n_6: Zeile 73-75



Selbstoptimierung

Nachdem die Selbstoptimierung in der Klasse *M_FUNCTIONPOINTESTIMATION* einen bedeutenden Einfluss auf das Gelingen zukünftiger Projekte hat, muss der Nutzer sich auf sie verlassen können, sodass auch hier Tests mit Zweigüberdeckung konstruiert wurden. Ausgehend von der nach Jones berechneten Abschätzung des Aufwands wird, durch Anwenden der *Math.log()*-Funktion, in der *calcCorrection()* Funktion ein Korrekturfaktor berechnet.

$$correctionFactor = \ln\left(\frac{realTime}{jonesPersonMonths}\right)$$

Der Natürliche Logarithmus, als Inverse der Exponential-Funktion sorgt dafür, dass der Korrekturfaktor durch relativ kleine und damit gut handhabbare Werte auch große Abweichungen leicht ausgleichen kann, die Auflösung bei kleineren Abweichungen jedoch nicht zu unscharf wird.

$$correctedDuration = e^{jonesPersonMonths} * jonesPersonMonths * e^{correctionFactor}$$

Aus dem Korrekturfaktor kann durch Umformung der Formel für die *correctedDuration*, also mit Hilfe folgender Formel, der Wert berechnet werden, dem die Summe aller Einflussfaktoren entsprechen müsste, damit der abgeschätzte Aufwand der tatsächlichen Dauer entspricht.

$$e2Correction = \left(\frac{\sqrt[0,4]{\frac{realTime}{e^0 * jonesPersonNo}}}{e1Sum} - 0,7\right) * 100$$

$$= \left(\frac{\left(\frac{realTime}{e^0 * jonesPersonNo}\right)^{\frac{1}{0,4}}}{e1Sum} - 0,7\right) * 100$$

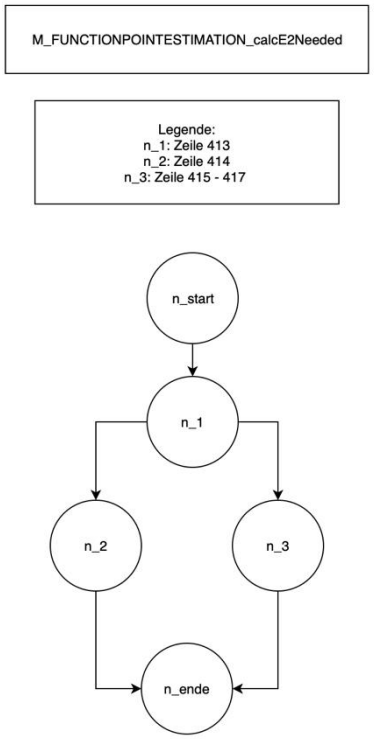
In Code umgesetzt entspricht diese Formel dann der Berechnung von *e2Correction* in Zeile 399. Zusätzlich wird das Ergebnis hier als Integer gecastet um einen, den Vorgaben der Faktoren entsprechenden, Wert zu erhalten.

Zeile 399:

```
e2Correction = (int) (((Math.pow((realTime / (Math.exp(0) * jonesPersonNo)), (1 / 0.4)) / e1Sum) - 0.7) * 100);
```

Die Berechnung dieses Werts in calcE2Needed beruht auf der Implementierung einer mathematischen Umformung. Da die Math.log()- sowie die Math.exp()-Funktionen als korrekt angenommen werden können, wird dieser Teil der Berechnung als korrekt angenommen. Sobald die notwendige Korrektur in Form der angepassten Summe der Einflussfaktoren E2 berechnet wurde (*M_PROJECTDATA_FUNCTIONPOINTESTIMATION.e2Correction*), folgt, wenn vom Nutzer der Knopf „Automatisch anpassen“ im Tab Berechnung oder Einflussfaktoren gedrückt wird, die automatische Anpassung. In der Klasse *C_EFFORT* sind hierfür die folgenden drei Methoden implementiert:

- *notifyAdjustFactors()*
- *increaseFactors()*
- *decreaseFactors()*



notifyAdjustFactors()

In der Methode *notifyAdjustFactors()* wird evaluiert ob und in welchem Umfang die Faktoren geändert werden müssen, sowie ob eine reine Änderung der Faktoren überhaupt die tatsächliche Dauer des Projekts ergeben kann. Über die Faktoren kann die Projektdauer maximal um ± 30 % verändert werden. In der Methode *notifyAdjustFactors()* wird zur Entscheidung ob und in welchem Umfang die Faktoren geändert werden eine Fehlervariable als Differenz aus *e2Sum* und *e2Correction* berechnet. (*e2Failure*) Aus diesen Informationen kann es logisch zu 3 Fällen kommen, welche weiter in insgesamt 5 Äquivalenzklassen aufgeteilt werden können:

- **e2Failure > 0**
  - **e2Sum - e2Failure >= 0**

    → *e2Sum* kann und sollte reduziert werden, die Methode *decreaseFactors()* wird mit dem Wert *Math.abs(e2Failure)* aufgerufen und der *output*-String auf „e2Sum needs to be decreased - Corrected factors“ gesetzt

    → **gewählte Werte im Model**: *int e2Sum = 10; int e2Correction = 5;*

  - **e2Sum - e2Failure < 0**

    → *e2Sum* sollte eigentlich reduziert werden, selbst die maximale Reduktion ergibt jedoch nicht den tatsächlichen Aufwand, deshalb wird der *output*-String auf „e2Sum needs to be decreased - Failure can't be corrected by just adjusting factors“ gesetzt

    → **gewählte Werte im Model**: *int e2Sum = 10; int e2Correction = -10;*

- **e2Failure < 0**
  - **e2Sum + Math.abs(e2Failure) <= 60**

    → *e2Sum* kann und sollte erhöht werden, die Methode increaseFactors() wird mit dem Wert *Math.abs(e2Failure)* aufgerufen und der *output*-String auf „e2Sum needs to be increased - Corrected factors“ gesetzt

    → **gewählte Werte im Model**: *int e2Sum = 5; int e2Correction = 10;*

  - **e2Sum + Math.abs(e2Failure) > 60**

    → *e2Sum* sollte eigentlich erhöht werden, selbst die maximale Erhöhung ergibt jedoch nicht den tatsächlichen Aufwand, deshalb wird der *output*-String auf „e2Sum needs to be decreased - Failure can't be corrected by just adjusting factors“ gesetzt

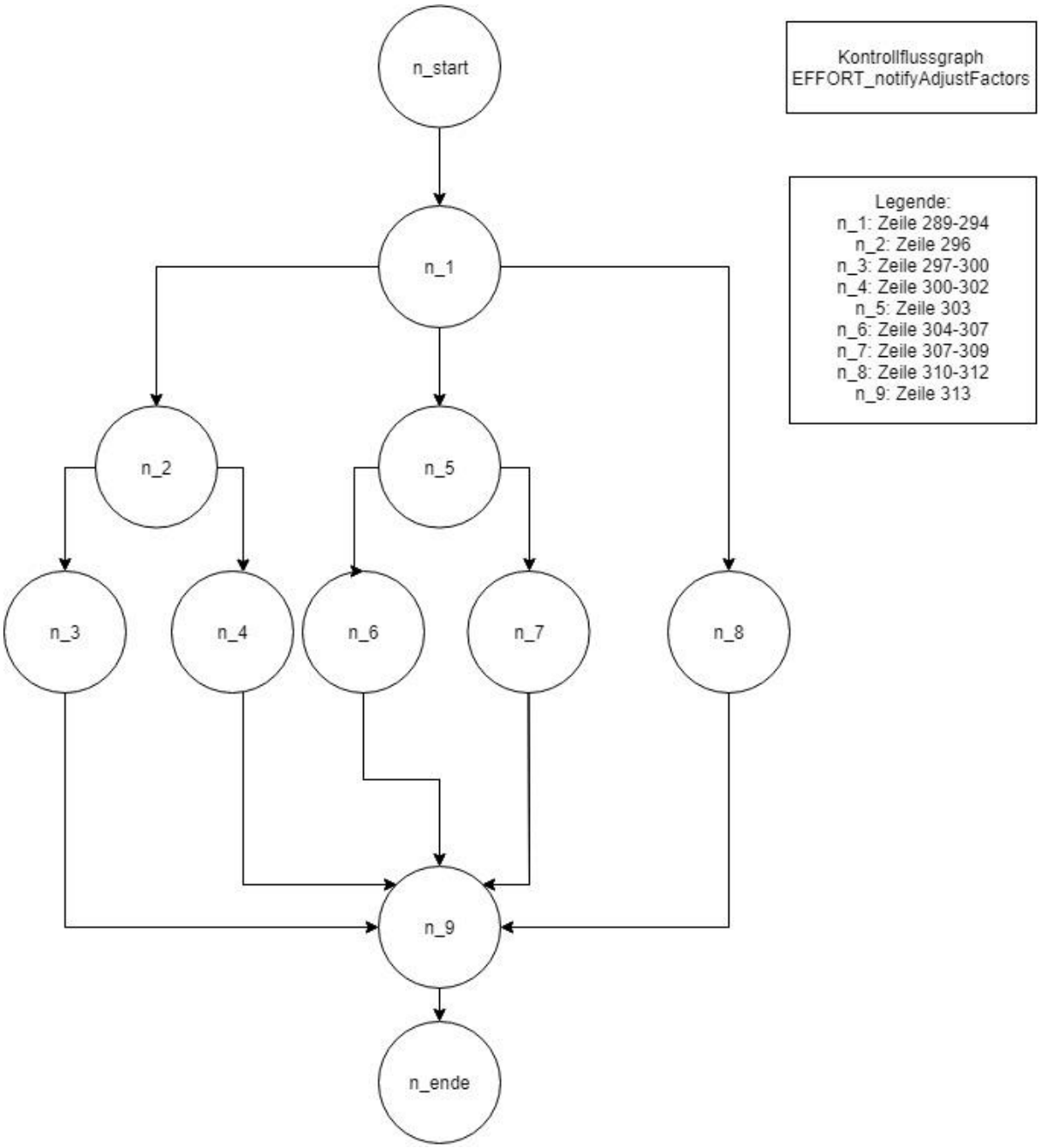
    → **gewählte Werte im Model**: *int e2Sum = 10; int e2Correction = 80;*

- **e2Failure = 0**

  → *e2Sum* muss nicht geändert werden, da der tatsächliche Aufwand bereits erreicht wurde, deshalb wird der *output*-String auf „No failure to correct“ gesetzt

  → **gewählte Werte im Model**: *int e2Sum = 10; int e2Correction = 10;*

Aus diesen 5 Äquivalenzklassen wurden nun 5 Test konstruiert, von denen jeder jeweils in einen speziellen Zweig läuft und den gesetzten Output-String mit dem erwarteten Output vergleicht. So kann gewährleistet werden, dass alle Zweige korrekt aufgerufen werden und somit die Zweigabdeckung erreicht wurde. Dies ermöglicht es dem Entwickler bereits 34 % aller möglichen Fehler zu erkennen.



Kontrollflussgraph
EFFORT_notifyAdjustFactors

Legende:
n_1: Zeile 289-294
n_2: Zeile 296
n_3: Zeile 297-300
n_4: Zeile 300-302
n_5: Zeile 303
n_6: Zeile 304-307
n_7: Zeile 307-309
n_8: Zeile 310-312
n_9: Zeile 313

increaseFactors()

Die Methode *increaseFactors()* wird mit dem Integer-Wert, um den die Faktoren erhöht werden sollen (*int increase*), als Eingabeparameter aufgerufen. Für diese Funktion lassen sich erneut 5 Äquivalenzklassen bilden:

- **increase < 0 | (e2Sum() + increase) > 60**

  → durch diese Abfrage kann verhindert werden, dass ein Methodenaufruf mit einem fälschlicherweise negativen Wert für increase Daten im Model verändert. Stattdessen wird der Fehler *IllegalArgumentException("increase out of bounds")* erzeugt.

  → gewählter Wert im Model: *int increase = -1*

- **increase >= 0**
  - **increase = 0**

    → gewählter Wert im Model: *int increase = 0*

  - **increase >= 0**
    - **factorIterator == 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

      → alle Faktoren von 0 bis 9 der Reihe nach durchgehen, bis der gewünschte Wert für *e2Sum* erreicht wurde

      → jeder Faktor wird aufgrund der Berechnung der möglichen Anpassung nur so weit verändert, dass sein Wert nach wie vor im festgesetzten Rahmen bleibt
      → nach jedem erfolgreichen Schleifendurchgang wird *increase* um 1 verringert, sodass sobald *increase* nichtmehr > 0 ist die Methode erfolgreich abgeschlossen wurde

      → gewählter Wert im Model: *int e2Sum = 60*

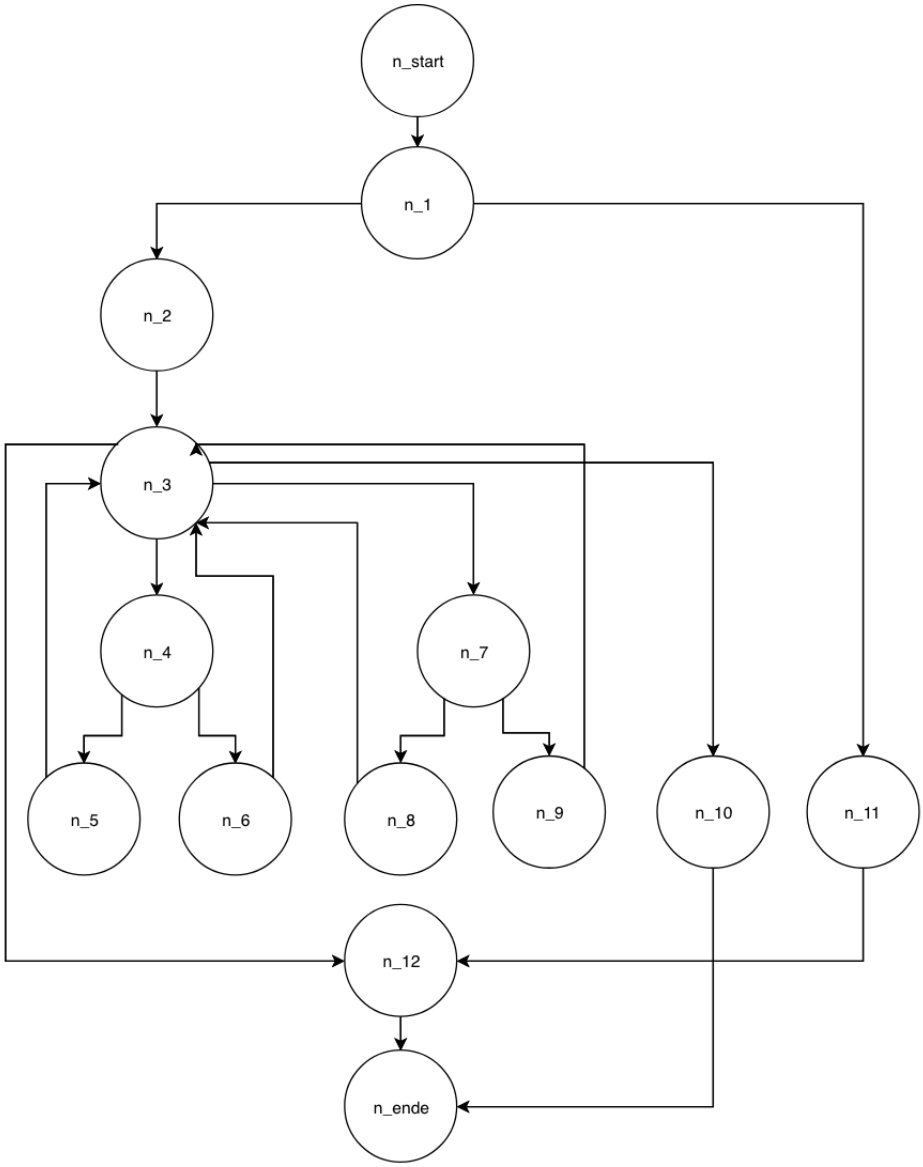
      (da 60 der maximal mögliche Wert ist, diese Methode jedoch mit Konfigurationsdaten aufgerufen wird, bei denen alle Faktoren = 0 sind, muss in diesem Fall zwangsläufig jeder Zweig durchlaufen werden um das gewünschte Ergebnis von *e2Sum* = 60 zu erhalten)

    - **factorIterator != 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

      → Dieser Zweig wird nur aufgerufen, falls der zu Anfang mit 0 initialisierte *factorIterator* außerhalb der Faktoren (0-9) liegt. Dies könnte zum Beispiel auftreten, wenn *increase* noch nicht Null erreicht hat, alle Faktoren jedoch bereits maximal erhöht wurden. Im normalen Programmverlauf sollte dies nicht auftreten, da der Wert von *e2Sum* vor Aufruf dieser Methode neu berechnet wird. Um jedoch eine Endlosschleife zu vermeiden, sollte der Wert von *e2Sum* nicht korrekt berechnet worden sein, würde in diesem Fall der Fehler *RuntimeException("factorIterator out of bounds")* auftreten und die Endlosschleife verhindert

      → gewählter Wert im Model: *int e2Sum = 10, int e2Correction = 60*

      → *int increase = 50*, alle Faktoren bereits maximal



Kontrollflussgraph
EFFORT_increaseFactors

Legende:
n_1: Zeile 323-324
n_2: Zeile 328
n_3: Zeile 329
n_4: Zeile 330-339
n_5: Zeile 340-344
n_6: Zeile 344-346
n_7: Zeile 347-350
n_8: Zeile 351-355
n_9: Zeile 355-357
n_10: Zeile 358-360
n_11: Zeile 325-328
n_12: Zeile 363

decreaseFactors()

Vergleichbar zu increaseFactors wird die Methode decreaseFactors() ebenfalls mit dem Integer-Wert, um den die Faktoren verringert werden sollen (int decrease), als Eingabeparameter aufgerufen. Für diese Funktion lassen sich erneut 5 Äquivalenzklassen bilden:

- **decrease < 0 | (e2Sum() - decrease) < 0**

  → durch diese Abfrage kann verhindert werden, dass ein Methodenaufruf mit einem fälschlicherweise negativen Wert für *decrease* Daten im Model verändert. Stattdessen wird der Fehler *IllegalArgumentException("decrease out of bounds")* erzeugt.

  → gewählter Wert im Model: *int decrease = -1*

- **decrease >= 0**
  - **decrease = 0**

    → gewählter Wert im Model: *int decrease = 0*

  - **decrease >= 0**
    - **factorIterator == 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

      → alle Faktoren von 0 bis 9 der Reihe nach durchgehen, bis der gewünschte Wert für *e2Sum* erreicht wurde

      → jeder Faktor wird aufgrund der Berechnung der möglichen Anpassung nur so weit verändert, dass sein Wert nach wie vor im festgesetzten Rahmen bleibt
      → nach jedem erfolgreichen Schleifendurchgang wird *decrease* um 1 verringert, sodass sobald *decrease* nichtmehr > 0 ist die Methode erfolgreich abgeschlossen wurde

      → gewählter Wert im Model: *int decrease = 60*

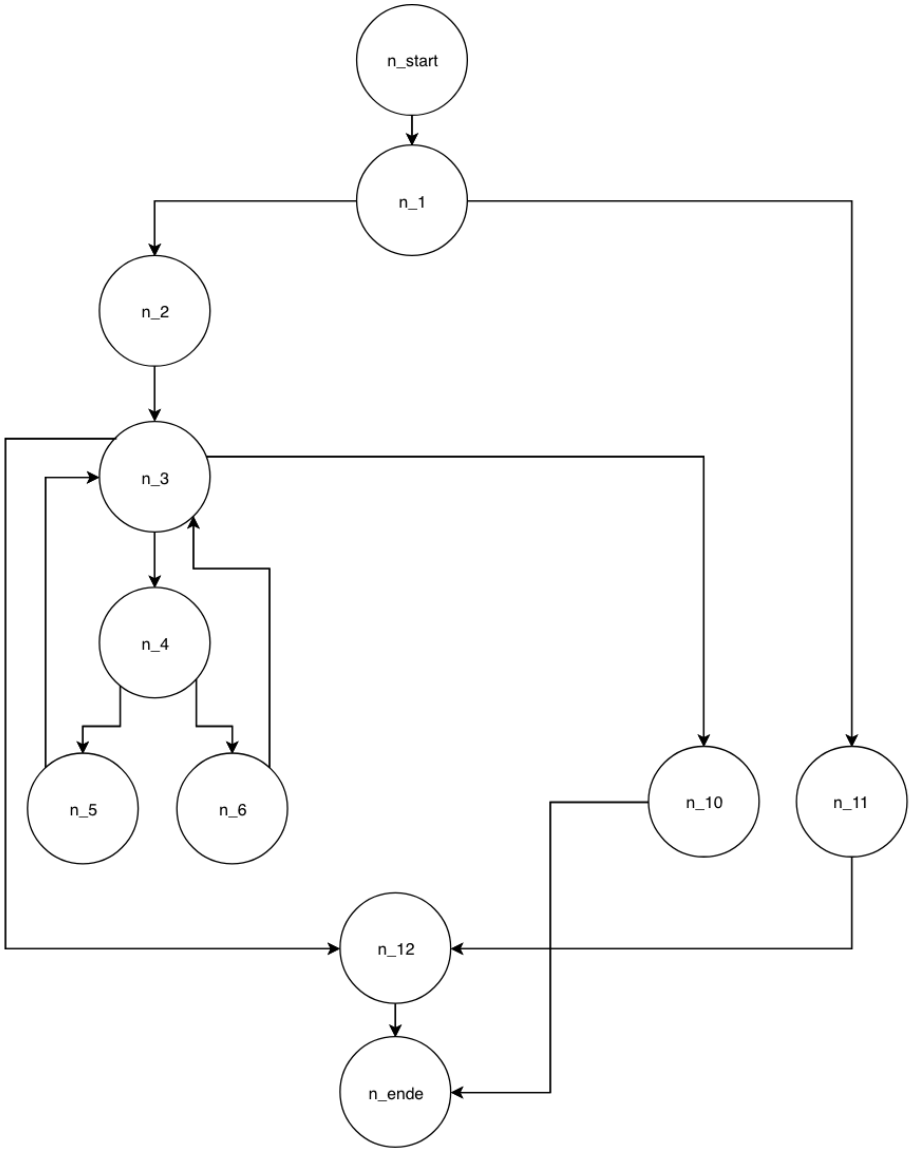
      (da 60 der maximal mögliche Wert ist, diese Methode jedoch mit Konfigurationsdaten aufgerufen wird, bei denen alle Faktoren = 0 sind, muss in diesem Fall zwangsläufig jeder Zweig durchlaufen werden um das gewünschte Ergebnis von *e2Sum* = 60 zu erhalten)

    - **factorIterator != (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)**

      → Dieser Zweig wird nur aufgerufen, falls der zu Anfang mit 0 initialisierte *factorIterator* außerhalb der Faktoren (0-9) liegt. Dies könnte zum Beispiel auftreten, wenn *decrease* noch nicht Null erreicht hat, alle Faktoren jedoch bereits auf ihr Minimum verringert wurden. Im normalen Programmverlauf sollte dies nicht auftreten, da der Wert von *e2Sum* vor Aufruf dieser Methode neu berechnet wird. Um jedoch eine Endlosschleife zu vermeiden, sollte der Wert von *e2Sum* nicht korrekt berechnet worden sein, würde in diesem Fall der Fehler *RuntimeException("factorIterator out of bounds")* auftreten und die Endlosschleife verhindert

      → gewählter Wert im Model: *int e2Sum = 10, int e2Correction = 5*

      → *int decrease = 5*, alle Faktoren bereits minimal



Kontrollflussgraph
EFFORT_decreaseFactors

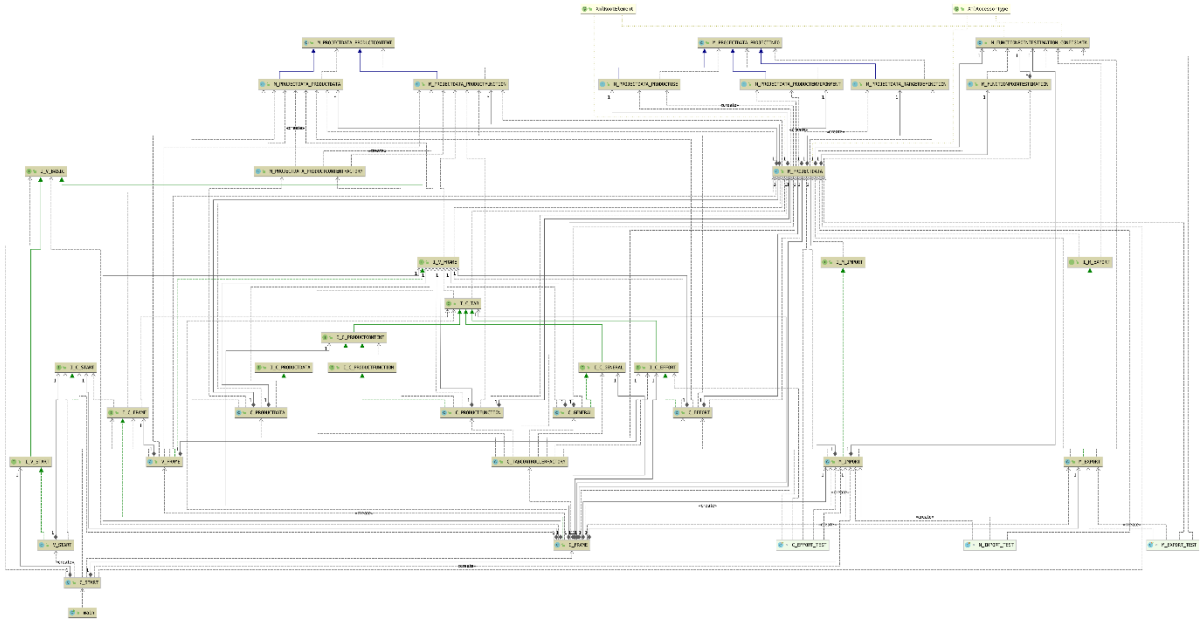
Legende:
n_1: Zeile 373-374
n_2: Zeile 378
n_3: Zeile 379
n_4: Zeile 380-390
n_5: Zeile 391-394
n_6: Zeile 394-397
n_10: Zeile 397-399
n_11: Zeile 375-378
n_12: Zeile 401

Anhang

Alle Diagramme sind in höherer Auflösung unter dem Pfad \SWEII_CASE_TOOL\Dokumentation\Diagramme zu finden

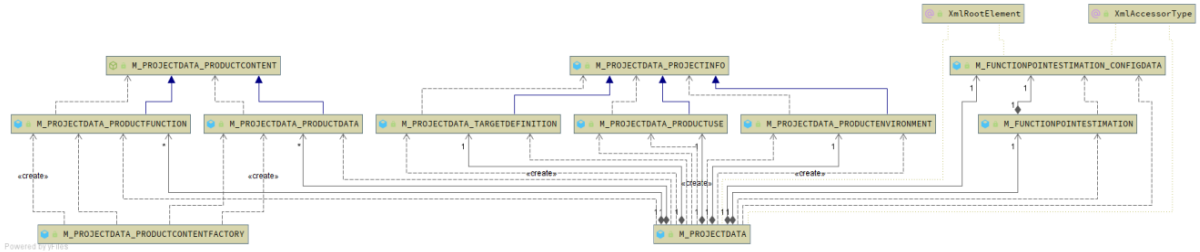
Klassendiagramme

Komplette Klassen Übersicht

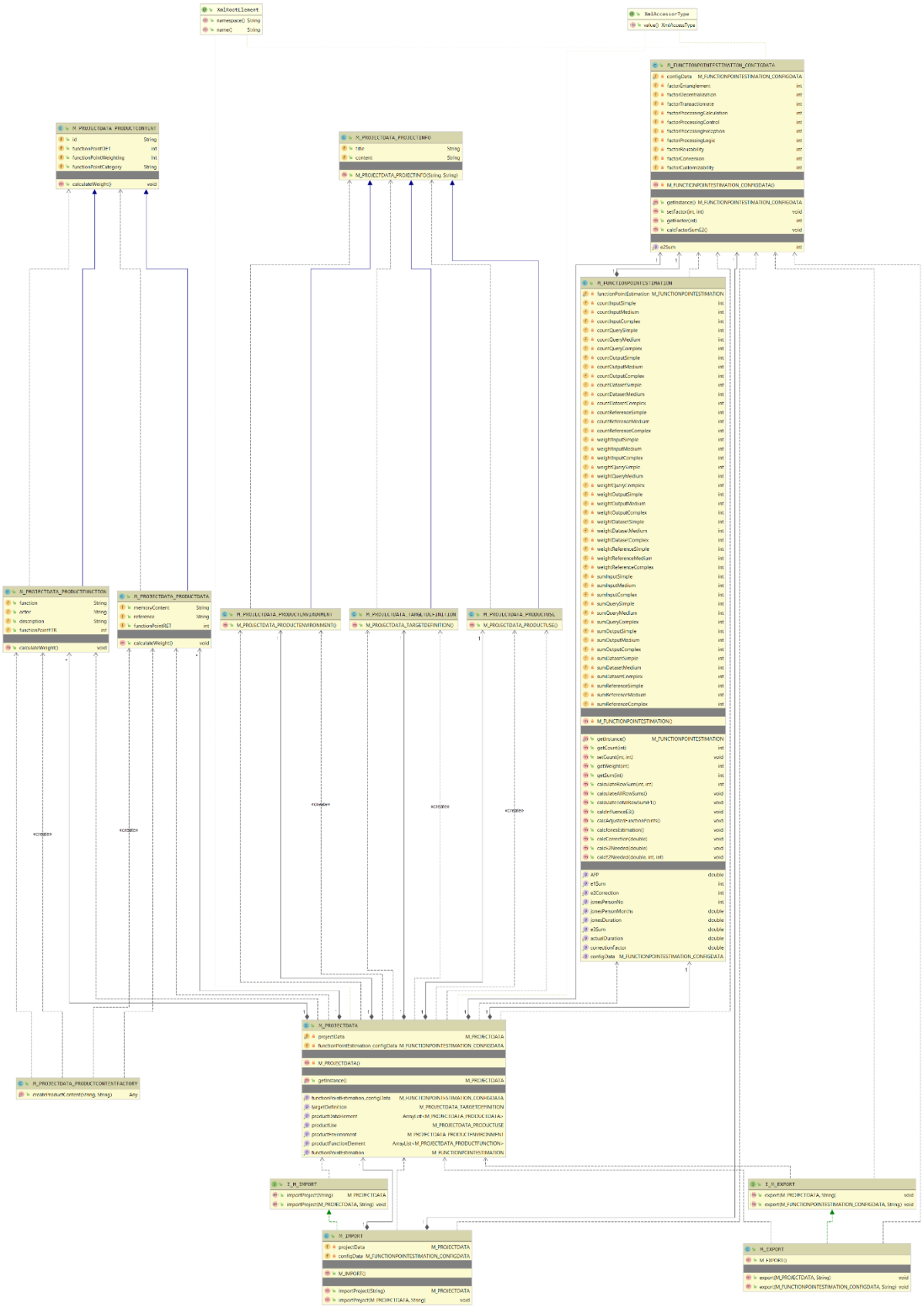


MODEL

Übersicht Kompakt

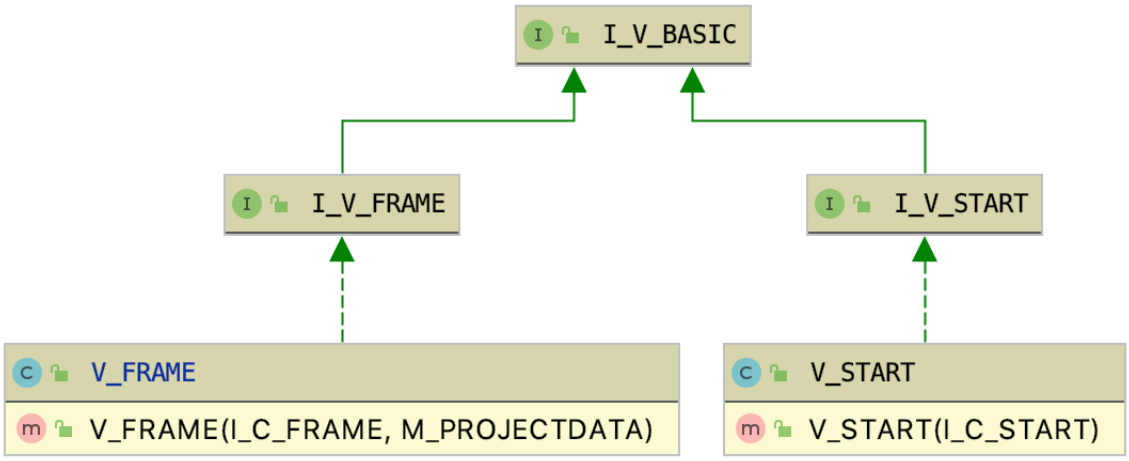


Übersicht Komplett



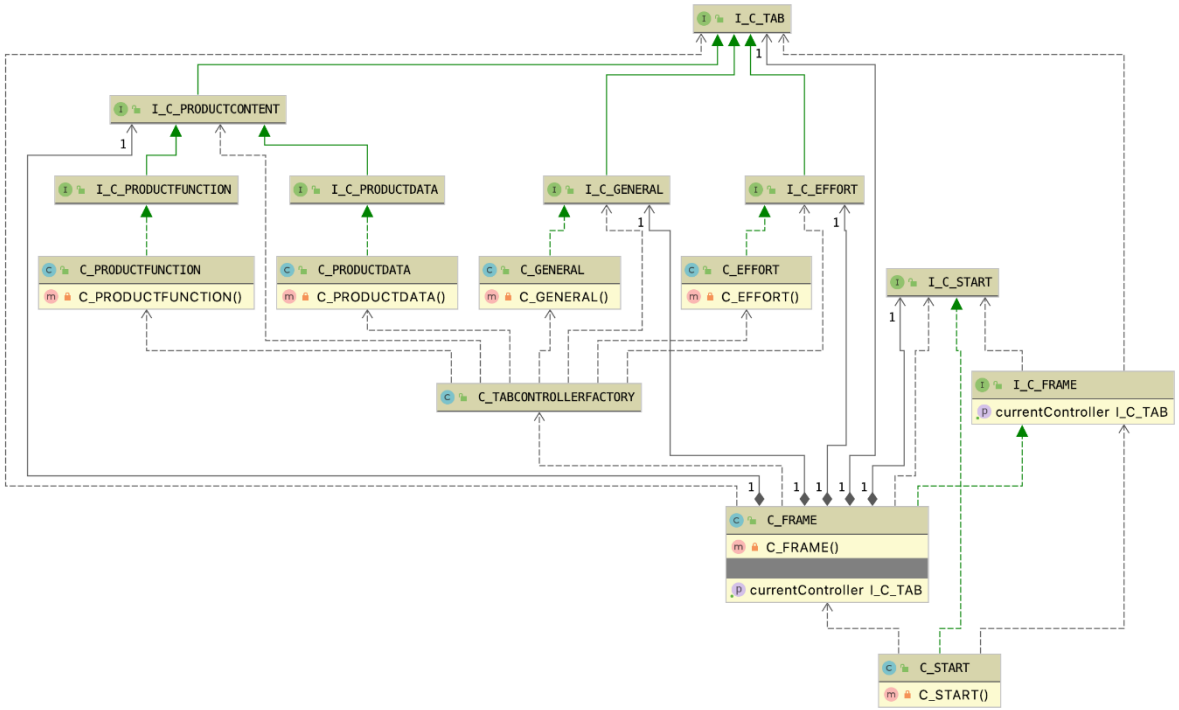
VIEW

Übersicht Kompakt

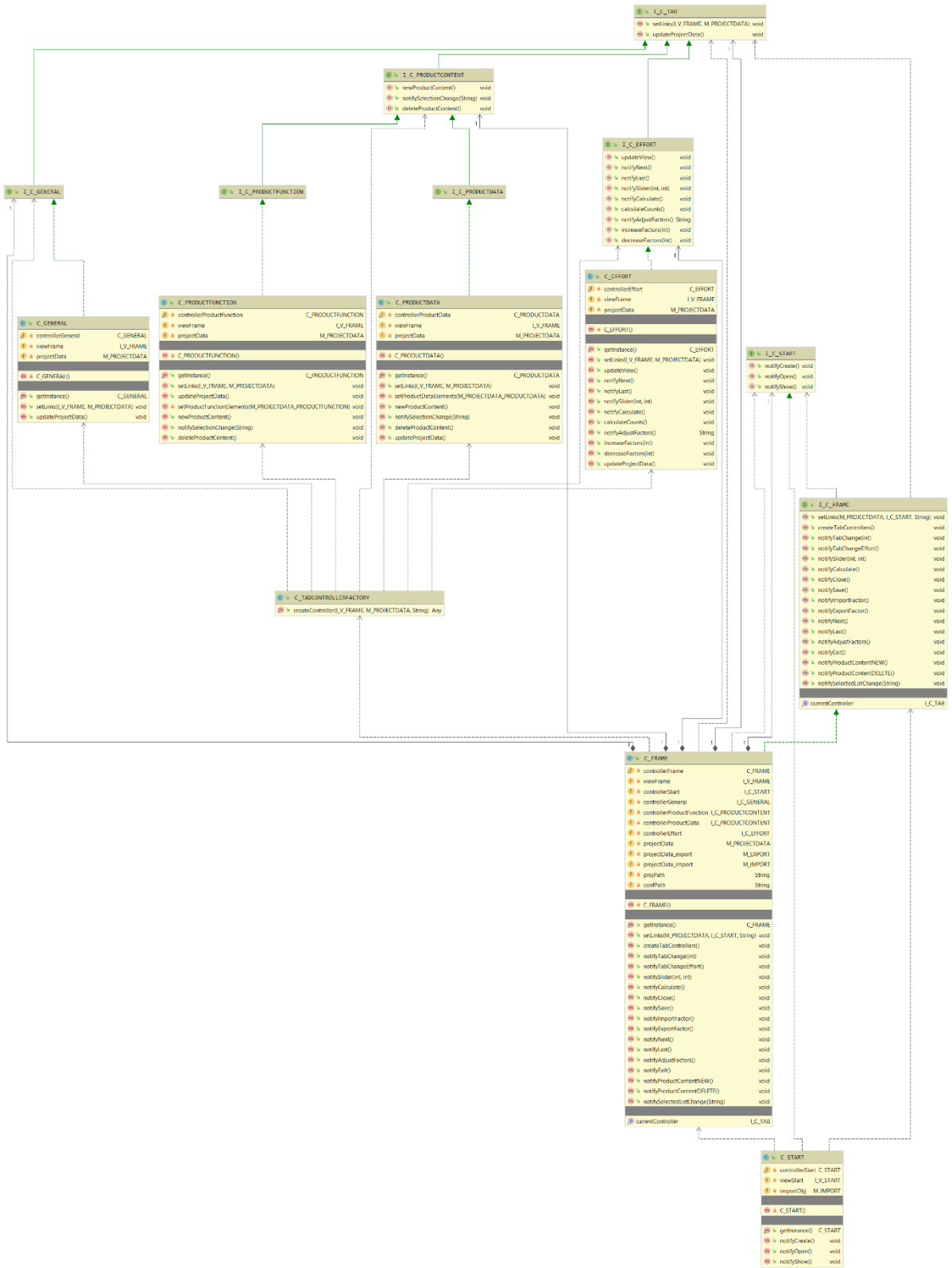


CONTROLLER

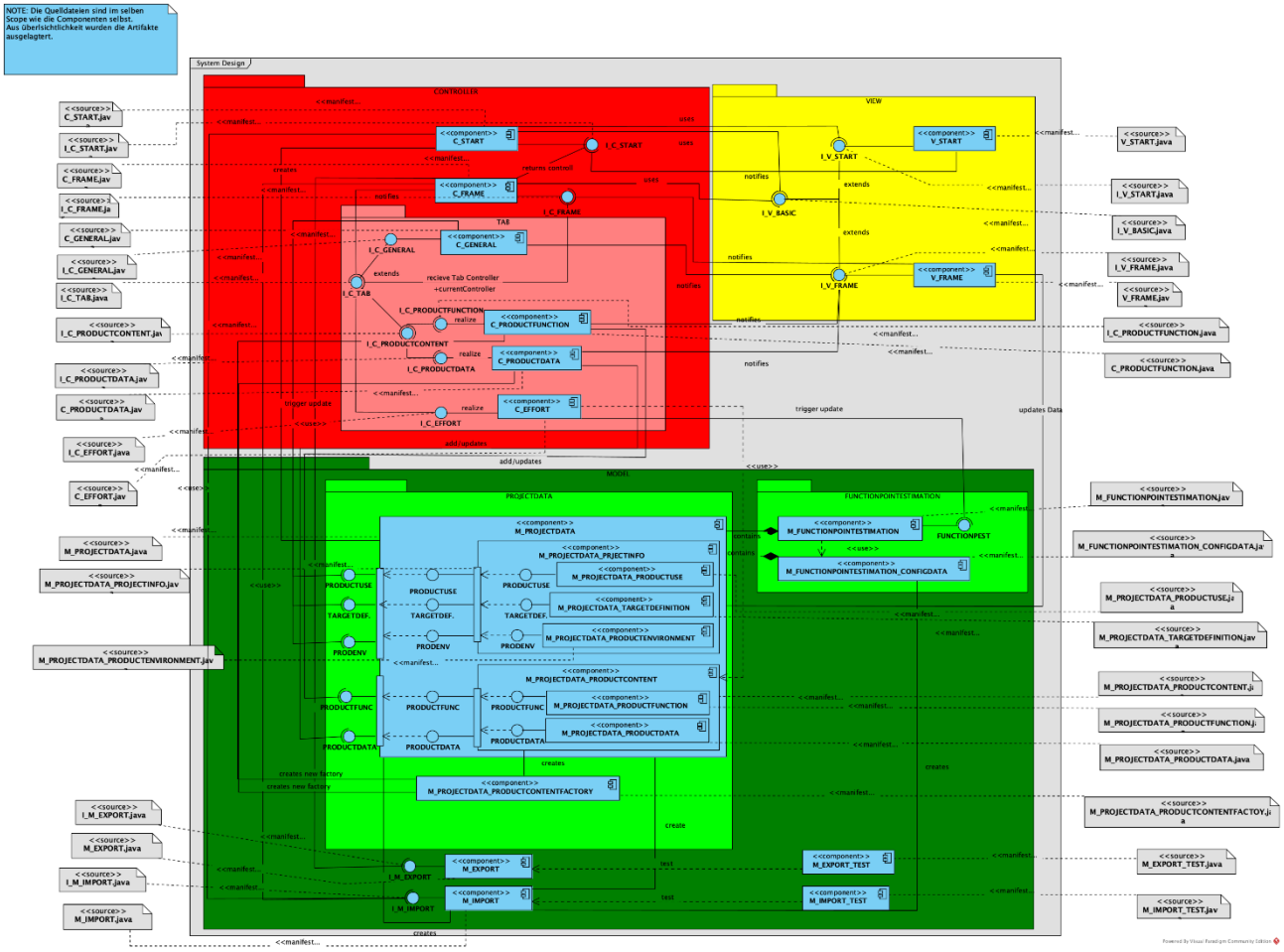
Kompakte Übersicht



Vollständige Übersicht



Komponentendiagramm



Paketdiagramm