Thema:

Entwicklung und Implementierung eines Frameworks zur Anbindung von Requirements Engineering Werkzeugen an ein Variantenmanagementsystem

Studienarbeit

Arbeitsgruppe Wirtschaftsinformatik

Themensteller: Dr. Danilo Beuche

Betreuer: Prof. Dr. rer. pol. habil. Hans-Knud Arndt

Vorgelegt von: Sebastian Rosenkranz

Abgabetermin: 30.04.06

Inhaltsverzeichnis

Abbildungsverzeichnis

Verwendete Abkürzungen und Akronyme

OLE	Object Linking and Embedding
TCP	Transmission Control Protocol
IP	Internet Protocol
DXL	Doors eXtension Language
XML	Extensible Markup Language
API	Application Programming Interface
LOC	Lines Of Code
MLOC	Methods Lines Of Code

1 Variantenmanagment für Requirements

Dieses Dokument stellt die Entwicklung und Implementierung einer Schnittstelle für pure::variants dar. Da die Integration von externen Datenquellen, z.B. von Softwarewerkzeugen, zu einem gewissen Maß redundante Entwicklung erfordert, ergibt sich die Möglichkeit mehrfache Implementierungen zu reduzieren indem eine Schnittstelle geschaffen wird, die möglichst alle Funktionen enthält die zur Anbindung von externen Datenquellen benötigt werden. Um die Benutzbarkeit der Schnittstelle zur Anbindung zu beweisen, werden die Softwarewerkzeuge Telelogic DOORS und Borland Caliber RM über die zu schaffende Schnittstelle integriert.

1.1 Motivation

Durch ständige Zunahme des Wettbewerbsdrucks auf die am Markt tätigen Unternehmen, ergeben sich immer komplexere Anforderungen an Produkte. Es erfolgt eine stetige Verkürzung des Produktlebenszyklus, also des Zeitraumes von der Entwicklung bis zum Ausscheiden des Produktes aus dem Markt. Beispielsweise lag die Vertriebsdauer eines Golf I bei 9 Jahren (1974-1983), beim Golf IV sank die Vertriebsdauer bereits auf 7 Jahre (1997-2003). Abbildung 1 zeigt den Vergleich des Produktlebenszyklus verschiedener Automobile in Jahren.

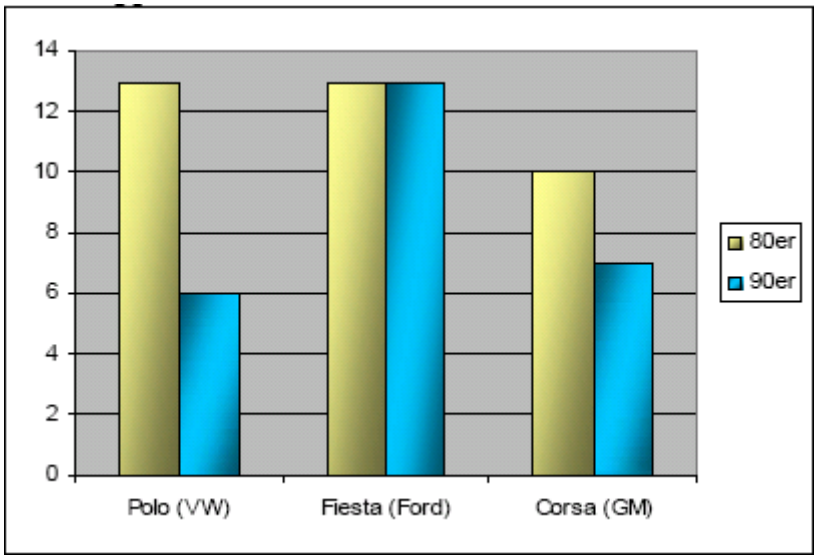


Abbildung 1: Vergleich Produktlebenszyklus ausgewählter Marken

Quelle: [Nied2000] S. 196

Erklärungen für die Verkürzung von Produktlebenszyklen, die auch für andere Branchen typisch sind, liefert das Marketing mit dem Erklärungsansatz der „Indiviualisierung“. Entsprechend diesem Ansatz steigt die Produktdifferenzierung in gesättigten Märkten, wie sie in Westeuropa zu finden sind, an.

Aus der Verkürzung des Produktlebenszyklus bei gleichzeitig steigender Anzahl von Produktvarianten ergeben sich neue Herausforderungen an das Management mit dieser Menge an Produktvarianten mitzuhalten. Der Begriff 'Management von Anforderungen', 'Anforderungsmanagement' oder engl. 'requirements management' wird folgend im Sinne von [Leff2000] als ein
„ ...systematischer Ansatz, Anforderungen des Systems zu erheben, zu organisieren und zu dokumentieren und einen Prozess, der eine Übereinkunft zwischen dem Kunden und

dem Projektteam etabliert und pflegt für die sich ändernden Anforderungen an ein System.“ verwendet.
Eine Anforderung oder engl. requirement im Anforderungsmanagement wird dabei als eine elementare Eigenschaft verstanden, die ein Produkt oder eine Dienstleistung erfüllen muss. Anforderungen werden vom Kunden bzw. vom Zielsegment des Marktes an ein Produkt[1] definiert. Durch das Management eines Unternehmens können weitere Anforderungen hinzukommen, zum Beispiel niedrige Kosten des Produktes oder hoher Wiedererkennungswert. Eine Anforderung an ein Produkt, für die eine Umsetzung existiert, wird als Merkmal eines Produktes bezeichnet.

Die Definition von Anforderungen ist der Ausgangspunkt für die Gestaltung eines Produktes. Fehler, die in dieser frühen Phase der Entwicklung gemacht werden, verursachen in späteren Phasen unnötigen Mehraufwand und damit höhere Kosten. Die unzureichende Festlegung von Anforderungen ist einer der Hauptgründe für das Scheitern von Projekten. vgl. [Eng2004]. Im Gegenschluss ist eine möglichst exakte Definition von Anforderungen Ausgangspunkt für ein erfolgreiches Projekt. Zur Dokumentation von Anforderungen existieren viele Möglichkeiten u.a. Requirements Engineering Werkzeuge. Diese Werkzeuge sind auf das Anforderungsmanagement spezialisierte Software mit wichtigen Funktionen, wie etwa Versionsverwaltung, Änderungsmanagement, etc. Telelogic DOORS, Borland Caliber RM und pure::variants werden später als solche vorgestellt. Die Merkmale eines Produktes werden mit dieser Software in Merkmalsmodellen engl. requirement documents verwaltet.

Ein Merkmalsmodell wird im Management in der Regel nicht für ein einzelnes Produkt, sondern für alle ähnlichen Produkte aus einer Produktfamilie, definiert. Die Menge von ähnlichen Produkten, die Varianten, die aus einer einheitlichen Menge von Bestandteilen konfiguriert sind, werden als Produktlinie bezeichnet. Entsprechend sind Produktvarianten durch kleine Unterschiede gekennzeichnet, etwa der Ausstattungstiefe von Mobiltelefonen oder zusätzlichen Sicherheitspaketen in Automobilen. Die Wissenschaft die sich mit der Konfiguration, der Bildung von Varianten, befasst wird als „Produktlinienmanagement“ oder „Product Line Management“ bezeichnet. Handelt es sich bei einer Produktlinie um den Bereich Software, wird von „Product Line Engineering“ gesprochen. Das Product Line Engineering wird als Bestandteil des Software Engineering angesehen.

Eine wichtige Aufgabe des Anforderungsmanagementes ist es Konflikte zwischen Produktbestandteilen zu erkennen, die eine gültige, funktionierende Produktkonfiguration verhindern. Manuelles Testen von Konfigurationen erfordert hohen Aufwand und verhindert eine große Anzahl kunden - und marktspezifisch differenzierte Produkten. Das wiederum führt zu einer Erhöhung der Kosten und wirkt sich nachteilig auf die Wettbewerbssituation eines Unternehmen aus.

Jede Variante, sei es nun Software oder nicht, besteht aus einer Menge von Merkmalen (engl.: feature). Die Menge der Merkmale werden in Merkmalsdiagrammen bzw. Merkmalsmodellen[2] dargestellt. Merkmalsmodelle sind Struktogramme, die als Strukturelemente Merkmale enthalten. Zwischen einzelnen Merkmalen können Beziehungen existieren. Beispielsweise beschreibt eine angenommene Beziehung, dass eine Variante Automobil Wolf nicht gültig ist, wenn das Merkmal Sportfahrwerk ohne das Merkmal Scheibenbremsen konfiguriert ist. Ein Merkmalsmodell stellt den allgemeinen Aufbau einer Produktlinie dar. Aus einem Merkmalsmodell können durch

1 Produkt wird synonym zu Dienstleistung verwendet
2 engl. Featuremodel

den Prozess der Konfiguration konkreter Produktvarianten entstehen. Während der Konfiguration einer Variante sind Beziehungen zwischen den Merkmalen zu beachten.

Abbildung 2 zeigt die einzelnen Schritte des Prozesses der Variantenbildung. Im ersten Schritt werden die Merkmale, die sich aus den Produktanforderungen ergeben in die Variante übernommen. Die Variante wird anschliessend auf ihre Gültigkeit zu den Beziehungen des Merkmalsmodelles hin überprüft. Ergibt sich eine technisch gültige Variante, die den Vorgaben des Merkmalsmodelles entspricht, ist abschliessend die Variante mit ihren Eigenschaften gegen die Produktanforderung zu prüfen. Erfüllt die Variante alle Anforderungen ist der Prozess abgeschlossen.

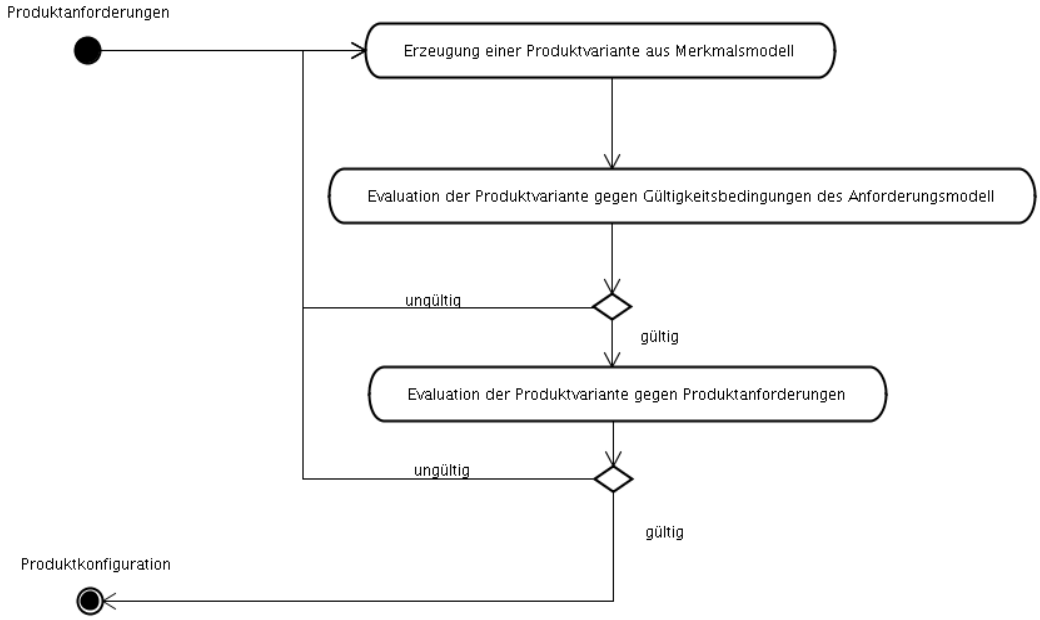


Abbildung 2: Prozess der Variantenbildung

1.2 Telelogic DOORS

DOORS[3] ist eine Softwarelösung für das Management von Anforderungen und ist nach eigenen Angaben der derzeitige Marktführer im Bereich Requirement Management. Der Schwerpunkt der Software liegt in der Modellierung von „ ...Geschäftszielen, Kundenanforderungen und Vorschriften... “ (vgl. www.telelogic.de, Stand März 2006) von Geschäftsprozessen. Alle folgenden Angaben beziehen sich auf die derzeit aktuellste Version DOORS 8.0.

3 Dynamic Object Oriented Requirements System

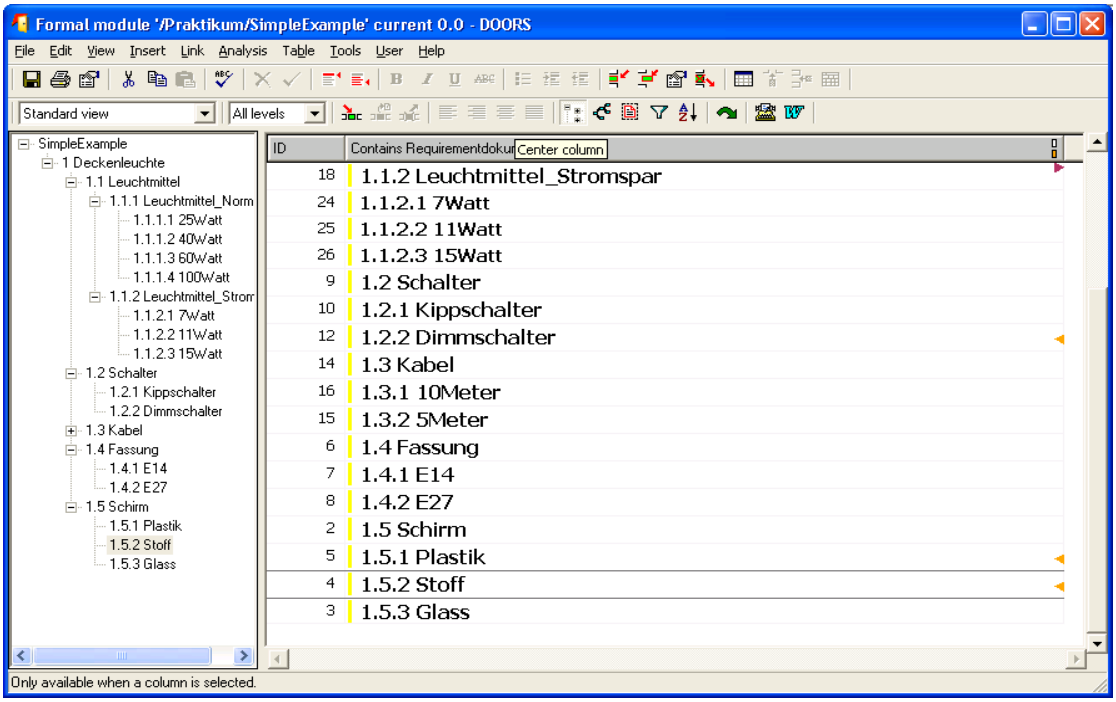


Abbildung 3: Anforderungsmodell einer Deckenleuchte in DOORS

Die Informationstruktur von Telelogic DOORS besteht aus Verzeichnissen und Projekten, wobei Projekte in Verzeichnissen und umgekehrt liegen können. Ein Projektverzeichnis ist dabei ein besonderes Verzeichnis , das alle Informationen eines Projektes enthält.
Innerhalb von Projekten können Module erstellt werden. In DOORS wird zwischen drei verschiedenen Modulen unterschieden:

- Formale Module
- Link Module
- Descriptive Module

Formale Module[4] enthalten hierarchisch strukturierte Objekte[5]. In Abbildung 3 ist das allgemeine Modell von Deckenleuchten in Telelogic DOORS nachgebildet. Jedes Objekt besitzt eine Menge von Attributen mit denen das Merkmal modelliert wird. Die Menge der Attribute wird für ein Module definiert und ist für alle Objekte des Modules gleich. Beziehungen zwischen Objekten und externen Informationen werden mit Hilfe von Links modelliert. Links werden in Link Modulen gespeichert und werden logisch einem Objekt zugeordnet. In Descriptiven Modulen, beschreibenden Modulen, werden zusätzliche Informationen abgespeichert, z.B. Grafiken und Kopien die weiterführende Informationen zu einem Informationselement (Projekt, Module, Object) in DOORS enthalten. Descriptive Module sind optional.

Die Bildung einer konkreten Variante, also einer Lampe in unserem Beispiel, muss durch manuelle Bearbeitung generiert werden. Eine rechnergestützte Evaluierung einer Variante ist in DOORS nicht möglich.

DOORS unterstützt verschiedene Benutzergruppen und Rechtemanagement. Es gibt 4 Typen von Benutzer:

4 Bezeichnung für ein Requirementdokument in Telelogic DOORS
5 Stellt eine einzelnes Merkmal in Telelogic DOORS dar.

Benutzer Typ	*Rechte*
Standard	Arbeiten mit DOORS Daten
Projekt Manager	• Wie Standard • Benutzergruppen erstellen • Archivierung • Partitionierung
Database Manager	• Wie Projekt Manager • Projektverwaltung • Benutzerverwalten • Datenbankverwaltung

Custom	• Wie Standard
	• frei definierbare Rechte aus:
	  • Benutzergruppenverwaltung
	  • Archivierung
	  • Partinionierung
	  • Projektverwaltung
	  • Benutzerverwaltung
	  • Datenbankverwaltung

Tabelle 1: Benutzerrechte in DOORS

Telelogic DOORS verfügt über 3 Schnittstellen, die grafische Benutzeroberfläche, Kommunikation über 'Object Linking and Embedding' (OLE) und eine TCP/IP Kommunikation. Da die grafische Benutzeroberfläche nicht zur Anbindung von pure::variants geeignet ist, wird sie nicht weiter betrachtet.
Die zweite Möglichkeit ist die Kommunikation über OLE Objekte. OLE ist eine Microsoft Windows proprietäre Technologie deren Benutzung mit der Einschränkung der Plattformunabhängigkeit verbunden ist. Das DOORS OLE Objekt kann benutzt werden, wenn pure::variants und Telelogic DOORS auf dem gleichen Rechner ausgeführt werden jedoch nicht, wenn der Rechner auf dem DOORS ausgeführt wird, nicht identisch ist mit dem Rechner auf dem pure::variants ausgeführt wird. Das führt zum Verlust der Möglichkeit von verteilter Anwendung auf verschiedenen Rechnern. Die Verwendung des OLE Objekt stellt für die Anbindung von pure::variants eine unzureichende Möglichkeit dar. Die Kommunikation via TCP/IP Socket ist die dritte Möglichkeit der Kommunikation zwischen pure::variants und DOORS. Die Technologie gewährleistet Plattformunabhängigkeit und Netzwerk gestütze Kommunikation.
Telelogic DOORS verfügt über die Sprache DOORS eXtentension Language – DXL. DXL ist eine Programmiersprache die sich an C und C++ anlehnt. Mit Hilfe von DXL ist es möglich auf die internen Daten zu zugreifen und wird im Hauptteil der Arbeit genauer erläutert, da DXL zur Anbindung von pure::variants benutzt wird.

1.3 Borland Caliber RM

Bei der für die Anbindung verwendeten Version, welche folgend beschrieben wird, handelt es sich um Version 2005 / 7.0. Caliber RM ist eine Softwarelösung die das Management bei „...betriebswirtschaftlichen, technischen, funktionalen und operationalen...“ (vgl. www.borland.de, Stand März 2006) Anforderungen unterstützt. Der Schwerpunkt der Softwarelösung wird von Borland für den Bereich Softwareentwicklung festgelegt.

Borland Caliber RM basiert wie Telelogic DOORS auf einer Datenbank. Die Informationseinheiten sind Projekte, die linear unter dem abstrakten Wurzelverzeichnis angelegt sind. Im Gegensatz zu DOORS existieren keine Verzeichnisse mit denen Projekte gegliedert werden können. Projekte enthalten als Wurzelelemente Requirementtypen. Requirementtypen definieren den Typ der Anforderungen[6], die Attributmenge und Gliederungsinformationen. Anforderungen werden hierarchisch unter dem ihnen entsprechendem Requirementtype gegliedert. Die Abbildung 4 zeigt das Modell einer Lampenproduktfamilie in Borland Caliber RM.

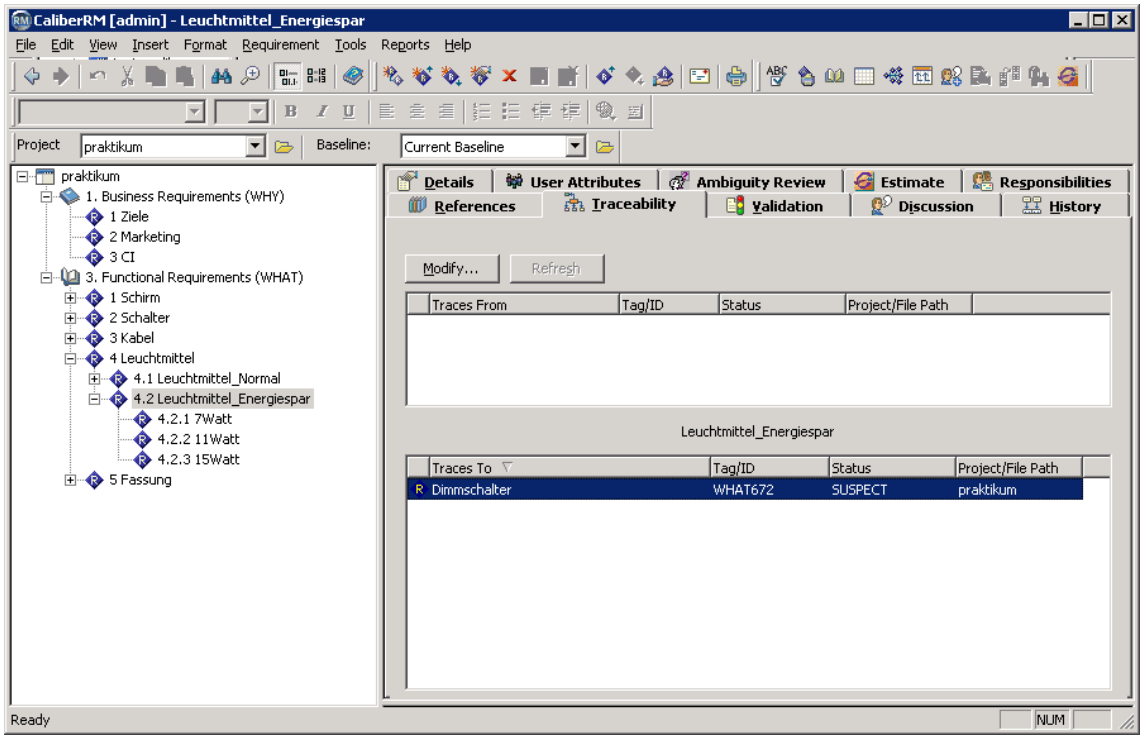


Abbildung 4: Anforderungsmodell einer Lampenfamilie in Borland CaliberRM

Im linken Bereich ist die hierarchische Anordnung der Requirementtypen und Requirements zu sehen, im rechten Bereich befinden sich die für die aktuelle Selektion zugehörigen Attribute und Einstellungen.

Abhängigkeiten zwischen Anforderungen werden in CaliberRM durch Traces modelliert. Ein Trace kann als Ziel jede andere Anforderung (im gleichen oder anderen Projekten), Tests, Softwarekompontenten[7] und Konfigurationsobjekten besitzen. Zusätzliche Ziele für Traces können durch Plugins, die CaliberRM erweitern, bereitgestellt werden.

Caliber RM verfügt nicht, wie Telelogic DOORS, über die Möglichkeit eine rechnergestütze Evaluierung von Varianten durchzuführen.

Borland Caliber RM verfügt über ein Java Application Programming Interface. Die Java Schnittstelle erleichtert die Anbindung von pure::variants, da keine neuen Programmiersprachenkenntnisse erworben werden müssen.

6 z.B. qualitative, funktionale,.. Anforderungen
7 Klassen, Packages

1.4 pure::variants

Pure::variants ist eine Softwarelösung die sich auf das Management von Varianten, die auf einer gemeinsamen Basis aufbauen, spezialisiert hat. Pure::variants unterscheidet zwischen 3 Modellen:

- Featuremodellen
- Komponentenmodellen
- Variantenmodellen

In diesem Softwarewerkzeug werden Anforderungen in Featuremodellen verwaltet. Die Merkmale, also die konkreten Umsetzungen einer Anforderung, werden in Komponentenmodellen verwaltet. Die konfigurierten Varianten werden in Variantenmodellen gespeichert. Es ergibt sich entsprechend eine Trennung zwischen Anforderungen und der Realisierung von Anfoderungen, den Merkmalen. In Abbildung 3 ist das Beispiel eines Featuremodelles einer Deckenleuchte dargestellt. Es sind alle Anforderungen enthalten, die eine Deckenleuchte aus der Produktfamilie besitzen kann. Einzelne Merkmale sind mit Beziehungen zu anderen Merkmalen gekoppelt, die im Prozess der Variantenbildung beachtet werden müssen. In Abbildung 3 ist beispielsweise die Konfliktbeziehung mit roten Pfeilen dargestellt. Entsprechend darf eine Deckenleuchte mit Plastikschirm nicht mit mehr als 60Watt Leuchtmittel ausgestattet sein, da sonst die technische Sicherheit nicht gewährleistet ist.

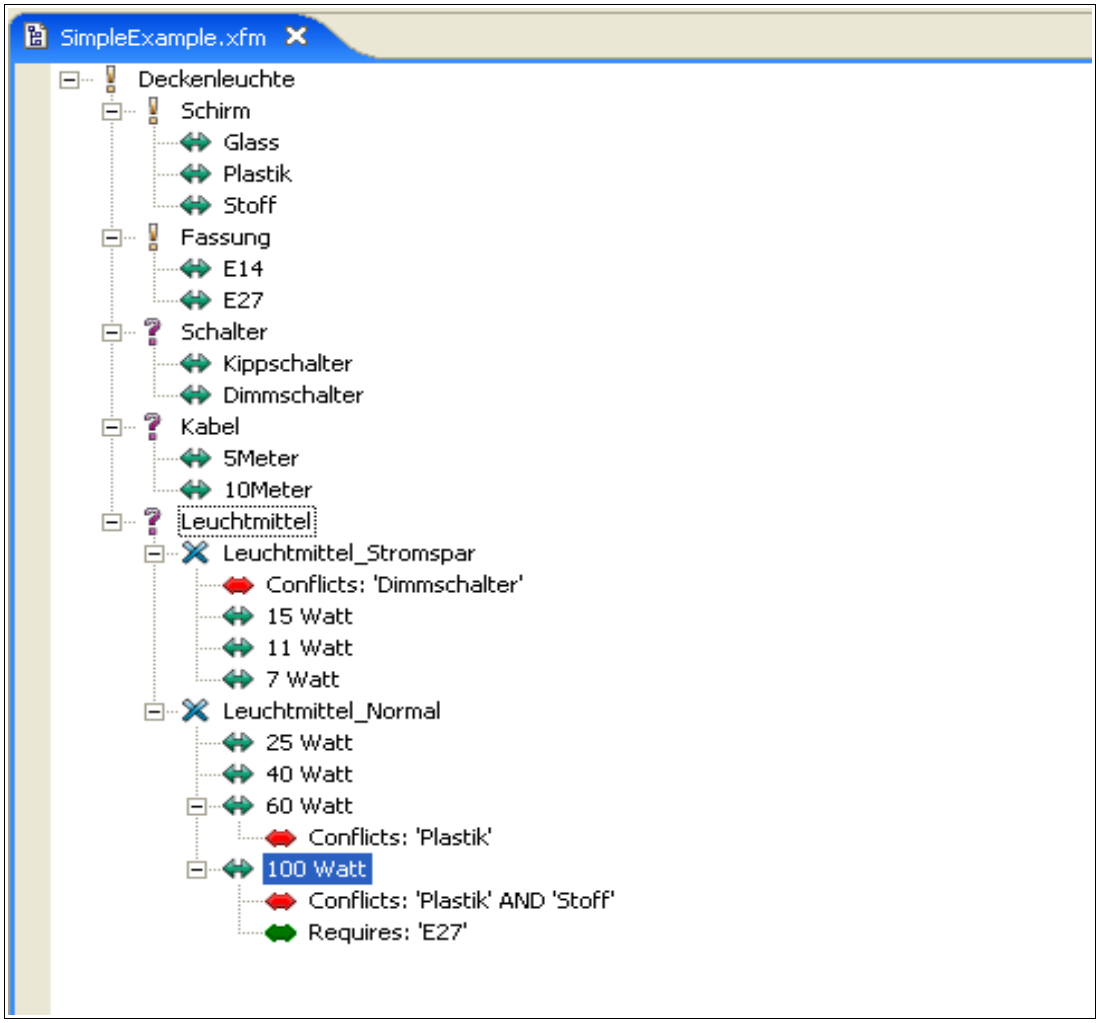


Abbildung 5: Beispiel eines Merkals für eine Deckenleuchte

Die Icons[8] in Abbildung 5 vor den einzelnen Anfoderungen zeigen die definierte Variabilität der Anforderung. Beispielsweise ist der 'Schirm' der Lampe als notwendig definiert. Entsprechend muss eine Variante genau ein Merkmal 'Schirm' haben, dagegen kann einen konkrete Lampe mit oder ohne Leuchtmittel konfiguriert werden.

Aus Merkmalsmodellen lassen sich je nach Anzahl und Variabilität der Merkmale[9] verschiedene Varianten konfigurieren. Die Anzahl der Varianten die sich aus dem in Abbildung 2 dargestellten Modell bilden lassen beträgt ohne Betrachtung der Gültigkeit 288 verschiedene theoretisch mögliche Lampen[10]. Es ist davon auszugehen dass die in der Praxis verwendete Modelle, um ein vielfaches komplexer sind.

1.5 Zusammenfassung

Telelogic DOORS und Borland Caliber RM sind in derzeitig verfügbaren Versionen nicht in der Lage den Benutzer bei der Evaluierung von Varianten durch automatische Auswertung und ggf. selbständiges Auflösen von Konflikten und Abhängigkeiten zwischen Merkmalen zu unterstützen. Eine Lösung für dieses Problem besteht darin, die Informationen die für die erfolgreiche Evaluierung notwendig sind aus den Softwarewerkzeugen in pure::variants Merkmalsmodelle zu importieren. Aus den Merkmalsmodellen können Varianten erzeugt und rechnergestützt evaluiert werden. Dabei werden Konflikte zwischen Anforderungen der Varianten herausgefunden. Sollte der interne pure::variants Prologinterpreter eine eindeutige Lösung für einen Konflikt finden werden diese Probleme automatische korrigiert. Der Benutzer muss ausschließlich Konflikte in den Varianten lösen, für die keine maschinelle Lösung gefunden werden kann. Ein Export von Varianten in das Softwarewerkzeug stellt sicher, dass die Infomationen über Varianten auch außerhalb von pure::variants verfügbar sind.
Abbildung 6 verdeutlicht abschliessend den oben beschriebenen Vorgang.

8 Die vollständige Liste der Icons befindet sich im Anhang
9 Merkmale in einem pure::variants Merkmalsmodell werden als Elemente bezeichnet
10 nach A. van Deursen, siehe Anhang [Deur2002]

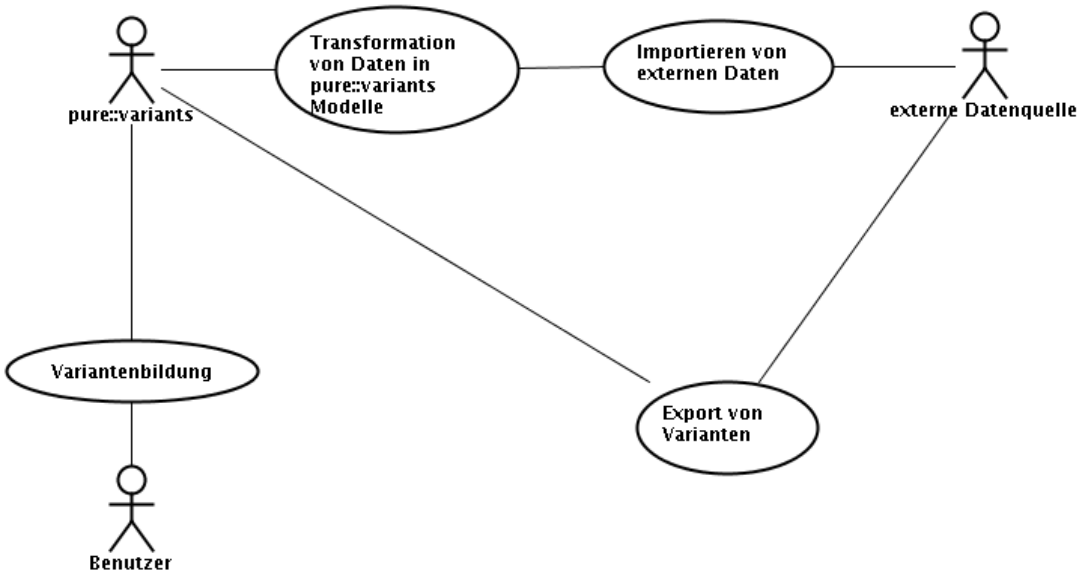


Abbildung 6: UML Darstellung Import Export von externen Daten

Die Integration von externen Datenquellen[11] an pure::variants erfordert den Import von externen Daten und die Abbildung auf das Datenmodell von pure::variants, sowie den Export von Varianten in die externe Datenquelle.

Abbildung 7 zeigt exemplarisch die verschiedenen Komponenten, die grundsätzlich erforderlich sind, eine Intergration zu implementieren. Einige Komponenten werden für jede Integration einer neuen Datenquelle redundant implementiert. Die Komponenten „Datenspeicherung“, der „Aufbau von pure::variants Merkmalsmodellen“, die „Speicherung“ von Modellen, die „Speicherung von Metainformationen“, sowie die „Variantendatenextraktion“ sind davon betroffen.

11 Damit sind in diesem Zusammenhang insbesondere Requirement Engineering Werkzeuge gemeint.

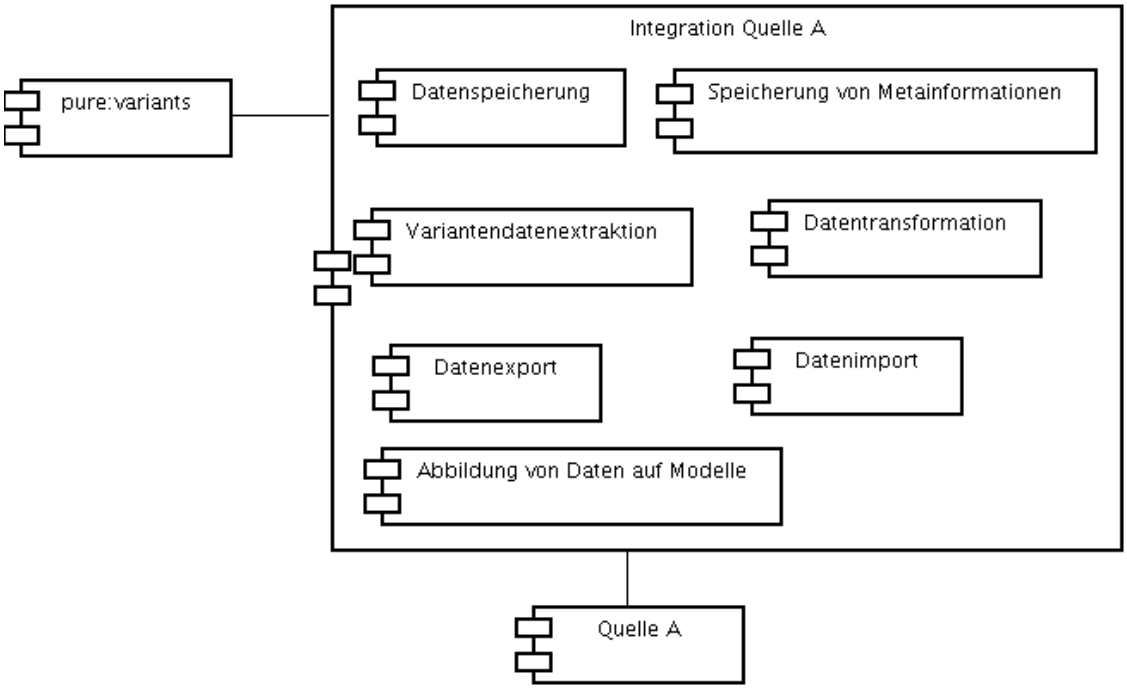


Abbildung 7: Integration einer externen Datenquelle ohne Framework

Abbildung 8 zeigt die Anbindung von externen Datenquellen unter Benutzung des Framework. Dabei sind alle Komponenten, die unabhängig von der externen Datenquelle sind, in das Framework verlagert. Alle Anbindungen, die das Framework benutzen, müssen nur noch die Komponenten „Datenexport“, „Datenimport“ und „Datentransformation“ implementieren. Diese Komponenten übernehmen die Kommunikation zwischen der externen Datenquelle und dem Framework.

Auf die Erklärung und Implementierungen der einzelnen Komponenten wird im hinteren Teil der Praktikumsarbeit eingegangen.

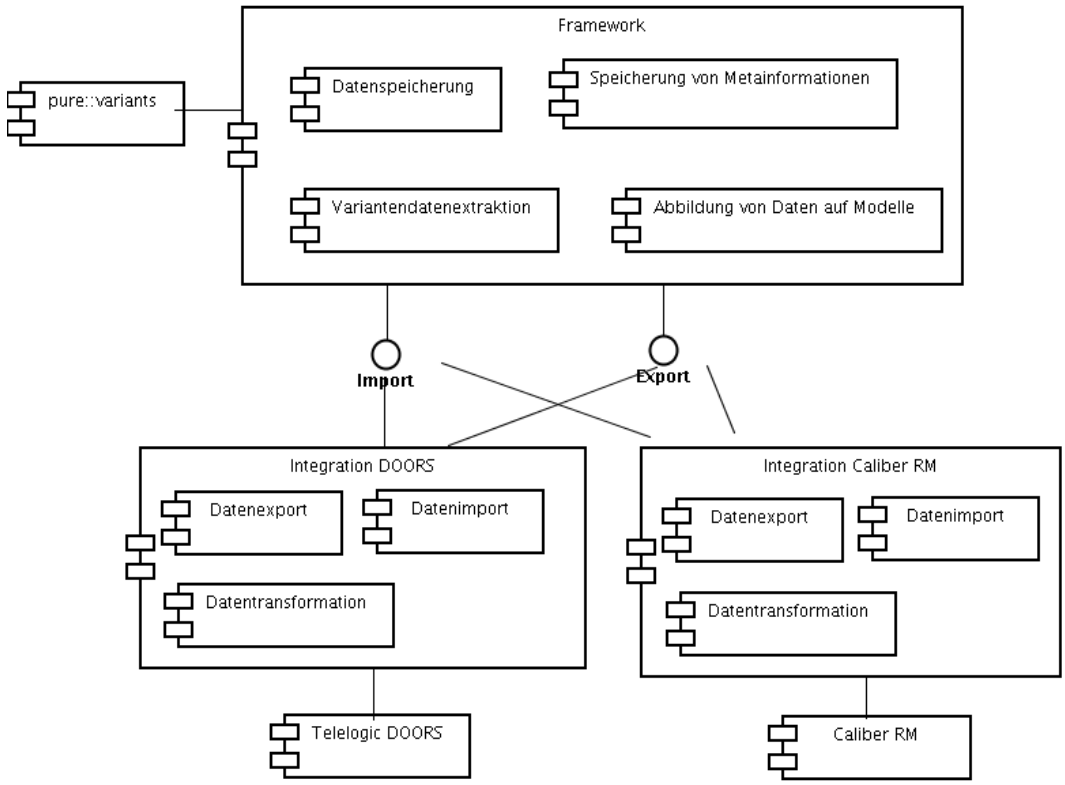


Abbildung 8: Integration von externen Datenquellen mit Framework

Bereits an Abbildung 7 und 8 lässt sich erkennen, dass der Aufwand der Integration weiterer Datenquellen an pure::variants durch ein Framework mit dessen Funktionalität reduziert werden kann. Erwartete Vorteile sind:

- geringerer Umfang des Quellcodes pro Integration,
- dadurch geringere Fehleranzahl da pro 1000 Zeilen Quellcode ca. 3 Fehler vorhanden sind. (vgl. [Du2003])
- geringerer Testaufwand
- einheitlicher Aufbau der Modelle durch Benutzung des gleichen Algorithmus

2 Umsetzung

Im folgenden Kapitel wird die konkrete Umsetzung der Anforderungen dargestellt. Es werden zunächst anhand von einfachen Pseudocode und Sequenzdiagrammen, die allgemeinen Abläufe eines Imports und Exports von Daten zwischen pure::variants und einer externen Datenquelle dargestellt. Ausgehend von dieser einfachen Sichtweise werden einzelne Funktionsgruppen definiert und in weiteren Schritten mit genauen Anforderungen beschrieben. Den Abschluss dieses Kapitels bilden Codebeispiele und ein kurzer Abriss der Probleme, die sich während der Implementierung ergaben und das theoretische Konzept stark beanspruchten.

2.1 Entwurf

Die 3 Funktionen, die durch das Framework unterstützt werden müssen, sind:

1) Import von externen Daten und Abbildung auf Merkmalsmodelle
2) Update von Merkmalsmodellen mit externen Daten
3) Export von Varianten in externe Datenquellen

2.1.1 Import

Als Import wird folgend der Prozess von der Bereitstellung der Daten aus der externen Datenquelle bis zum vollständigen Aufbau eines pure::variants Merkmalmodelles bezeichnet.

Zunächst besteht die Schwierigkeit darin einen Ausgangspunkt für die Planung eines Frameworks zu finden. Dafür werden folgende Annahmen getroffen. Es existiert keine Aussage über die Art der Daten und wie auf diese Daten zugegriffen werden kann – Das Framework soll sich für alle Arten von externen Datenquellen eignen, die Anbindung unterstützen, aber keine Einschränkung machen.

Da die externen Datenquellen unbekannt sind, kann entsprechend keine Aussage über deren Beschaffenheit getroffen werden. Fest steht, dass diese externen Daten auf pure::variants Merkmalsmodelle abgebildet werden müssen. Die Modellabstraktion[12] der Merkmalsmodelle ist fest definiert und stellt den Ansatzpunkt für die Planung eines Framework dar. Ein pure::variants Merkmalsmodell ist aus verschiedenen Bestandteilen aufgebaut, die Informationen über die Merkmale enthalten. Die Bestandteile werden folgend als Infomationsträger bezeichnet. Ein pure::variants Merkmalsmodell besteht aus folgenden Informationsträgern:

- Modellmetainformation {Author, Version, Name}
- Merkmalen
- Abhängigkeiten zwischen Merkmalen
- Attributen
- Restriktionen

Auf genau diese Informationsträger müssen sich externe Daten abbilden lassen. Dafür wird für jede Informationseinheit ein festes Interface definiert. Jedes Interface beschreibt alle Informationen, die die zugehörigen pure::variants Informationseinheit enthalten kann. Durch die Definition von festen Schnittstellen ist eine bijektive Abbildung zwischen

12 Pure::variants Modelle benutzen eine XML Datenstruktur,

pure::variants Informationsträger und zugehörigem Interface möglich. Es ist dadurch möglich den Aufbau eines pure::variants Merkmalsmodell von den externen Daten zu trennen. Der konkrete Algorithmus lässt sich allgemein im Pseudocode verfassen:

```
Programm Import
    consulModel := createConsulModel (URL, Metadata)
    consulRoot  := getRoot (consulModel)
    extRoots    := getExternalRootElements ()
    for each in extRoots do
    beginn
        newElement:= appendAsChildren(consulRoot,each
extRoots)
        call function traverseTree (newElement, each
extRoots)
        extRelations := getExternalRelations()
        appendRelations()
    end

    function appendAsChildren(consulElement ,extElement)
        newElement := create Children below consulElement
        setData from extElement to newElement
        append all attributes from extElement to newElement
        return newElement
    end function

    function traverseTree( consulElement, extElement)
        extChildren := getExternalChildren(extElement)
        for each in extChildren do
        beginn

    newEl:=appendAsChildren(consulElement,eachextChildren)
        mapp newEl with each extChildren
        call function traverseTree (neuesEl,extChildren)
        end
    end function
end programm
```

Für den Import von externen Daten ist die bisher beschriebene Funktionalität ausreichend. Für Update und Export müssen Informationsträger aus den pure::variants Modellen ihren externen Quellen zugeordnet werden können. Beispielsweise muss für einen Vergleich eines Merkmals, welches aus einer externen Quelle importiert wurde, das zugehörige Datum in der externen Datenquelle gefunden werden. Fehlt die Information zu welchen externen Daten ein Informationsträger zugeordnet ist, ist ein Vergleich nicht möglich. Es ist also notwendig für jeden Informationsträger in pure::variants eine eindeutige Referenz[13] auf die externen Daten zu speichern, die abgebildet werden.

Das Mapping von Informationsträgern muss folgende Anforderungen erfüllen:

- Eindeutige Zuordnung von Informationsträgern aus pure::variants zu dem externen Datensatz, aus dem der Informationsträger generiert wurde.

13 In der Informatik wird diese Zuordnung als Mapping bezeichnet

- Die Eindeutigkeit der Zuordnung darf nicht durch Veränderung des Informationsträgers in pure::variants verloren gehen, beispielsweise durch die Umbenennung eines Modells oder Änderung eines Attributwertes
- Die Zuordnung sollte erhalten bleiben, wenn sich das externe Datenobjekt ändert um durch Änderungen in externen Datenquellen die Zuordnung zu dem pure::variants Informationsobjekt nicht zu verlieren.

Das Finden einer eindeutigen Zuordnung für Informationsträger in pure::variants erfordert die genaue Kenntnis der externen Datenquelle. In der Praxis werden hierfür in der Regel Identifikatoren bzw. ID's vergeben, die sich als Zuordnung eignen. Da für jedes Informationsobjekt eine solche Zuordnung erforderlich ist und die Kenntnis der externen Datenquelle erfordert, um ID's zu erzeugen, wird für das Mapping ein eigenes Interface[14] durch das Framework bereitgestellt.

Als Möglichkeit der Speicherung der Mappinginformationen existieren verschiedene Möglichkeiten. Es ist zum einen möglich die Informationen in „räumlicher Nähe“ zu den Informationsträgern zu speichern. Dafür bietet sich das Merkmalsmodell selber an. Beispielsweise können Merkmale ein Attribut erhalten, was die Zuordnung auf die externe Quelle enthält. Alle Informationen, die für eine spätere Zuordnung zu externen Daten notwendig sind, sind damit immer verfügbar. Der Nachteil dieses Vorgehens ist die Änderung des Merkmalmodell durch Daten die nicht von der externen Quelle stammen.

Die zweite Möglichkeit ist die Mappinginformationen vom Modell zu trennen. Da theoretisch unbegrenzt viele Möglichkeiten der Speicherung dieser Informationen existieren, etwa die verschiedenen Verzeichnisse der Festplatte eines Computers, wird ein weiteres Interface bereitgestellt. Im Framework wird es eine Standardimplementierung geben, welche die Mappinginformationen im Modell speichert. Durch eine andere Implementierung des Interface bleibt das Framework bei der Speicherung der Mappinginformationen flexibel.
Abschliessend müssen alle Informationen gespeichert werden über:

- welche Implementierung wurde für das Mapping benutzt, bzw. welche Implementierung kann die externen Daten zu einem Informationsträger finden
- welche Implementierung wurde zum Bereitstellen der externen Daten verwendet
- welche Implementierung wurde zum Speichern der beiden ersten Werte benutzt

2.1.2 Update

Das Aktualisieren von Merkmalsmodellen die aus exteren Datenquellen hervorgegangen sind, stellt eine grundlegende Anforderung an das Framework dar. Beispielsweise wurde ein Requirementsdokument aus Telelogic DOORS in pure::variants importiert. Aufgrund von geänderten Merkmalen einer Produktfamilie wurde auch das zugehörige Requirementsdokument erweitert. Um diese Änderungen in pure::variants zu übernehmen muss eine leistungstarke Aktualisierungsfunktion implementiert werden. Die Aktualisierung, die im folgenden betrachtet wird, ist nur auf pure::variants gerichtet. Es werden also ausschließlich externe Änderungen in pure::variants Merkmalsmodelle eingepflegt. Änderungen, die in pure::variants an den Merkmalsmodellen vorgenommen wurden, werden nicht in die externe Datenquelle übernommen!
Bei genauer Betrachtung lässt sich das Aktualisieren von Merkmalsmodellen in 2 Teilbereiche trennen, das Importieren der externen Daten und der Vergleich mit einem

14 Das Interface wird in der späteren Implementierung als 'IObjectMapping' bezeichnet

Originalmodell.

Bevor ein vorhandenes Modell aktualisiert werden kann müssen die Mapping-informationen, welche während des Importvorgangs gespeichert wurden, aufgelöst werden. Beispielsweise wurde ein Module Automobile aus Telelogic DOORS in pure::variants importiert. Um dieses Merkmalsmodell zu aktualisieren, muss ermittelt werden, aus welcher externen Datenquelle das Model stammt, also Telelogic DOORS. Weiterhin muss ermittelt werden, welches Module abgebildet wird. Ist bekannt, dass das Beispielmodell das Telelogic DOORS Module Automobile abbildet, wird dieses Module erneut importiert. Im Ergebnis liegt das original pure::variants Modell und das neu importierte Referenzmodell vor.

Der zweite Schritt ist der Vergleich von zwei Modellen, das Finden von Änderungen und die Darstellung der Änderungen in grafischen Editoren und letztendlich das Übernehmen von Änderungen. Dieser Bereich wird folgend nicht weiter betrachtet, da er kein Bestandteil des Framework darstellt und bereits implementiert ist. Das Framework hat an dieser Stelle die Funktion für (früher) importierte Modelle die externe Datenquelle zu finden und ein Referenzmodell erneut zu importieren. Beide Modelle sind Ausgangspunkt für die Aktualisierungsfunktion.

2.1.3 Export

Nach der Bildung und Überprüfung einer Variante muss die Information welche Merkmale verwendet werden, in die externe Datenquelle übertragen werden. Die Daten für den Export bestehen folglich aus einer Menge von Merkmalen und den zugehörigen Mappinginformationen.

Für die Transformation und Darstellung dieser Daten ist, wie beim Import, spezifisches Wissen über die externe Datenquelle notwendig. Das Framework kann an dieser Stelle nur unterstützende Funktion liefern. Das erfolgt durch die Bereitstellung aller Informationen die zum Exportieren von Varianten erforderlich sind:

- welche Varianten sollen exportiert werden
- aus welcher Datenquelle stammen die Merkmale
- welche Merkmale werden in einer Variante verwendet und
- welche externen Daten bilden die verwendeten Merkmale ab

Da keine Informationen vorliegen, wie die Variante in externe Datenquellen exportiert werden, wird im Framework ein Interface bereitgestellt. Neue Anbindungen externer Datenquellen können dieses Interface implementieren. Die unterstützenden Funktionen des Frameworks sind folgend im Pseudocode dargestellt:

```
Programm Export
    variantModel  :=    openVariantModel(location)
    inputModels[] :=    findUsedModels (variantModel)
    extVDM := createExternalVDMRepresentation()
    for each vdmElement in variantModel do{
        if (isSelekted (vdmElement) ){
            element := findElement(vdmElement,
inputModels[])
            extReference := getExternalReference(element)
            link extReference to extVDM
```

```
        }
    }
    function findElement ( vdmElement, inputModels[] ){
        refID := getReferenceID(vdmElement)
        for each model in inputModels[] do {
            element := findElement(model, refID)
            if( element != null){
                return element
            }
        }
    end function
    function getExternalReference ( element )
        mapp := getMappObject (element)
        return
    end function
end programm
```

Das Interface 'IObjectMapping' das für die Umsetzung des Mapping zwischen den pure::variants Informationsträgern und den externen Daten verantwortlich ist, wird in einer Standardimplementierung bereitgestellt. Für das Interface IModelSyncStore wird die Standardimplementierung XMLModelSyncStore implementiert. Diese Klasse speichert die Informationen der während des Import benutzen Instanzen an das Merkmalsmodell. Werden die Standardimplementierungen für die Integration einer externen Datenquelle benutzt sind nur die Interfaces „ICommitHandler“ und „ISourceHandler“ zu implementieren. Abbildung 9 zeigt den Zussamenhang der Interfaces im Framework.

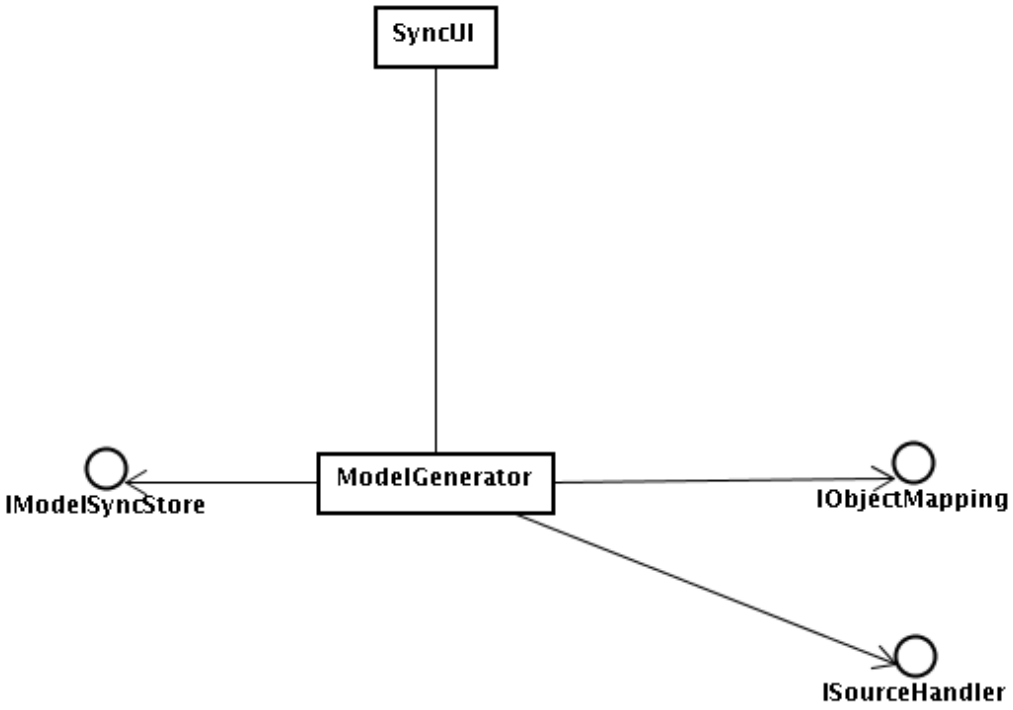


Abbildung 9: Interfaces für den Import

2.2 Implementierung

In diesem Kapitel wird die Implementierung des Framework vorgestellt. Zunächst werden für das allgemeine Verständnis die Rahmenbedingungen in der die Implementierung erfolgt, vorgestellt. Es wird beispielhaft auf die Umsetzung der Frameworkplanung eingegangen und Standardimplementierungen der durch das Framework bereitgestellten Interfaces erläutert.

2.2.1 Eclipse, pure::variants und Erweiterungspunkte

Eclipse[15] ist nach [Daun2003] „eine Plattform für alles Mögliche und nichts im Besonderen“, was etwas konkreter bedeutet, dass Eclipse eine Open Source Entwicklungsplattform ist, deren Funktionalität durch Erweiterungen nachträglich hinzugefügt wird. Eclipse besteht also aus einer Menge von Erweiterungen sg. Plugins, die die eigentliche Funktionalität enthalten. Jedes Plugin kann seine Funktionen durch Schnittstellen bzw. Erweiterungspunkte[16] an andere Plugins verfügbar machen, die diese ausnutzen und erweitern.

Pure::variants steht in seiner aktuellen Version als Eclipse Plugin zur Verfügung. Genauer betrachtet besteht pure::variants selbst aus einer Menge von auf einander aufbauenden Plugins. Eclipse benutzt die Programmiersprache Java. Dieses Paradigma erfüllt alle Anforderungen, die für die Umsetzung notwendig sind. Es werden insbesondere keine plattformabhängigen Komponten für die Umsetzung des Framework benötigt, die die Anbindung einer anderen Programmiersprache erfordern.

2.2.2 Framework als Plugin

Das Framework selbst erweitert die Funktionalität des pure::variants Plugin und wird selbst als Plugin implementiert. Das bietet die Möglichkeit ohne Änderungen des bestehenden Produktes dessen Funktionalität zu erweitern. Das Framework Plugin wird mit der ID com.ps.consul.eclipse.sync (folgend als Sync Plugin bezeichnet) in die bestehende Architektur eingebunden. Das Sync Plugin stellt die Interfaces 'SourceHandler', 'ICommitHandler', 'IObjectMapping' und 'IModelSyncStore' als Erweiterungspunkte zur Verfügung.

Die Integration einer externen Datenquelle erfolgt wiederum als Plugin unter Ausnutzung der Erweiterungspunkte des Sync Plugin.

2.2.3 Umsetzung der Interfaces

Das SyncUI stellt für den Benutzer des Sync Plugin die zentrale Schnittstelle dar. Über diese statische Klasse werden alle Erweiterungen des Sync Plugin ausgelesen, Instanzen der, über die bereitgestellten Erweiterungspunkten angemeldeten Erweiterungen erzeugt und zurückgegeben. Insbesondere die Klassen Modelgenerator und CommitHandler werden über das SyncUI bereitgestellt.

15 Open Source Entwicklungsumgebung von IBM, siehe www.eclipse.org
16 Erweiterungspunkte werden im Eclipseumfeld als Extension Points bezeichnet

Im ersten Schritt erfolgt die Definition von externen Interfaces für jeden pure::variants Informationsträger:

pure::variants Informationsträger	***externe Datenquelle***
IConsulModel	ImportModelDescriptor
IElement	IExternalElement
IProperty	IExternalProperty
IPropertyConstant	IExternalConstant
IDescription	IExternalDescription
...	...

Tabelle 2: Zuordnung zwischen pure::variants Interfaces und externen Interfaces

Kennzeichnend für die externen Interfaces ist, dass sie die gleichen Methoden zur Verfügung stellen, wie das zugeordnete pure::variants Interface. Alle externen Interfaces erweitern das Interfaces 'IExternalReference', welches die eindeutige Referenz bzw. Mappinginformationen der Daten zurückliefert.

Mit den definierten Interfaces ist es möglich die ModelGenerator Klasse mit dem in Kapitel 2.1.2 vorgestellten Pseudocode umzusetzen. In der Implementierung werden die, durch die externen Interfaces, bereitgestellten Daten auf die interne pure::variants XML Modellstruktur transformiert.

Im Framework wurden Standardimplementierungen der Interfaces 'IObjectMapping' und 'IModelSyncStore' umgesetzt. Die Implementierung des 'IObjectMapping' wurde 'ModelBasedObjectMapping' genannt und speichert die Mappinginformationen innerhalb der pure::variants Anforderungsmodelle. Dabei wird für jede externe Referenz ein neuer XML Knoten in das XML Modell eingefügt, der die Referenz enthält.

Die Implementierung des 'IModelSyncStore' speichert die während des Import benutzten Klassen in das XML Modell. Dafür werden am Rootelement neue Attribute angelegt, die die Klassen enthalten. Um Änderungen durch den Benutzer auszuschließen, sind die Attribute nicht sichtbar.

2.2.4 Probleme der Umsetzung

Während der Implementierungsphase und der Umsetzung der geplanten Klassenhierarchie gab es einige wesentliche Probleme hinsichtlich des Softwaredesign.
Das größte Problem, was sich durch die Anbindung von Telelogic DOORS zeigte, war die mangelnde Performance bei dem Aufbau eines pure::variants Modelles, die auf die Benutzung von pure::variants Server Kommandos zurückzuführen war.

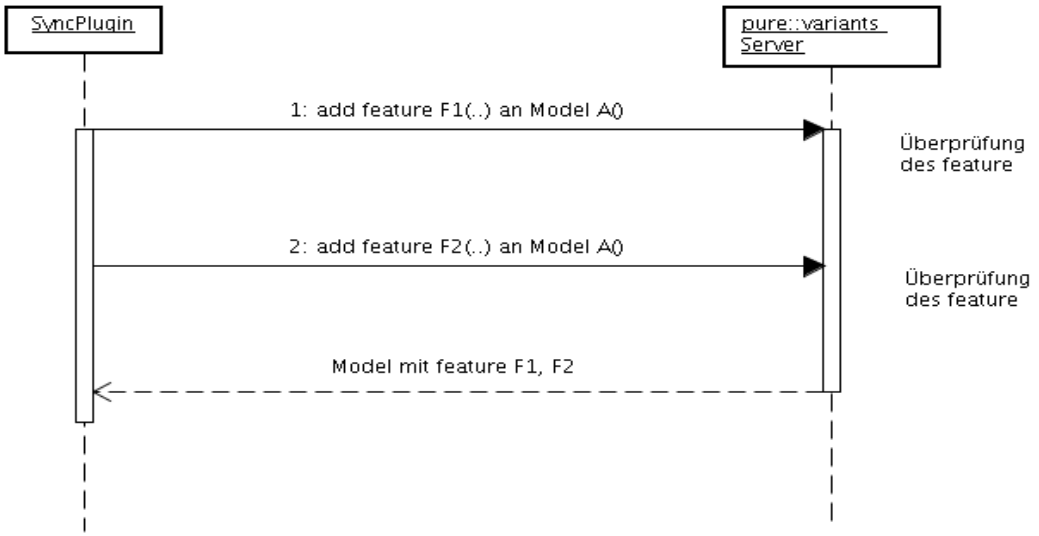


Abbildung 10: Aufbau eine pure::variants modell mit Server Kommandos

Die ursprüngliche Implementierung fügte jedes externe Merkmal an das Merkmalsmodell einzeln hinzu. vgl. Abb. 10. Das führte zu einer hohen Rechenbelastung des pure::variants Servers. Es existieren zwei Möglichkeiten um die Rechenlast zu senken. Das Sammeln aller Modeling Kommandos und die vollständige Liste, aus der das Modell erstellt wird, zusammen als ein Kommando an den Server zu schicken oder das pure::variants Modell vollständig durch das Framework zu generieren und dann im Server zu importieren. Es wurde die zweite Variante aus 2 Gründen implementiert. Der Aufbau des pure::variants Modell also des XML Dokumentes in Java kommt vollständig ohne die rechenintensiven Server Kommandos aus.

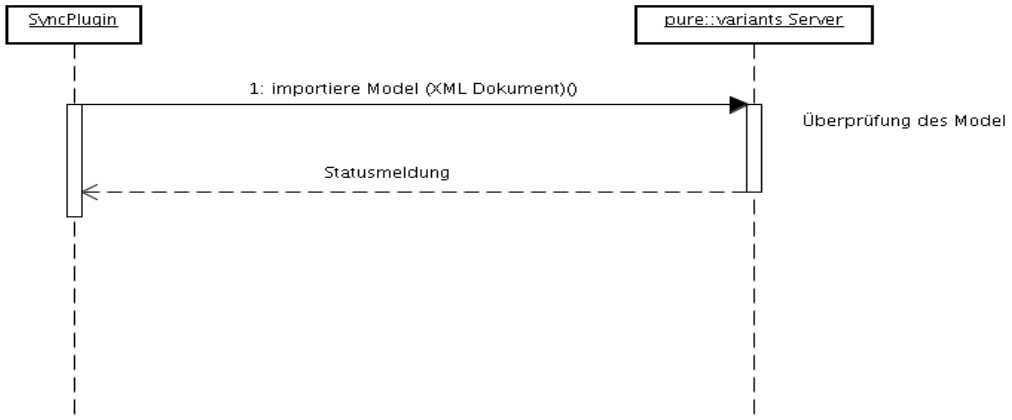


Abbildung 11: Aufbau eine pure::variants Modell ohne Server Kommandos

Der Import des XML Dokumentes zum Server und die damit verbundene Prüfung des Modelles, ist mit deutlich weniger Rechenaufwand verbunden. Zweitens benötigt die pure::variants Aktualisierungsfunktion ein XML Dokument, welches durch das Framework generiert wird.

Das zweite große Problem, was in der Spezifikation nicht erkannt wurde, bestand aus folgendem Sachverhalt. Der in pure::variants implementierte Aktualisierungsalgorithmus findet zusammengehörige Elemente über die gleiche ID. Das heisst Merkmale müssen in beiden Modellen die gleiche ID besitzen. Ist das nicht der Fall, ist der Aktualisierungs-algorithmus nicht in der Lage diese zu vergleichen. Da während des Designs die Annahme getroffen wurde, dass Merkmale über ihr Mapping gefunden werden können, lag hier ein elementares Problem vor. Es musste sicher gestellt werden, dass Merkmale die ein externes Datum abbilden, bei jedem Importvorgang die gleiche ID zugewiesen bekommen damit diese durch den Aktualisierungsalgorithmus als „Paar“ das verglichen werden kann, erkannt wird. Die Lösung für das Problem konnte aber in die vorhandene Designstrukur eingebettet werden, ohne die Architektur grundsätzlich in Frage zustellen. Das wurde erreicht, indem das Interface das die externen Daten abbildet um eine ID erweitert wurde die für den Import benutzt werden kann!

Es ist mit dieser Anpassung möglich die ID der pure::variants Informationsträger aus externen Daten abzubilden und als Mapping zu benutzen.

3 Anbindungen über das Framework

Nachdem im ersten Kapitel die Werkzeuge DOORS und Caliber RM vorgestellt wurden, wird in diesem Kapitel die Integration erläutert. Es werden die Ideen der Anbindungen vorgestellt und mit Abbildungen erläutert, wie die Umsetzung erfolgt ist. Da nicht alles ohne Probleme umzusetzen war, sollen die kritischen Punkte der Implementierungen nicht unerwähnt bleiben und werden ebenfalls kurz vorgestellt.

3.1 Integration Telelogic DOORS

Im einführenden Teil wurde festgestellt, dass es zwei Möglichkeiten gibt, die Daten von Telelogic DOORS zu pure::variants zu übertragen, wobei nur der Zugriff über eine TCP/IP Verbindung als realistische Lösung in Frage kommt. Folgende Probleme waren zu lösen:

- Kommunikation zwischen pure::variants und DOORS
- Datenextraktion aus DOORS
- Abbildung der Daten in pure::variants
- Export von Varianten zu DOORS

Für die Kommunikation zwischen DOORS und pure::variants die via TCP/IP umgesetzt wird, mussten 4(!) Sockets implementiert werden, über die Informationen ausgetauscht werden können. Auf der pure::variants Seite werden 2 Sockets geöffnet, ein Serversocket und ein Client. In DOORS wird ein DXL Skript ausgeführt, der ebenfalls einen Serversocket öffnet. Während der Skriptabarbeitung wird ein weiterer Socket geöffnet.

Das Ablaufdiagramm der Kommunikation ist in Abbildung 12 dargestellt. Dabei werden im ersten Schritt DXL Skripte, die von DOORS verarbeitet werden können, von der Festplatte geladen. Diese erhalten dann die notwendigen Funktionsaufrufe mit den steuernden Parametern. Beispielsweise enthält der DXL Skript 'ModuleImportScript.dxl'

die Parameter für das Module, welches ausgelesen werden soll, welche Attribute nicht importiert werden sollen und wie lang die maximale Attributwertlänge ist. Im nächsten Schritt wird eine TCP/IP Verbindung zu DOORS aufgebaut und das Skript übertragen. Das DXL Skript Server ruft die DXL Methode 'eval_' mit dem Skript auf. Das Skript wird von DOORS compiliert und ausgeführt. Um Ergebnisse eines Skript wieder an pure::variants zu senden, wird durch einen Aufruf im Skript selber ein TCP/IP Socket aufgebaut, der die Ergebnisse an den Serversocket von pure::variants sendet.

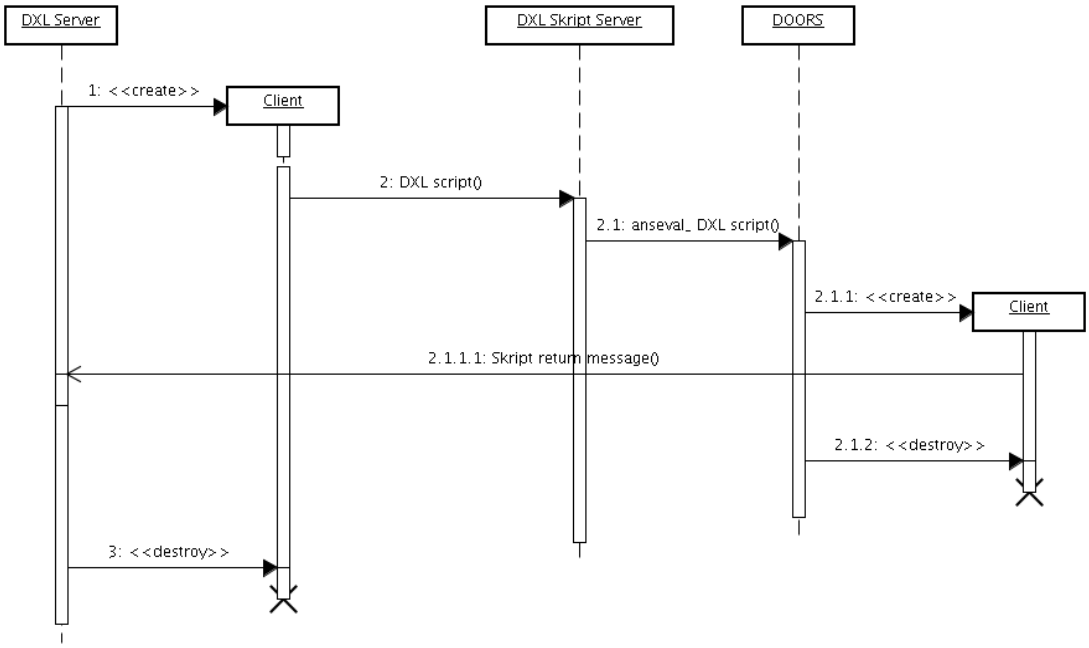


Abbildung 12: Kommunikation zwischen pure::variants und Telelogic DOORS

Die Ergebnisse der Skripte werden in einer XML Struktur[17] erzeugt. Über das vorgegebene Format können neben dem eigentlichen Ergebnis des Skripts, Fehlermeldungen, Warnungen und Hinweise zurückgegeben werden.

Die vorgestellte Umsetzung hat einen entscheidenden Schwachpunkt. Wird ein Skript aufgrund von Fehlern durch den DXL Skript Server nicht compiliert, kann er nicht ausgeführt werden. Das bedeutet, dass kein Clientsocket geöffnet wird über den die Ergebnisse zurückgeschickt werden. Der DXLServer in pure::variants kann nicht zwischen den Ereignissen a)„Skript wird noch abgearbeitet und sendet Ergebnis nach Beendigung “ und b) „Skript wird wegen Fehlern nicht ausgeführt und wird nie ein Ergebnis senden“ unterscheiden. Hat ein Skript nach 2 Minuten kein Ergebnis an den DXLServer gesendet wird davon ausgegangen das Ereignis b) vorliegt.

Mögliche Lösungen für diese Probleme sind:

- der DXL Skript Server sendet eine Fehlermeldung, wenn das Skript nicht erfolgreich compiliert werden kann und
- das DXL Skript sendet in festen Zeitintervallen einen Ping an den DXLServer und signalisiert eine erfolgreiche Abbarbeitung.

17 Der genaue Aufbau der Ergebnisstruktur ist im Anhang vorhanden

Auf Daten in DOORS wird mit Hilfe der eingebauten Programmiersprache DXL zugegriffen. DXL ist an C und C++ angelehnt. Für die Datenextraktion wurde ein Skript in DXL programmiert, das alle Informationen eines Module in ein XML Format abbildet. Dieses Skript lädt das angeforderte DOORS Module, iteriert über alle Objekte, Attribute, Attributwerte und Links des Module und fügt die Daten in ein XML Dokument[18] ein.

Das XML Dokument wird in pure::variants durch die Implementierung des Interfaces 'ISourceHandler' ausgewertet. Die externen Daten eines Module werden auf die entsprechenden Interfaces abgebildet. Es erfolgt folgende Zuordnung:

- DOORS Module ↔ pure::variants Modell
- DOORS Objekt ↔ pure::variants Element
- DOORS Attribute ↔ pure::variants Attribute
- DOORS Attributewert ↔ pure::variants Konstante
- DOORS Link ↔ pure::variants Relation

Eine Sonderstellung nimmt das DOORS Attribut vom Typ „Enumeration“ ein, da es in pure::variants keinen entsprechenden Attributetyp gibt. Die Werte der Enumeration werden als Liste in einem Attribute von Typ String abgebildet.

Für den Export einer Variante nach DOORS muss vom Benutzer ein Module angegeben werden. In dem Module wird ein Objekt angelegt, welches den Namen der Variante enthält. Das Objekt wird mit allen Objekten, die in der Variante enhalten sind, verknüft. Das Variantenobjekt zeigt mit Hilfe der Links auf die Merkmale, die in ihm enthalten sind. Optional kann während des Exports aus dem Merkmalsdokument die Variante generiert werden. Das Variantendokument enthält genau dann die Merkmale, die in der Variante benutzt werden, alle anderen Merkmale sind in diesem Dokument nicht enthalten.

3.2 Borland Caliber RM

Die Anbindung von Caliber RM gestaltet sich durch die vorhandene Java API deutlich einfacher als die Integration von Telelogic DOORS. Die API von Caliber RM übernimmt vollständig die Funktion der Kommunikation. Auf eine eigene Implementierung der Verbindung kann daher vollständig verzichtet werden.

Die Daten aus Caliber RM müssen für den Import auf das Datenmodell von pure::variants abgebildet werden. Diese Funktion übernimmt die Implementierung des vom Framework bereitgestellten Interface ISourceHandler. Die Daten aus Caliber RM werden wie folgt auf pure::variants Informationsträger abgebildet:

- Caliber RM Project ↔ pure::variants Modell
- Caliber RM RequirementType ↔ pure::variants Element
- Caliber RM Requirement ↔ pure::variants Element
- Caliber RM (System)Attribute ↔ pure::variants Attribute
- Caliber RM User Defined Attribute ↔ pure::variants Attribute
- Caliber RM Attributewert ↔ pure::variants AttributeKonstante
- Caliber RM Trace ↔ pure::variants Relation

Für den Export muss der Benutzer ein Caliber RM Projekt angeben, in das die Variante

18 Die genaue Struktur des XML Dokumentes befindet sich im Anhang

exportiert werden soll. In diesem Projekt wird ein RequirementType mit dem Namen 'Variants' angelegt, wenn dieser Typ noch nicht existiert. Unter dem Variants-RequirementType wird ein Requirement mit dem Namen der Variante angelegt. Das Varianten Requirement wird mit allen Requirements, die in der Variante verwendet werden, anhand von Traces verbunden. Analog zu DOORS enthält das Variantenobjekt Traces zu allen Merkmalen, die in der Variante benutzt werden. Bereits vorhandene Variantenobjekte, beispielsweise aus früheren Exporten, werden entsprechend aktualisiert.

3.3 Lines of Code

Tabelle 3 stellt die Anzahl der Quellcodezeilen[19] (LOC) der Integration von Telelogic DOORS und Borland Caliber RM im Vergleich gegenüber.

Komponente	***mit Sync Plugin***		***ohne Sync Plugin***	
	LOC	MLOC	LOC	MLOC
com.ps.consul.eclipse.ui.door	5766	3471	5695	3504
com.ps.consul.eclipse.ui.caliber	3544	2025	5684	3437

Tabelle 3: LOC der Integration über das Framework

Die Berechnung der LOC ist die Summe der Zeilen der betrachteten Plugins ohne Leerzeilen und Kommentarzeilen. Die MLOC Metrik ist die Differenz der LOC und der Methodensignaturen und beinhaltet entsprechend die LOC innerhalb von Methoden ohne Leerzeilen und Kommentare.

Es ist zu beachten, dass die gegenübergestellten Integrationen von DOORS und Caliber RM mit bzw. ohne Anbindung über das Framework nur bedingt zu vergleichen sind, da die Funktionalität der Version, die das Framework benutzt, erweiterte wurde. In der ursprünglichen Version des DOORS Plugin wurden die pure::variants Featuremodelle in DOORS über DXL aufgebaut und als Ergebnis zurückgeliefert, was zu einer Verzerrung der LOC führt, da DXL Skripte nicht in der Berechnung enthalten sind. Unter Berücksichtigung der LOC der DXL Skripte ergibt sich folgende, berichtigte LOC:

Komponente		***LOC***	***MLOC***
com.ps.consul.eclipse.ui.door –	über Sync Plugin	6841	4617
com.ps.consul.eclipse.ui.door –	ohne Sync Plugin	6915	4735

Tabelle 4: LOC der Integrationen mit Berücksichtigung der DXL Skripte

19 Englisch Lines of Code - LOC

Bei der Berechnung der LOC der DXL Skripte wurde nur die Funktionalität berücksichtigt, die in beiden Versionen enthalten ist. Als prozentualle Reduzierung der Integrationen über das Framework, wobei die Integrationen ohne Framework als Basis dienen, ergeben sich folgende Werte:

Komponente	***Mit Framework***	
	LOC	***MLOC***
com.ps.consul.eclipse.ui.door[20]	98,9%	97.5%
com.ps.consul.eclipse.ui.caliber	62.3%	58,91%

Tabelle 5: prozentualler Vergleich der LOC der Integrationen

Das Framework selbst enthält 4712 LOC und 2431 MLOC.

4 Resümee

Anhand der Gegenüberstellung der Quellcodezeilen der Integrationen von Telelogic DOORS und Borland Caliber RM mit und ohne Benutzung des Framework - (vergleiche Kapitel 4.1, 4.2)- wurde dessen Nutzen zur Aufwandsminimierung neuer Anbindungen externer Datenquellen an pure::variants bewiesen. Natürlich kann nicht behauptet werden das die fertiggestellte Implementierung frei von Fehlern ist, das währe vermessen. Aber es zeigt sich durch Caliber RM und DOORS bzw. durch deren Integrationen, dass sich kaum neue Fehler ergeben. Woran liegt das? Viele Klassen sind übersichtlich und kommen mit weniger als 300 Zeilen Quellcode aus, was selbstredend zu übersichtlichen Programmen führt. Nicht zu letzt sind alle die Integrationen über das Framework ähnlich vom Ablauf gegliedert, bestimmte Funktionen sind immer in den jeweiligen Interfaces zu finden. Das führt, vorrausgesetzt die Zusammenhänge sind erstmal verstanden worden, zu einem schnellen Verstehen von Anbindungen. Ein anderer Programmierer, der übrigens mit der Integration des Filesystems beschäftigt war, bestätigte, dass nach der Einführung in die Schnittstelle eine schnelle Umsetzung möglich war. Leider die intuitive, also der selbsterklärende Aufbau nicht so gelungen. Eine Erklärung, über die Dokumentation hinausgehend, ist für Entwickler hilfreich. Notwendig ist eine Einweisung freilich nicht, aber einige wenige Minuten für eine kurze Einweisung können dem Programmierer einige Stunden an Arbeit für die Integration sparen helfen.

20 Berechnung erfolgt unter Berücksichtigung der LOC der DXL Skripte

Anhang

Spezifikation der Ergebnisstruktur von DXL Skripten

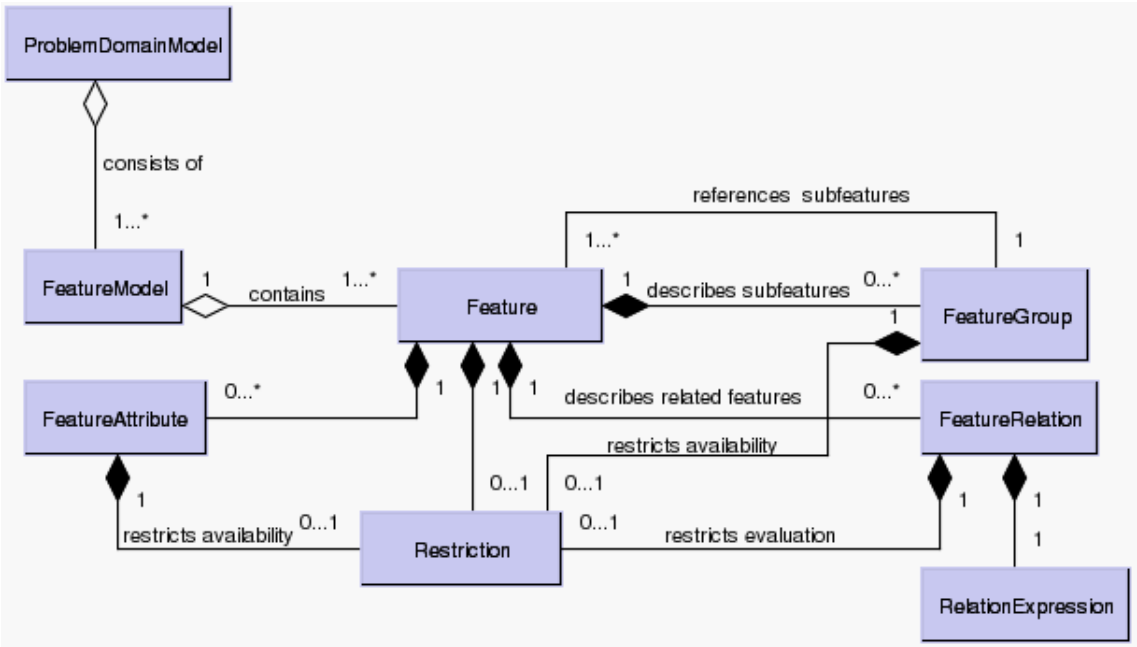
```
<response result="success|failed">
  <result>
     ... Ergebnis eines Skript ...
  </result>
  <errors>
     <error severity = "error|warning|info" code = "errorcode"> error
msg  </error>
  </errors>
</response>
```

Spezifikation des XML Dokument beim Moduleimport

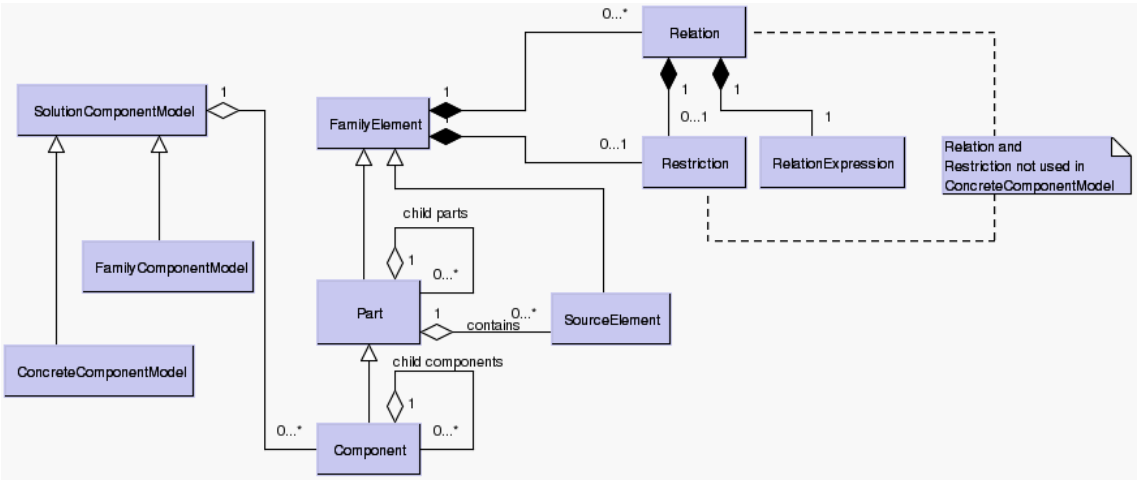
```
<module name = „Modulename“ id = „ModuleID“>
     <object absNo = „ObjectID“>
          <object absNo = „childrenID“>
          <property
               attrName=“Attributename“ attrType=“type“
               defaultValue=“default“ inherit =“true|false“>
               <value>
                    „Attributewert“
               </value>
          </property>
               ...
          </object>
     </object>
</module>
```

Spezifikation der pure::variants XML Modelle

Featuremodell



Komponentenmodell



Verwendete Requirements Management Werkzeuge

Hersteller	***Produktname***	***Aktuelle Version***	***Anschrift***	***Web***
Borland	Caliber RM	V. 2005 (7.1)	Robert-Bosch-Strasse 11, 63225 Langen	www.borland.de
Telelogic Deutschland GmbH	DOORS	V. 8.0	Otto-Brenner-Strasse 247, Bielefeld	www.telelogic.de
Pure-systems GmbH	pure::variants	V. 2.0.2	Agnetenstrasse 14, 39106 Magdeburg	www.pure-systems.com

Featuremodell Icons

Bezeichnung	***Icon***
Notwendig	
Optional	
Alternative	
Oder	

Literaturverzeichnis

[Daun2003] Berthold Daun, „Javaentwicklung mit Eclipse 2“, 1. Auflage 2003 dpunkt.verlag GmbH

[Leff2000] Dean Leffingwell, Don Widrig: „Managing Software Requirements – A unified approach“, 1. Auflage 2000, Addison-Wesley

[Nied2000] Frank Niederländer, „Dynamik in der internationalen Produktpolitik von Automobilherstellern“ , 1. Auflage 2000, Betriebswirtschaftlicher Verlag Gabler

[Du2003] Rainer Dumke, „Software Engineering“, 4. Auflage 2003, Vieweg Verlag

[Deur2002] A. van Deursen, P. Klint, „Domain-Specific Language Design Requires Feature Descriptions“, 2002, http://homepages.cwi.nl/~arie/papers/fdl/fdl.pdf

[Eng2004] Claus Engel, „Erfolgreich Projekte durchführen“
www.gpm-ipma.de/docs/fdownload.php?
download=GPM_2004_Ergebnisse_final.pdf

Tabellenverzeichnis