DHBW SOFTWARE SYSTEMS GMBH

ReadMe CASET

Implementierung eines CASE-Tools

Version 1.0

16.05.2013

Sebastian Gugel

Aaron Ochs

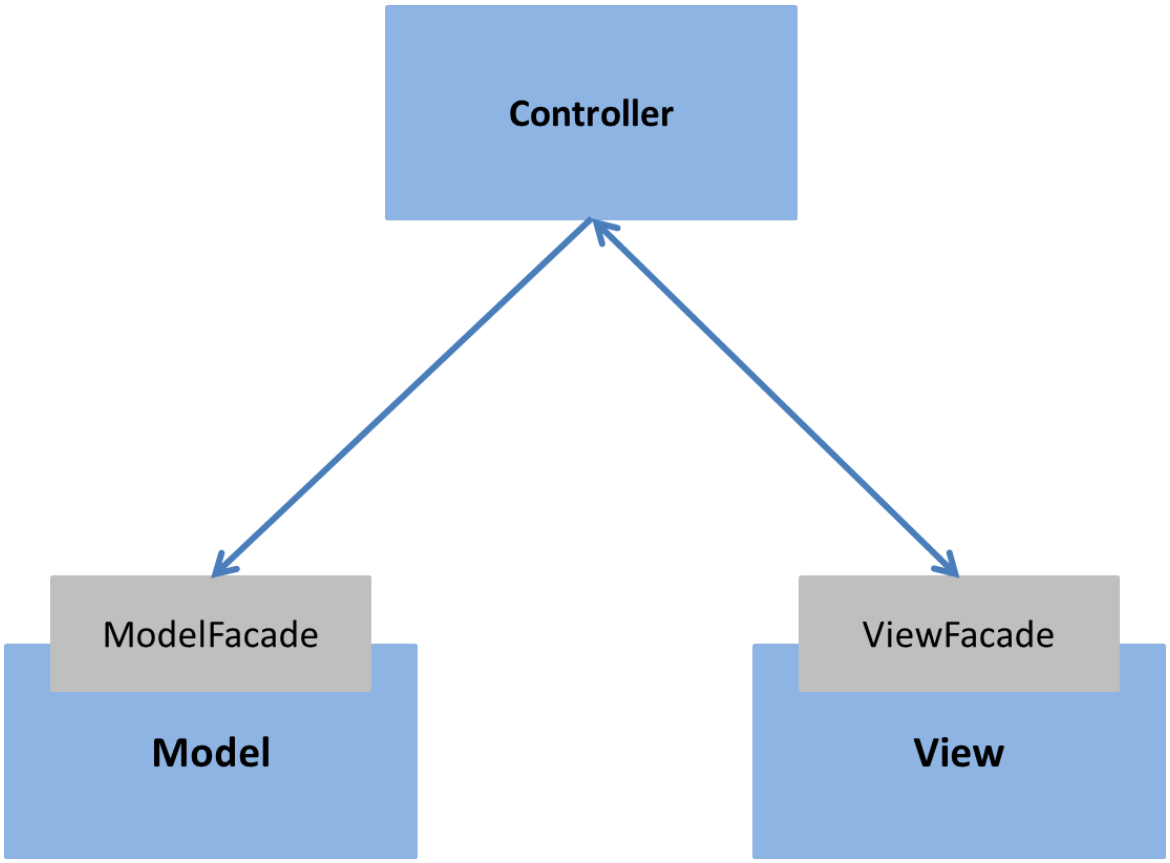
Markus Zukunft

Bitte Beachten!

Für die Ausführung des Tools wird die x86 JRE in der Version 1.7 benötigt.

Das Tool wurde unter **Eclipse Juno Service Release 2** geschrieben und mit JUnit getestet.

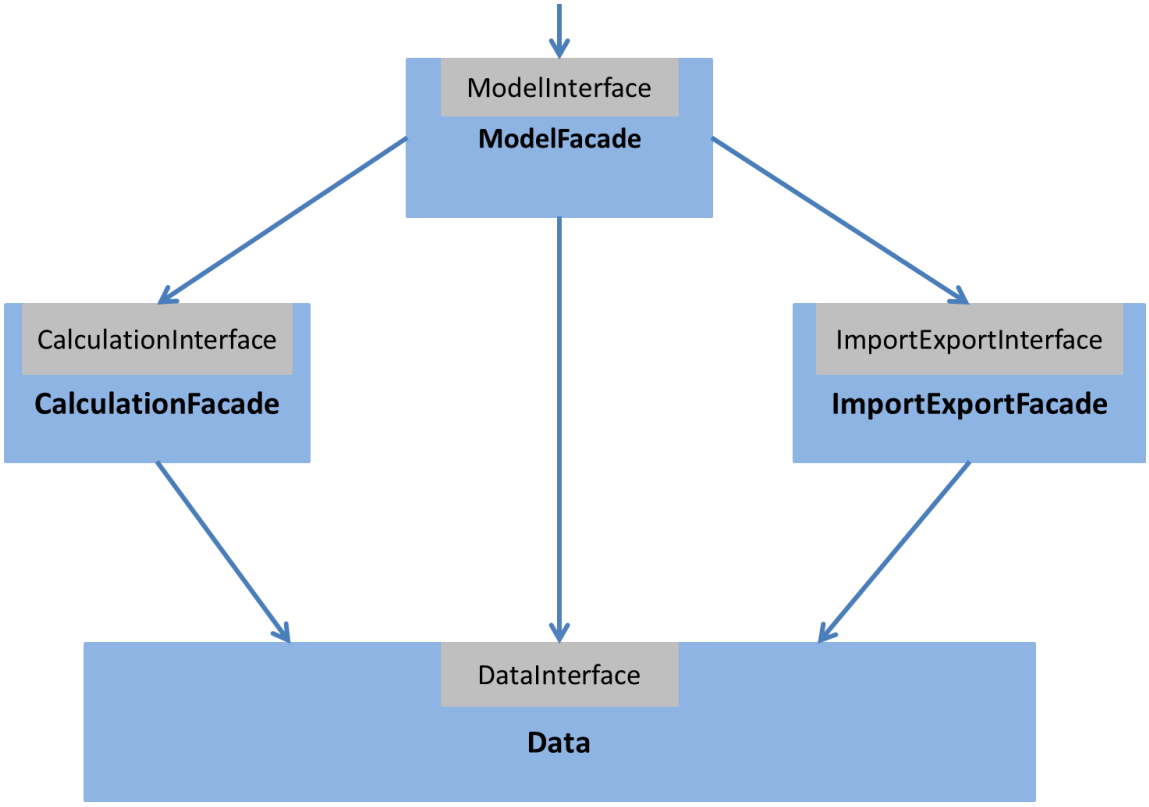
1. Gesamtstruktur



Die Gesamtstruktur von CASET ist nach dem Model-View-Controler-Pattern aufgebaut. Wie oben zu sehen besitzt das Model und die View zusätzlich eine Facade-Klasse, die die entsprechenden Schnittstellen (`ModelInterface`, `ControlerInterface`, `ViewInterface`) implementieren. Dadurch wird die interne Struktur abstrahiert und verborgen. Die Facade-Klassen, sowie die Controller-Klasse sind jeweils als Singletons realisiert. Im Folgenden sollen die Aufgaben, sowie die einzelnen Unterstrukturen des Models, der View und des Controllers dargestellt werden.

2. Model

Das Model ist intern in einem Repository-Pattern realisiert. Hierbei gibt es verschiedene Services, wie die Berechnung und der Import/Export auf ein Repository zugreifen. Dieses besteht aus den Daten. Die einzelnen Komponenten besitzen ihre jeweiligen Schnittstellen, so dass Zugriffe von außen über die `ModelFacade` auf die entsprechenden Unterkomponenten und die Kommunikation der Komponenten untereinander realisiert werden kann. Zudem sind die Klassen `Data, CalculationFacade und ExportFacade` sind als Singletons realisiert. Im Folgenden ist die Unterstruktur des Model dargestellt.

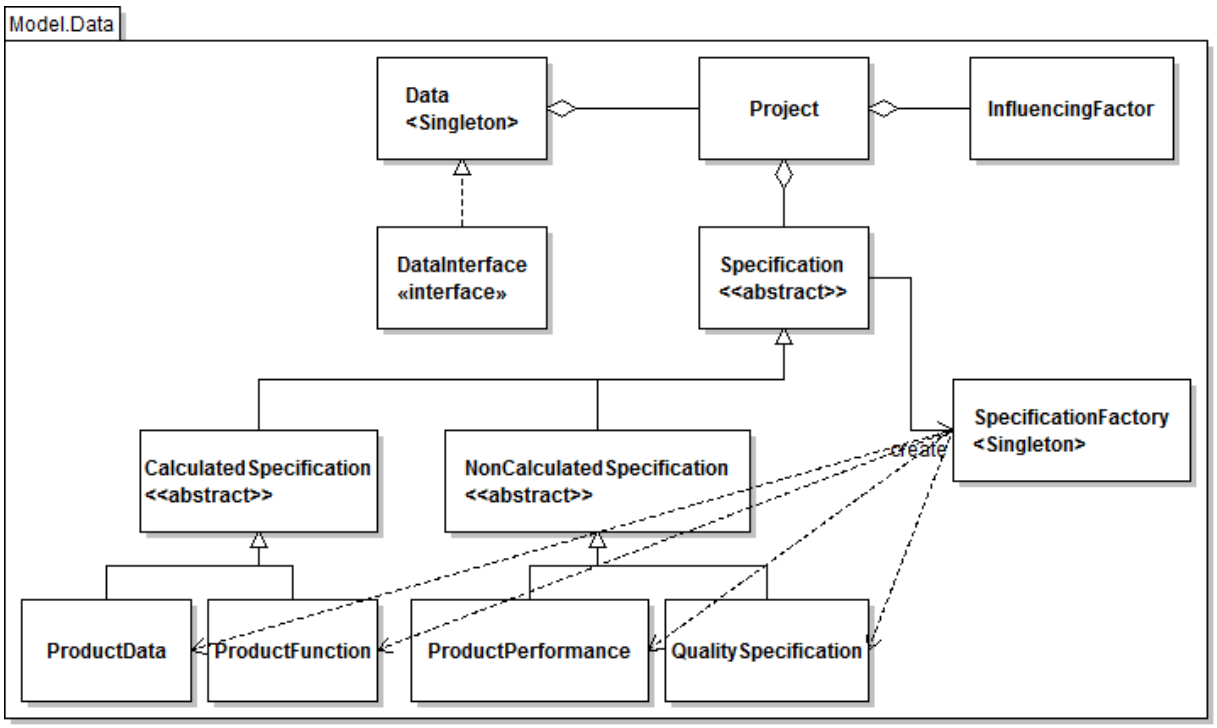


Die einzelnen Services, wie Import/Export und Calculation, sowie die Daten werden in den folgenden Unterpunkten erläutert und dargestellt.

2.1 DATA

Die Daten, die das Repository bilden, sind in untenstehendem Bild dargestellt. Die Kommunikation zu den Daten läuft über die Klasse `Data`, die das `DataInterface` implementiert. Der wichtigste Bestandteil dieser Klasse ist zudem eine Array List, welche die verschiedenen Projekte hält. Des Weiteren haben Projekte verschiedene Attribute, sowie Anforderungen und Einflussfaktoren. Für die Function-Point-Berechnung werden die `CalculatedSpecifications` verwendet. Zudem kann zwischen einer gewichteten Berechnung durch die `InfluencingFactors` und einer ungewichteten Berechnung gewählt werden.

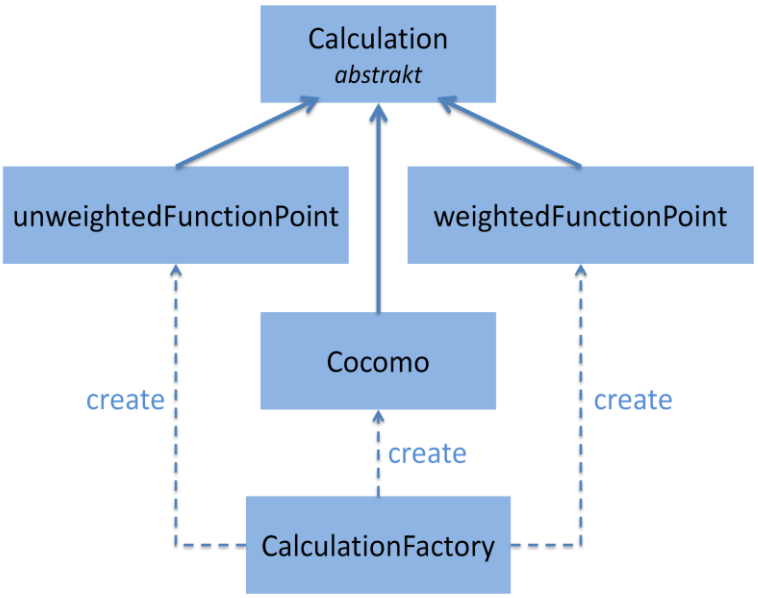
Für die Erzeugung der verschiedenen Anforderungen dient eine `SpecificationFactory`. In der Abbildung nicht dargestellt sind die Enums, die für die Datenstruktur verwendet werden. Diese sind: `FunctionCategoryEnum`, `GlossaryFieldEnum`, `InfluencingFactorTypeEnum`, `ProjectFieldEnum`, `SpecificationClassificationEnum`, `SpecificationFieldEnum`, `SpecificationTypeEnum`. Diese sind in den JavaDoc-Kommentaren erklärt.



2.2 Calculation

Die Calculation folgt dem Muster der Factory Methode. Über die `CalculationFactory` kann in der `CalculationFacade`, der Implementierung des `CalculationInterfaces`, einfach zur Laufzeit die gewünschte Art der Aufwandsberechnung erzeugt werden. Zur Auswahl stehen:

- *Cocomo*
- *WeightedFunctionPoint*: FunctionPoint mit Einflussfaktoren
- *UnweightedFunctionPoint*: FunctionPoint ohne Einflussfaktoren



Die `CalculationFacade` ist darüber hinaus ein Singleton um mehrere Instanzen zu verhindern.

Um das Ergebnis einer Aufwandsschätzung zurück geben zu können, wird eine Containerklasse `CalculationResult` verwendet. In dieser können die Personenmonate und die Entwicklungszeit gespeichert werden. Da sie die Ergebnisspeicherung bei *Cocomo* und *FunctionPoint* leicht unterscheidet, ist `CalculationResult` abstrakt und kann in Form von `CocomoResult` und `FunctionPointResult` instanziert werden.

Da sich die Berechnung der *weighted* und *unweighted FunctionPoints* in vielerlei Hinsicht überschneidet, erben beide von einer Superklasse `FunctionPoint`, die gemeinsame Funktionen zur Verfügung stellt.

2. 3 Export

Das Caset – Tool bietet die Möglichkeit bestimmte Daten eines Projektes in Form einer XML – Datei auf einem Datenträger zu speichern.

Um die Implementierung anderer Exportformate zu vereinfachen, wird eine Factory verwendet. Somit können für andere Dateitypen einfach Subklassen der abstrakten Klasse Export geschrieben werden.
Aller verfügbaren Dateitypen werden in dem Enum ExportType festgehalten und können über dieses in ExportFactory ausgewählt werden.

Für den Zugriff auf den Export gibt es ein Interface, welches durch die Klasse ExportFacade implementiert wird. Diese ist außerdem als Singelton realisiert.

3. Controller

Der Controller besteht aus einer `ControllerFacade`, die zentral alle benötigten Methoden zur Verfügung stellt, sowie einigen Listenern.

Die `ControllerFacade` ist ein Singleton um zu verhindern, dass mehr als ein Controller vorhanden ist.

Generell teilt der Controller, wenn in der Oberfläche eine Änderung, wie das Ändern des Textes einer Textbox eintritt, dem Modell mit, dass die zugehörige Eigenschaft geändert werden soll. Die benötigten Daten bekommt er dabei von der View.

Tritt eine solche Änderung auf, wird eine Methode in einem Listener ausgeführt. Gui-Elemente, also Teile der View, bekommen Instanzen dieser Listener vom Controller.

In einem Listener und so auch im Controller wird also bestimmt, was passiert, wenn ein Ereignis, wie das ändern des Textes einer Textbox, eintritt.

Der Controller beinhaltet auch einige flexible Listener, wie zum Beispiel `ModifyFieldListener`. Im Gegensatz zu den meisten einfacheren Listenern ist er eine eigene Klasse und keine anonyme Klasse, bei der nur eine Methode implementiert wird. Er ist eine eigene Klasse mit einem Feld für den Typ der Projekteigenschaft, zu der er gehört, enthält.

Wird der instanziiert und einer Textbox zugewiesen, muss dieser Typ übergeben werden. So kann dieser Listener beispielsweise für jede Projekteigenschaft benutzt werden, statt einzelne Listener für jede Eigenschaft zu implementieren.

Generell teilt der Controller, wenn in der Oberfläche eine Änderung, wie das Ändern des Textes einer Textbox eintritt, dem Modell mit, dass die zugehörige Eigenschaft geändert werden soll. Die benötigten Daten bekommt er dabei von der View.

4. VIEW

Die View hat, genauso wie Controller und Model eine Facade, die alle benötigten Methoden bereitstellt.

Die `ViewFacade` hat das Applikationsfenster `MainWindow` und alle anderen Oberflächenelemente sind Kinder des MainWindows oder anderer Elemente. Die View ist also in einer Baumstruktur aufgebaut.

Die `ViewFacade` muss dem Controller die Möglichkeit geben, Daten aus der View zu bekommen und Daten in der View zu verändern. Dies passiert über die Methoden `setData` und `getData`, denen über Enums mitgeteilt wird, was gebraucht wird oder geändert werden soll.

Genau wie die anderen Facade-Klassen ist die ViewFacade ein Singleton, um zu verhindern, dass mehr als eine View vorhanden ist.