Modellgetriebene homogenisierte Datenhaltung und Synchronisation mit vorhersagbaren Eigenschaften
für plattformübergreifende Anwendungen

Förderkennzeichen: 01 S12012

AP 9

Testaufbau und Durchführung, Ergebnisdokumentation, Testfallbeschreibung und QS-Kriterien für Modelle, Plattform und Generat

Autor: A. Schwichtenberg (CAS)
Co-Autoren: E. Gailus (B2M), G. Hübsch (CAS), H. Groenda (FZI)
Fertiggestellt am: 30. 06. 2014
Schlagworte: Qualitätsmanagement

Projektpartner

GEFÖRDERT VOM

Änderungshistorie

Version	Änderungen von	Datum	Anmerkung
Draft	A.S./G.H	06.06.2014	Gliederung und Zuständigkeiten
0.1	E.G.	12.06.2014	Kapitel 2.1.1 hinzugefügt
0.2	A.S.	25.06.2014	CAS spezifische Teile
0.3	H.G.	25.06.2014	Beiträge und Kapitel FZI, Review
1.0	A.S.	01.07.2014	Review, Überarbeitung

ToDos

Abschnitt	von	Beschreibung	Pritorität	Todo

Inhalt

Abbildungen

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

1 Einleitung

Das vorliegende Dokument beschreibt die im Arbeitspakt 9 angewandten Maßnahmen zur Qualitätssicherung der im MOHITO-Projekt erstellten Software- und Modellartefakte. Ziel ist die Einhaltung von Qualitätskriterien, um eine leichte Übernahme dieser Projektergebnisse in Produktivsysteme der Industriepartner zu erlauben (wie in den Verwertungsplänen vorgesehen). Ausgangspunkt für die hier dokumentierten Arbeiten sind die in (Schwichtenberg, Gailus, Klatt, & Hübsch) definierten Qualitätskriterien und Qualitätssicherungsmaßnahmen.
Entsprechend dem jeweiligen Fortschritt des Projekts in den Bereichen des DSL-Entwurfs sowie der Generatorableitung und der Framework-Entwicklung wurden kontinuierlich jene QS-Detailmaßnahmen und Tests angewandt, die sich während des Projekts als notwendig erwiesen haben.
Prinzipiell wurden für alle Artefakte, die innerhalb des MOHITO Projektes erstellt wurden, Maßnahmen definiert, welche die Qualität der entsprechenden Artefakte sicherstellen. Klarerweise unterscheiden sich die Mechanismen zur Qualitätssicherung abhängig von der Art des zu entwickelnden Artefaktes. So werden für die Qualitätssicherung des generierten Codes automatische Tests (z.B. JUnit Tests) bereitgestellt, die die Funktionalität der Applikation testen; andere Teile hingegen, wie z.B. das Datenmodell, lassen sich nicht auf diese Weise überprüfen und erfordern andere Mechanismen zur Qualitätssicherung (bzw. zur Erhöhung der Qualität).
In dem vorliegenden Dokument werden die spezifischen Mechanismen zur Qualitätssicherung beschrieben und für jedes Artefakt geeigneten Methoden definiert, wie dies sichergestellt wurde. Da die im Rahmen des MOHITO Projektes entstehenden Ergebnisse auch in die Produkte der Partner B2M und CAS Software AG übernommen werden soll – sofern das möglich ist – werden anschließend für diesen Prozess Qualitätssicherungsmechanismen definiert, um eine möglichst einfach Übernahme der Projektergebnisse zu gewährleisten. Eine umfassende Migrationsstrategie wurde im Bericht L5.1 Migrationsstrategie für Bestandsanwendungen zur MOHITO Referenzarchitektur (Klatt, Gailus, Hübsch, & Schwichtenberg, 2013) und L5.2 Migrationsstrategie für die sanfte Überführung von Bestandssoftware zu modellbasierten Multi-Plattformanwendungen mittels MOHITO (Schwichtenberg, Hübsch, Gailus, & Groenda, 2014) innerhalb des Arbeitspaketes fünf des MOHITO Projektes beschrieben.

Im Rahmen des Projektes wurden für die folgenden Artefakte Qualitätssicherungsmechanismen umgesetzt und im Folgenden beschrieben:

- Code-bezogene Artefakte (s. Kap. 2.1):
  - Bibliotheken (libs)
  - Editoren auf Basis des Eclipse Modelling Frameworks
  - Reverse-Engineering Werkzeuge (CAS Code-to-Model Generierung)
  - Code-Generatoren
  - Ergebnis der Code-Generierung
- Modell-Entwicklungsartefakte (s. Kap. 2.2):
  - Meta-Modelle
  - Domänen-spezifische Sprachen (DSL)
  - Modell-Instanzen
- Begleitdokumentation, Beispielprojekt (s. Kap. 2.3)
  - Vereinbarte Lieferungen an den Projektträger
  - interne Arbeitsdokumente zur Software-Entwicklung
  - Anwendungsdokumentation

2 Angewandte Qualitätssicherungsmechanismen

Generell muss unterschieden werden zwischen automatischen und manuellen Mechanismen zur Qualitätssicherung. Automatische Mechanismen werden vor allem zur Überprüfung der syntaktischen oder funktionalen Korrektheit eingesetzt. Um die semantische Korrektheit des Modells oder die Usability der Applikation zu testen, muss in der Regel auf manuelle Überprüfungen zurückgegriffen werden, da sich dies nur in Ausnahmen automatisch prüfen lässt. Gerade hier ist es wichtig, Anforderungen an die Entwicklungsumgebung und die Modellsprache zu definieren, um möglichst viele Fehlerquellen auszuschließen.
Alle im Rahmen von MOHITO erzeugten Artefakte durchlaufen vor ihrer Veröffentlichung einen Qualitätssicherungsprozess, der sich an der Art des jeweiligen Artefakts orientiert. Generell wird im Projekt zwischen Code-Entwicklungsartefakten, Modell-Entwicklungsartefakten und Dokumenten unterschieden (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).
Für alle Artefakte gilt, dass sie vor ihrer Veröffentlichung wechselseitig durch die Projektpartner qualitätsgesichert werden. Dabei wird am Ende für jedes Artefakt die Qualität durch Abgleich mit den Anforderungen aus L 2.1: „Dokumentation der Anwendungsszenarien, der Anforderungen und des Stands der Technik“ (Gailus, Klatt, Hübsch, Krogmann, & Schwichtenberg) geprüft; das Ergebnis dessen ist in L2.1 Dokumentation der Validierungsergebnisse (Gailus, Hübsch, Schwichtenberg, & Groenda, 2014) detailliert beschrieben.
Eine Freigabe von Artefakten erfolgt nach positiver Evaluation. In allen anderen Fällen erfolgt eine Rücksprache und Überarbeitung durch den entwickelnden Projektpartner.
Im Folgenden wird für alle Artefakte, die im MOHITO Projekt entwickelt werden, definiert, welche Qualitätssicherungsmechanismen sich eignen.

2.1 Code-Entwicklungsartefakte

Für die Qualitätssicherung von Code-Artefakten wird auf die etablierten Standards aus der Software-Entwicklung zurückgegriffen. Um bereits im frühen Stadium der Entwicklung ein gutes Design der Software sicherzustellen und Fehlentscheidungen verhindern wurden frühzeitig Reviews von Architektur und Entwurf durchgeführt und die Erkenntnisse daraus zeitnah umgesetzt. Solche Reviews wurden regelmäßig manuell und in Teamarbeit, auch Projektpartner-übergreifend abgehalten. Wesentlicher Teil der manuellen Code Reviews war die Überprüfung der im Projekt definieren Coding-Guidelines, die einen einheitlichen Code Stil und damit die Lesbarkeit des Codes sicherstellen.

Unit Tests wurden genutzt um auf feingranularer Ebene, wie zum Beispiel Klassen und Methoden den Code automatisiert und reproduzierbar zu prüfen. Im Folgenden wird für die einzelnen Bereiche detailliert beschrieben, welche Qualitätssicherungsmaßnahmen durchge-

führt wurden. Zusätzlich werden die in L9.1 QM-Maßnahmen und Tests (Schwichtenberg, Gailus, Klatt, & Hübsch) definierten Qualitätskriterien auf Ihre Erfüllung hin geprüft.

2.1.1 Bibliotheken des MOHITO Frameworks

Die Bibliotheken des MOHITO Projektes mussten in zweierlei Hinsicht auf ihre Qualität geprüft werden. Zum einen musste die Qualität des eigentlichen Quellcodes sichergestellt werden. Zum anderen musste die generelle Architektur des MOHITO Frameworks und dessen Plattformunabhängigkeit überprüft werden. Im Folgenden soll zuerst auf die Sicherung der eigentlichen Codequalität und danach auf die Qualitätsicherungsmechanismen zur Softwarearchitektur eingegangen werden.

Sicherung der Codequalität

Um eine gute Qualität der Quellcodes zu gewährleisten wurde besonders auf die Lesbarkeit, die Testbarkeit und die Änderbarkeit des Quellcodes geachtet. Zur Lesbarkeit trugen die Verwendung eines einheitlichen Programmierstils, die strikte Beachtung von „Coding Conventions“ und eine durchgehende Dokumentation im Quellcode bei. Die Testbarkeit des Codes zu gewährleisten war wichtig, da ein Teil, des MOHITO Frameworks durch Unit Tests abgedeckt ist. Die erzielte Testabdeckung erleichterte auch Code-Refactorings, die nach regelmäßigen Code-Reviews aller beteiligten Projektpartner durchgeführt wurden, um aufgefundene Defekte zu beseitigen. Darüber hinaus wurden auch Werkzeuge zum Ermitteln von Code-Metriken eingesetzt, die Stellen aufzeigten, wo Quellcode nicht optimal geschrieben war, um dort Verbesserungen und Refactorings vor zu nehmen. All diese Mechanismen und Werkzeuge trugen zu einer guten Codequalität und Änderbarkeit des Quellcodes bei, welche im Hinblick auf eine Nutzung und Weiterentwicklung in Form eines „Open-Source“ Projektes unabdingbar sind.

Qualitätssicherung der Softwarearchitektur

Um die Qualität der Softwarearchitektur sicher zu stellen, wurden zuerst im Rahmen der Anforderungsanalyse alle Anforderungen an das MOHITO Framework definiert, die während des kompletten Entwicklungszyklus immer wieder validiert und zu Ende des Projektes auch letztendlich erreicht wurden. In einem ersten Schritt nach der Anforderungsanalyse wurde ein initiales Konzept auf dem Stand der Technik erarbeitet. Danach wurde unter Verwendung der Referenzimplementierungen eine erste Version des MOHITO Frameworks implementiert. Diese wurde dann über die gesamte Projektlaufzeit immer wieder überarbeitet und an neue Erkenntnisse bzw. Gegebenheiten der Modellierung und Referenzimplementierungen angepasst. Ein wichtiger Aspekt der Qualitätssicherung der Softwarearchitektur stellt aber auch hier der regelmäßige Review Prozess aller beteiligter Projektpartner dar, der letzten Endes zu einem qualitativ hochwertigen MOHITO Framework führte.

2.1.2 Editoren auf Basis des Eclipse Modelling Frameworks

Die Benutzerschnittstelle soll einfach bedienbar sein. Die Editoren von MOHITO basieren aus diesem Grund auf den gleichen Konzepten wie bereits vorhanden Editoren der Eclipse IDE. Die Qualitätssicherung erfolgte zusätzlich durch manuelle Tests der Projektpartner, die eine adäquate Umsetzung sicherstellten und schnelle Rückmeldungen ermöglichten.

2.1.3 Reverseengineering Werkzeuge

Um das sehr umfangreiche Datenmodell des CAS Anwendungsfalls nicht manuell modellieren zu müssen, sondern die bereits bestehenden Beschreibungen des Datenmodells nutzen zu können, wurde von CAS eine Modelltransformation entwickelt, die auf der Basis einer existierenden XML Datentyp Beschreibung ein MOHITO-fähiges ecore Modell erzeugt. Dies wurde auf der Basis von QVTo (Object Management Group, 2011) umgesetzt. Für die Konvertierung wurden automatische Tests geschrieben, die sicherstellen:

1. Dass der Konvertierungsdurchlauf erfolgreich war
2. Das Konvertierungsergebnis syntaktisch valide ist (also der ecore Spezifikation entspricht)

Die semantische Korrektheit wurde manuell sichergestellt, sowie durch eine Zurücktransformation des ecore Modells in das XML Datenmodell untermauert.

2.1.4 Code-Generatoren

Die Performance der Generatoren war weniger kritisch. Eine einmalige Erzeugung eines umfangreichen Modells darf durchaus Minuten Anspruch nehmen, da dies nur Einmalig bei der Entwicklung nach Änderungen am Datenmodell notwendig ist. Auf Basis von Testmodelle wurde die Laufzeit gemessen und blieb mit mehreren Sekunden stets deutlich unter der gesetzten Akzeptanzschwelle.

Die Lesbarkeit des Quellcodes der Generatoren wurde durch größtenteils durch Reviews abgedeckt. Die Durchsetzung semantisch sinnvoller Bezeichnungen und eine übersichtliche Formatierung sowie die Definition von Schnittstellen konnte durch Personen besser als durch automatisierte Werkzeuge abgedeckt werden. Zusätzlich wurde auf diese Weise Wissen über den Aufbau im Team verteilt,

Der Quellcode ist durchgehen dokumentiert und in Verbindung mit den architekturellen Beschreibungen und Entwurfsentscheidungen eine gute Basis für weitere Entwicklungen. Die

Qualität der Dokumentation wurde durch Reviews sicher gestellt. Dies erlaubt es, die Notwendigkeit und Verständlichkeit der Kommentare höher zu gewichten als deren reine Anzahl.

2.1.5 Ergebnis der Code-Generierung

Ergebnis der Code-Generierung ist hier sowohl der Code, der durch den Code Generator erzeugt wurde, als auch Deskriptor- und Konfigurationsdateien, die durch eine Modell-zu-Text-Transformation erstellt wurden.

xRM Demonstrator

Bei allen Generatortemplates wurde darauf geachtet, dass der zu erzeugenden Code gemäß den Standardrichtlinien formatiert und definiert ist – so wurde beispielsweise darauf geachtet, dass Variablenbezeichnungen immer sinnvoll und beschreibend sind. Dies garantiert eine gute Lesbarkeit des erzeugten Codes und erleichtert die Fehlersuche, die auch dadurch weiter unterstützt wird, dass aus dem erzeugten Code immer hervorgeht, welches Generatortemplate diesen erzeugt hat. So weiß der Fehlersuchende sofort, an welcher Stelle zu suchen ist. Die praktische Anwendung hat gezeigt, dass das Auffinden von fehlerhaften Generationsanweisungen kein Problem darstellt.

Für alle Basisoperationen des generierten Codes wurden automatische Tests in JUnit geschrieben, die auch während der Weiterentwicklung sicherstellen, dass die erzeugten Codeteile fehlerfrei funktionieren. Für die xRM Anwendung wurden u.a. JUnit Tests mit verschiedenen Testdaten geschrieben für:

1. Das Erzeugen von neuen Datensätzen
2. Das Ändern bestehender Daten
3. Das Löschen von Daten

Mobile Demonstrator

Der Mobile Demonstrator der B2M ist an das EMF Tutorial „Generating an EMF Model“ angelehnt. Dieses Tutorial beschreibt eine einfache Bücherei Anwendung, in welcher Bücher verschiedener Autoren in unterschiedlichen Kategorien vorhanden sind, die dann entsprechend ausgeliehen, also aus dem System entfernt, und später wieder zurückgebracht, also im System wieder verfügbar gemacht werden können. Der Demonstrator wurde im Hinblick darauf entwickelt als Tutorial im Rahmen einer Open-Source Veröffentlichung des MOHITO Projektes dienen zu können. Daher ist der Code und die entsprechende Programmdomäne einfach und verständlich gehalten, demonstriert aber gleichzeitig alle Funktionalitäten des MOHITO Frameworks. Im Zusammenhang mit einer Open-Source Veröffentlichung ist auch eine gute Codequalität notwendig. Gerade eine gute Lesbarkeit und Klarheit des Codes ist hier von Nöten und sollte sich Zusätzlich an typischen Konventionen für Android Anwendun-

gen orientieren.. Diese wurde ähnlich zu der Entwicklung des MOHITO Frameworks durch einen einheitlichen Programmierstil sichergestellt.

2.2 Modell-Entwicklungsartefakte

Neben den klassischen Code Artefakten entstanden im MOHITO Projekt auch diverse Modell-Entwicklungsartefakte, für deren Qualitätssicherung in (Schwichtenberg, Gailus, Klatt, & Hübsch) entsprechende Mechanismen definiert wurden. In diesem Gebiet gibt es wenige etablierte Möglichkeiten zur automatischen Prüfung. Daher wurde die Qualität von Modell-Entwicklungsartefakten primär durch manuelle Reviews sichergestellt. Bei diesen Reviews wurden sowohl übergreifende Modellkonzepte, als auch detailliertere Aspekte, wie Referenzen zwischen Elementen und Elementattributen bewertet.

2.2.1 Meta-Modelle

Die Verständlichkeit der MOHITO Datenmodelle wurde durch die Nutzug etablierter Standards gewährleistet. Die grafische Darstellung und einfache Bearbeitung wird mittels Ecore Diagrammen unterstützt. Die notwendige Einarbeitung für die Zielgruppe der Anwender ist durch die Nutzung dieser an UML Klassendiagrammen orientierten Darstellung möglichst gering.

Die Ausdrucksmächtigkeit und Vollständigkeit wurde durch die Abdeckung Modellierungskonstrukte von Klassen (inklusive Aufzählungen) und Beziehungen sowie Berücksichtigung der Anforderungen aus der industriellen Praxis an Hand der Anwendungsfälle sichergestellt. Die Prüfung erfolgte manuell, da es sich um eine Konzeptprüfung handelt.

Ein Beispielmodell ist in Form des Tutorials öffentlich verfügbar. Ein Modell mittlerer Komplexität wurde im Rahmen des Here-I-Am Demonstrators entwickelt, ein komplexes Modell für den xRM Demonstrator. Diese stehen den Projektpartnern zur Verfügung.

2.2.2 Domänen-spezifische Sprachen

Die Mächtigkeit der DSL wurde durch die Abdeckung der Modellierungskonstrukte von Klassen (inklusive Aufzählungen) und Beziehungen sowie der Berücksichtigung der Anforderungen aus der industriellen Praxis an Hand der Anwendungsfälle sichergestellt. Beispiele unterschiedlicher Komplexität existieren, wie in Abschnitt 2.2.1 beschrieben.

Implementierung und Generierung wurden deutlich voneinander abgegrenzt. Generierter Code wird getrennt von der restlichen Implementierung gespeichert. Diese Trennung bereits im Generator sorgt zuverlässig für ein konfliktfreies Entwickeln. Das Datenmodell selber sowie Klassen für seine lokale und entfernte Speicherung werden vollautomatisch erzeugt. Klassen des Datenmodells werden getrennt von Funktionen des Rahmenwerks in unterschiedlichen Namensräumen vorgehalten. Diese Trennung erlaubt die einfache Wartung und

einen schnellen Blick auf Domänenspezifische Anteile. Anwender können das Datenmodell direkt nutzen. Einer der wenigen notwendigen Erweiterungspunkte existiert für den manuellen Abgleich nach Erkennung von Konflikten bei der Synchronisierung. Der Entwickler kann an dieser Stelle direkt auf alle Konflikte inklusive der Details zum lokalen und entfernten Objekt zugreifen. Die MOHITO API stellt die Trennung sicher. Eine Prüfung des Konzepts erfolgt manuell, die technische Zusicherung der Trennung der Anteile durch die verwendete Eclipse IDE.

Die Einfachheit des Modellierens wurde durch Testpersonen bei den Projektpartnern getestet und sichergestellt. Im Vergleich zu einer manuellen Implementierung enthielt der erzeugte Code weniger Fehler und deckte eine größere Anzahl an Sonderfällen ab. Das Schreiben korrekten Codes ist somit auch schneller.

Die syntaktische Validierung des Modells erfolgt vollautomatisch durch Prüfung auf korrekte Sprache. Fehlermeldungen werden direkt in den Modelldateien angezeigt, sollten durch eine manuelle Anpassung Inkonsistenzen entstanden sein.

Der Austausch und die Versionierung von Modellinstanzen wurde durch Verwendung des Eclipse Modeling Framework sichergestellt. Die Serialisierung im Format XMI erlaubt sowohl die natürlichsprachliche Erfassung als auch vollautomatische Verarbeitung. Zusätzlich können CDO oder EMFStore als Technologien verwendet werden, welche die Verwendung komplexer und verteilter Modelle erleichtern. Diese Qualitätskriterien wurden per Konstruktion bei der Technologieauswahl und den Realisierungsentscheidungen sichergestellt.

CAS spezifische Erweiterungen der MOHITO DSL

Für den CAS Anwendungsfall – den mobilen xRM Demonstrator – waren umfangreiche Erweiterungen der MOHITO Annotationen nötig, die weil sie nicht von anderen Anwendungen genutzt werden können auch nicht Teil der MOHITO DSL sind. Die Vollständigkeit und Korrektheit dieser Annotationen wurden durch den Reverseengineering-Ansatz und die damit verbundenen Tests (beispielsweise auch durch das Zurückkonvertieren und den Vergleich mit der Ausgangsdatentypdefinition) sichergestellt. Ferner zeigt die erfolgreiche Implementierung des mobilen xRM Demonstrators die Korrektheit der Annotationen. Hierbei ist Hervorzuheben, dass die Erweiterung des Demonstrators um weitere Datentypen problemlos war und gezeigt hat, dass die Annotationen (MOHITO DSL+CAS Erweiterungen) vollständig sind.

2.3 Dokumente

Die im Projekt entstehenden Dokumente wurden vor allem die auf ihre Verständlichkeit, Vollständigkeit und Richtigkeit geprüft. Dies geschah primär durch gemeinsame Templates und

manuelle Reviews der Dokumente selbst. Generell wurde jedes Dokument von mindestens einem Vertreter jedes Projektpartners geprüft.

Gerade weil die MOHITO Plattform als Open Source zur Verfügung stehen wird, ist es relevant, dass die Meta-Modelle, exemplarische Modellinstanzen, die Transformation und der generierte Code so dokumentiert sind, dass auch ein ungeschulter Entwickler dies verstehen und anwenden kann. Auch für diese textuellen Teile wurden die genannten Qualitätssicherungsmechanismen eingehalten.

3 Integration

Im Zuge der Implementierung der Demonstratoren, sowie um die korrekte Funktionsweise des MOHITO Frameworks sicherzustellen, wurden zwei Anbindungen an bestehende Server implementiert: Den CAS Open Server und den MML von B2M. In beiden Fällen konnte die Integration problemlos umgesetzt und damit gezeigt werden, dass MOHITO auch im industriellen Umfeld leicht an bestehende Systeme angebunden werden kann.

3.1 Integration in CAS OPEN

Während der Entwicklung der xRM App wurde permanent gegen ein live System (einen laufenden Standard CAS Open Server entwickelt. Dieser hat eine klar definierte API (dessen REST Schnittstelle in der xRM App intensiv genutzt wurde. Durch das stringente Testen der xRM App mit Daten des live Systems wurde die korrekte Funktionsweise sichergestellt und eine potentielle, dauerhafte Nutzung der MOHITO Prototypen möglich gemacht. Ein weiterer wichtiger Integrationsaspekt ist die Generierung nicht nur von Client-seitigen Datenhaltungsteilen, sondern auch von Server-seitigen Teilen. Wenn es ein zentrales Datenmodell gibt, auf dessen Basis sowohl Client- als auch Serverteile generiert werden, ist sichergestellt, dass Änderungen am Datenmodell zu keinen Inkonsistenzen führen. So wird auf der Basis des MOHITO Frameworks aus dem ecore Datenmodell ein XML Datenmodell generiert, auf dessen Basis der CAS OPEN Server arbeitet. Die erfolgreiche und effiziente Integration an die CAS Server Komponente ist daher gewährleistet und ausgiebig getestet.

3.2 Integration in den MML

Der MML diente im MOHITO Projekt sowohl als Teil der Referenzimplementierung, als auch als beispielhafte Bestandsanwendung, an welcher die generelle Integration serverseitiger Komponenten in den MOHITO-Softwarestack analysiert werden konnte. Gerade in diesem Zusammenhang war es wichtig die bestehenden Mechanismen der Datenhaltung des MML genauer zu betrachten, um eine entsprechende Anbindung bzw. Migration hin zu einem MOHITO Datenhaltungsstack zu gewährleisten. Hierzu war es nötig, die Datenhaltung des MML auf eine Technologie zu migrieren, welche zur „Java Persistence API“ kompatibel ist, da der von MOHITO verwaltete Datenhaltungsstack auf dieser Technologie als Basis aufsetzt. Die Java Persistence API (JPA) ist ein Java Persistenz Standard, welcher Speichertechnologie agnostisch ist, und nur das Persistieren von simplen Java Objekten in objektrelationale Datenbanken vorsieht. Der Standard wurde von der EJB 3.0 Expert Group entwickelt und wurde im Mai 2006 erstmals veröffentlicht. (Oracle Technology Network- Java, kein Datum) Er kann heute im Java Umfeld als wichtigster Persistenz Standard angesehen wer-

den, für den es etliche Implementierungen gibt. Für den MML wurde das Hibernate Framework gewählt, welches eine komplette Implementierung des JPA Standards beinhaltet. Dies stellt eine stabile Lauffähigkeit des MMLs sicher.

Der Standard definiert Persistenz über ein objektrelationales Mapping. Das bedeutet Java Objekte werden durch definierte Regeln auf relationale Datenbanken abgebildet und können von dort auch wieder geladen und instanziiert werden. Die Beschreibung, welche Felder einer Klasse wie und auf welche Spalten abgebildet werden, kann in Form von XML Metadaten oder per Java Annotation realisiert werden. Im MML wurde die Datenhaltung durch XML Metadaten beschrieben.

Nachdem diese Umstellung vollzogen war, musste noch eine striktere Trennung von Programmlogik und eigentlicher Datenhaltung vollzogen werden. Dies geschah in Teilschritten im Zuge mehrerer Refactorings. Am Ende dieses Prozesses stand dann eine MML Server Komponente, die voll in MOHITO integrierbar war.

Die korrekte Funktionsweise und erfolgreiche Anbindung an den MML wurde explizit durch den im Rahmen des Projektes entwickelten Demonstrator sichergestellt. Hierbei garantieren automatische J-Unit Tests die Korrektheit der Basisoperationen der Schnittstelle und manuelle Tests die korrekte Funktionsweise der Abfolge von Operationen auf den Daten.

4. Zusammenfassung

Es hat sich gezeigt, dass die im Rahmen von MOHITO anfänglich definieren Qualitätssicherungsmaßnahmen sinnvoll waren. So gut wie alle anfänglich definierten Maßnahmen wurden eingehalten und damit eine sehr gute Qualität der Ergebnisse gewährleistet. Gerade im Bereich der automatischen Tests von Modellartefakten gibt es nach wie vor Spielraum für Verbesserung. Ebenso wie im Falle von Dokumentationen wurde hier maßgeblich auf manuelle Reviews und Tests zurückgegriffen. Die Qualität und der Funktionsumfang des MOHITO Frameworks wurde insbesondere durch die erfolgreiche Umsetzung zweier Demonstratoren und der damit verbundenen (automatischen) Tests sichergestellt und in Form von L2.2 Dokumentation der Validierungsergebnisse (Gailus, Hübsch, Schwichtenberg, & Groenda, 2014) mit Fokus auf den Funktionsumfangs beschrieben.

5. Literaturverzeichnis

Gailus, E., Hübsch, G., Schwichtenberg, A., & Groenda, H. (2014). *L2.2 Dokumentation Validierungsergebnisse.* Von http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details abgerufen

Gailus, E., Klatt, B., Hübsch, G., Krogmann, K., & Schwichtenberg, A. (kein Datum). *L 2.1: Dokumentation der Anwendungsszenarien, der Anforderungen und des Stands der Technik.*

Klatt, B., Gailus, E., Hübsch, G., & Schwichtenberg, A. (2013). *L 5.1: Migrationsstrategie für Bestandsanwendungen zur MOHITO Referenzarchitektur.*

Object Management Group. (2011). *QVT 1.1 Spezifikation*. Von http://www.omg.org/spec/QVT/1.1/ abgerufen

Oracle Technology Network- Java. (kein Datum). (Oracle) Abgerufen am 24. 8 2012 von http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

Schwichtenberg, A., Gailus, E., Klatt, B., & Hübsch, G. (kein Datum). *L9.1 QM-Maßnahmen und Tests.*

Schwichtenberg, A., Hübsch, G., Gailus, E., & Groenda, H. (2014). *L5.2 Migrationsstrategie für die sanfte Überführung von Bestandssoftware zu modellbasierten Multi-Plattformanwendungen mittels MOHITO.*

