Modellgetriebene homogenisierte Datenhaltung und Synchronisation mit vorhersagbaren Eigenschaften für plattformübergreifende Anwendungen

Förderkennzeichen: 01 S12012

AP 9

QM-Maßnahmen und Tests

Autor: A. Schwichtenberg (CAS)
Co-Autoren: E. Gailus (B2M), G. Hübsch (CAS), B. Klatt (FZI)
Fertiggestellt am: 30. 11. 2012
Schlagworte: Qualitätsmanagement

Projektpartner

GEFÖRDERT VOM

Änderungshistorie

Version	Änderungen von	Datum	Anmerkung
Draft	A.S.	5.10.12	Erste Ideen
	B.K.	22.10.12	Feedback und Ergänzungen
	A.S.	24.10.12	Veränderte Struktur + Zuständigkeiten
	A.S.	03.12.12	Qualitätskriterien
	B.K.	04.12.12	Artefakte + QS Mechanismen, Review
1.0	A.S.	08.01.2013	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 erarbeiteten 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). Des Weiteren wird beschrieben, wie diese Maßnahmen umgesetzt, d.h. wie die entwickelten Software- und Modellartefakte daraufhin geprüft werden können.

Die Anwendung der QS-Maßnahmen und die Durchführung der entsprechenden Tests finden im Rahmen von Unterarbeitspaket 9.2 statt. Dabei werden entsprechend dem jeweiligen Fortschritt des Projekts in den Bereichen des DSL-Entwurfs sowie der Generatorableitung und der Framework-Entwicklung kontinuierlich jene QS-Detailmaßnahmen und Tests ergänzt, die sich während des Projekts als zusätzlich notwendig erwiesen haben bzw. im Rahmen von AP 9.1 noch nicht planbar waren.

Prinzipiell sollen für alle Artefakte, die innerhalb des MOHITO Projektes erstellt werden, Maßnahmen definiert werden, die 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 Kapitel 2 werden alle im Rahmen dieses Projektes relevanten Qualitätssicherungsmöglichkeiten und -prozesse überblicksartig beschrieben. In Kapitel 2 werden alle im Projekt zu entwickelnden Artefakte kurz beschrieben. Dies bildet die Grundlage für die Definition der Qualitätskriterien, die in Kapitel 3 beschrieben werden und die die Basis für die Bewertung der Qualität darstellen. Die Qualitätskriterien sind gegliedert in unterschiedliche Nutzergruppen, da diese teilweise ganz unterschiedliche Anforderungen an die Qualität stellen, bzw. unterschiedliche Aspekte beleuchten.
Im Kapitel 4 wird näher auf die spezifischen Mechanismen zur Qualitätssicherung eingegangen und für jedes Artefakt werden die geeigneten Methoden definiert. Da die im Rahmen des MOHITO Projektes entstehenden Ergebnisse auch in die Produkte der Partner B2M und CAS Software AG übernommen werden soll, so fern das möglich ist, werden in Kapitel 5 für diesen Prozess Qualitätssicherungsmechanismen definiert, um eine möglichst einfach Übernahme der Projektergebnisse zu gewährleisten. Eine umfassende Migrationsstrategie wird

im Bericht L5.1 Migrationsstrategie für Bestandsanwendungen zur MOHITO Referenzarchitektur und L5.2 Migrationsstrategie für die sanfte Überführung von Bestandssoftware zu modellbasierten Multi-Plattformanwendungen mittels MOHITO innerhalb des Arbeitspaketes fünf des MOHITO Projektes beschrieben.

2 Artefakte

Innerhalb des MOHITO Projekts werden Modell-bezogene und Code-bezogene Artefakte sowie Dokumentation entwickelt. Was genau darunter zu verstehen ist wird in diesem Abschnitt beschrieben. In den folgenden Kapiteln werden Qualitätskriterien und –sicherungsmaßnahmen im Bezug zu diesen Artefakt-Typen erläutert.

2.1 Modell-Entwicklungsartefakte

In MOHITO werden ein modellgetriebenes Framework und eine modellgetriebene Vorgehensweise entwickelt. Dementsprechend entstehen modellbezogene Artefakte, für die eine Qualitätssicherung notwendig ist.
Grundlage sind Meta-Modelle. Sie werden auch als abstrakte Syntax bezeichnet und definieren einen Sachverhalt. Vergleichbar mit einer klar spezifizierten Grammatik können in MOHITO mit Meta-Modellen beispielsweise Datentypen und die Anforderungen an Ihre Speicherung und Übertragung definiert werden.
Domänen-spezifische Sprachen (DSL) stellen für eine abstrakte Syntax, sprich das Meta-Modell, eine konkrete Syntax dar, in der etwas mit dem Meta-Modell beschrieben werden kann. Hierbei können sowohl textuelle als auch graphische Sprachen genutzt werden. Für beide ist eine Qualitätssicherung gleichermaßen notwendig.
Als dritten Untertypen von Modell-Entwicklungsartefakten werden konkrete Modell-Instanzen, beispielsweise für die Referenz-Implementierungen entwickelt. Hierbei handelt es sich um konkrete Modelle, die einen bestimmten Sachverhalt in einer konkreten Syntax (DSL) für ein bestimmtes Meta-Modell (abstrakte Syntax) spezifizieren.

2.2 Code-bezogene Artefakte

Im MOHITO Projekt werden diverse, unterschiedliche Implementierungs-Artefakte und damit auch Code-bezogene Artefakte entstehen.
Zum einen wird die MOHITO Plattform selbst mit einer Laufzeitumgebung in Form von Bibliotheken (libs) entwickelt und als OpenSource bereitgestellt. Gegen diese plattformspezifi-

schen Bibliotheken werden dann die Anwendungen entwickelt. Gegen diese Bibliotheken wird dann der Code aus dem modellgetriebenen Vorgehen generiert und ausgeführt.
Zur Erstellung der Modelle werden Editoren auf Basis des Eclipse Modelling Frameworks entwickelt. Dies bietet bereits viele Grundlagen, die es erlauben mittels Java Code solche domänenspezifischen Modelleditoren zu entwickeln. Hierzu muss jedoch eigener Java Code geschrieben und im Laufe des Projekts qualitätsgesichert werden.
Um von den Modellen zu lauffähigem Code zu gelangen werden Code-Generatoren verwendet, die eine Modellinstanz in ein textuelles Format (beispielsweise Java Programm Code) überführen. Solche Generatoren arbeiten in der Regel mit Templates, die die zu erzeugende Ausgabe bestimmen. Beides, sowohl die Generatoren als auch die Templates, müssen Qualitätsgeprüft werden, da sie kritische Bestandteile in der modellgetriebenen Vorgehensweise darstellen.
Zu guter Letzt bleibt noch das Ergebnis der Code-Generierung. Der hierbei entstehende Code sollte auch wieder qualitätsgesichert werden. Hierbei handelt es sich um die Implementierung der eigentlichen, fachlichen Anwendung. Die Qualitätssicherung dieses Codes ist unter zwei Aspekten zu betrachten. Zum einen aus Sicht des MOHITO Projekts, das dafür sorgen tragen muss verständlichen und korrekten Code zu erzeugen. Zum anderen aus Sicht der Nutzers des MOHITO Frameworks und der MOHITO Werkzeugkette, dessen fachliche Richtigkeit in Bezug auf das Generotergebnis und dessen Anpassbarkeit gegeben sein muss. Die genauen Qualitätskriterien werden in Abschnitt 3 beschrieben.

2.3 Begleitdokumentation, Beispielprojekt

Im MOHITO Projekt entstehen drei verschiedene Arten von Dokumenten: Vereinbarte Lieferungen an den Projektträger, interne Arbeitsdokumente zur Software-Entwicklung wie zum Beispiel zum Softwareentwurf, und Anwendungsdokumentation, die mit dem MOHITO Framework und Werkzeugen als OpenSource Ergebnisse der Öffentlichkeit zur Verfügung gestellt werden.
Im Kontext der Qualitätssicherung, wie sie in diesem Dokument beschrieben wird, stehen vor allem Dokumente von letzterem Typ im Fokus. Hierzu zählen sowohl Bedienungsanleitungen für das Framework, sowie Beispiele und Beispielprojekte mit denen Interessierte die Verwendung der Projektergebnisse nachvollziehen und sich in deren Anwendung einarbeiten zu können.

3 Qualitätskriterien

Im Folgenden werden für die unterschiedlichen Artefakte Qualitätskriterien definiert, so wie sie sich aus der Sicht der unterschiedlichen MOHITO Nutzergruppen darstellen: MOHITO Plattform Entwickler, Anwendungsentwickler (Nutzer der Plattform) und End-Nutzer der Applikation. Aus den im Folgenden beschriebenen Kriterien lassen sich zum einen Qualitätssicherungsmechanismen ableiten und zum anderen lässt sich zu einem späteren Zeitpunkt die Qualität der entwickelten Artefakte evaluieren.[1]
Um die Qualitätskriterien und damit die Zielsetzung der Qualitätssicherungsmaßnahmen möglichst präzise formulieren zu können, wird definiert, welche Benutzergruppen welche Artefakte benutzen. Prinzipiell gibt es im Rahmen von MOHITO zwei Typen von Benutzern mit den folgenden Rollen, bzw. Aufgaben.

MOHITO Plattform Entwickler:

- Stellt Laufzeitumgebung für unterschiedlichen Plattformen wie Android und iOS zur Verfügung (gegen die dann generiert wird)
- Entwickelt Tools (z.B. Modell Editor)
- Entwickelt das Metamodell (weiter)
- Entwickelt die Transformatoren (weiter)
- Definiert die Hooks

Anwendungsentwickler:

- Benutzt die Plattform
- Erstellt Domain-spezifische Daten-Modelle
- Füllt Hooks und Templates

3.1 Qualitätskriterien für Modelle

Aus der Sicht des Plattform-Entwicklers

Qualitätskriterien für Meta-Modelle (abstrakte Syntax)

- **Verständlichkeit des Modells:** Das Modell soll in einer Sprache abgebildet werden, die möglichst intuitiv zu verstehen ist. Eine Anlehnung an UML ist hier sinnvoll. Die graphische Darstellung soll übersichtlich sein und es sollen detaillierte Darstellungsmöglichkeiten zur Verfügung stehen, z.B. durch Aufklappen. Auch sollte der Aufwand,

[1] Siehe auch L9.2. "Testaufbau und Durchführung, Ergebnisdokumentation, Testfallbeschreibung und QS-Kriterien für Modelle, Plattform und Generat", fällig zu Projektende.

um die Modellierungssprache zu verstehen und anwenden zu können, möglichst gering sein.

- **Ausdrucksmächtigkeit und Vollständigkeit:** Das Meta-Modell sollte alle Modellelemente definieren, die für die Modellierung der Datenhaltungsschichten von Multi-Plattform Anwendungen notwendig sind.
- **Ein exemplarisches Gesamt-Modell** sollte in einem Eclipse Beispielprojekt zur Verfügung gestellt werden.

Aus der Sicht des Anwendungsentwicklers

Qualitätskriterien für die DSL (konkrete Syntax)

Mächtigkeit der DSL: Auf Basis eines erstellten Datenmodells muss es möglich sein, die diversen (Code-) Generate für alle unterstützen Plattformen in den jeweiligen Ausprägungen zu generieren. Auf der anderen Seite sollte sie möglichst einfach handhabbar und einfach zu erlernen sein. Dies erfordert, dass die DSL so umfangreich wie nötig, aber auch so kompakt wie möglich sein soll. Vor allem sollen unnötigen Modellierungsmöglichkeiten konsequent vermieden werden.

- **Grenze/Konsistenz zwischen Generierung und manueller Implementierung:** Nicht alle Laufzeitartefakte werden generiert. Es muss möglich sein, Erweiertungspunkte wie Templates und Hooks zu generieren und diese dann mit bestehenden Implementierungen zu füllen. Ein entscheidendes Qualitätskriterium ist dabei die Vollständigkeit, d.h. die Abdeckung möglichst aller vom Verwender des generierten Codes für typische Anwendungsfälle benötigten Erweiterungspunkte. Hierfür sollte es geschickte Mechanismen in der Entwicklungsumgebung geben, die beispielweise das schnelle Auffinden der generierten Erweiterungspunkte unterstützen. Die manuelle Implementierung darf nicht durch eine erneute Generierung überschrieben werden. Wie kann die Konsistenz der manuellen Implementierung mit der automatischen Generierung überprüft werden? Wie wird gegen bestehende APIs modelliert?
- **Einfachheit des Modellierens:** Das Modellieren sollte prinzipiell einfacher und schneller sein als das Quellcode-Schreiben.
- **Syntaktische Validierung des Modells:** Es muss möglich sein, das Modell auf syntaktische Korrektheit hin zu validieren.
- **Modellinstanzen** sollten leicht zwischen Entwicklern austauschbar und versionierbar sein. Außerdem soll es möglich sein, Modellinstanzen sehr einfach anlegen zu können. Wenn es Änderungen am Modell gibt, die zu Konflikten in den Instanzen führen, soll wenn möglich darauf hingewiesen werden.

3.2 Qualitätskriterien für Code-bezogene Artefakte

Aus der Sicht des Plattform Entwicklers

Qualitätskriterien für die MOHITO Plattform

- **Laufzeitumgebung und Interpreter:** Fehlerfreiheit, Vollständigkeit, Robustheit und Performance sind Schlüsselkriterien für die Qualität der Laufzeitumgebung und des Interpreters. Ein weiteres wichtiges Qualitätskriterium ist der Umfang der Maßnahmen für Laufzeitanalysen, z.B. zur Unterstützung der Fehlersuche durch Logging.
- **Strikte Trennung zwischen Generat und manuellen Ergänzungen:** Die Trennung zwischen generierten und nicht-generierten Teilen soll dem Plattform-Entwickler eine Weiterentwicklung der Transformatoren ermöglichen ohne dass die Gefahr besteht, dass die von Hand entwickelten Teile beeinträchtigt oder geändert werden.
- **Dokumentation:** Gerade weil die MOHITO Plattform als OpenSource zur Verfügung stehen wird, sollten die zur Verfügung gestellten Libraries gut strukturiert und dokumentiert sein, so dass auch ein ungeschulter Entwickler diese verstehen und anwenden kann. (s. Kapitel 3.3 Dokumentation)

Qualitätskriterien für Modell-Editoren:

- **Einfache Bedienbarkeit:** Die Editoren sollen möglichst einfach und intuitiv zu bedienen sein. Hierfür eignet sich eine Anpassung an bestehenden Paradigma wie z.B. das User Interface Design von Eclipse.

Qualitätskriterien für Code-Generatoren

- **Performance:** Die Performanz der Transformation ist prinzipiell eher unkritisch (hier sind Minuten akzeptabel).
- **Lesbarkeit des Codes:** Bei der Entwicklung der Code-Generatoren sollen strikte Codekonventionen eingehalten werden, um die Weiterentwicklung und Wartung zu vereinfachen. Hierzu zählt u. A. die Verwendung sinntragender Bezeichnungen für Variablen, Methoden, Klassen und Packages, sowie eine übersichtliche Formatierung des Codes.
- **Dokumentation:** Da die Generatoren von unterschiedlichen Entwicklern weiterentwickelt werden können, müssen strikte Vorgaben zur Quellcode Dokumentation eingehalten werden. (s. Kapitel 3.3 Dokumentation)

Qualitätskriterien für Templates

- **Lesbarkeit des Codes:** Bei der Entwicklung der Templates sollen strikte Codekonventionen eingehalten werden, um die Weiterentwicklung und Wartung zu vereinfa-

chen. Hierzu zählt u. A.: die Verwendung sinntragender Bezeichnungen und eine übersichtliche Formatierung.

Aus der Sicht des Anwendungsentwicklers

Qualitätskriterien für Code-Generate

- **Lesbarkeit des Code-Generats:** Eine häufige Schwierigkeit bei generiertem Code ist das Fehler-Finden, da die Gefahr besteht, dass der generierte Code im Vergleich zu manuell erzeugtem Code insgesamt schwerer verständlich ist. Z.B. haben Variablenbezeichnungen oftmals keine sinntragenden Bezeichnungen. Auch zu klären ist die Frage, wie man anhand des Codes zurückverfolgen kann, ob der Fehler aus der Transformation oder aus dem Modell resultiert.
- **Trennung und Konsistenz zwischen Generierung und manueller Implementierung:** Nicht alle Laufzeitartefakte werden generiert. Es muss möglich sein, Templates und Hooks zu generieren und diese dann mit bestehenden Implementierungen zu füllen. Hierfür sollte es geschickte Mechanismen in der Entwicklungsumgebung geben und die manuelle Implementierung darf nicht durch eine erneute Generierung überschrieben werden. Es wäre wünschenswert, wenn es Mechanismen zur Überprüfung der Konsistenz der manuellen Implementierung mit der automatischen Generierung gäbe. Es soll gegen bestehende APIs modelliert werden.
- **Automatische Tests:** Für den generierten Code müssen Tests in der jeweiligen Sprache (z.B. JUnit Tests) geschrieben werden können bzw. durch Generierung erzeugt werden. Idealerweise sollen auch plattformspezifische Funktionen getestet werden können und es sollte "echte" Testdaten geben, so dass die Tests aussagekräftig sind.
- **Exception handling:** Idealerweise sollen auch Teile des Exception handlings generiert werden können.

3.3 Dokumentation

Gerade weil die MOHITO Plattform als Open Source zur Verfügung stehen wird, ist es erwünscht, 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 sollen geeignete Qualitätssicherungsmechanismen definiert und eingehalten werden.

4 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 auf 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 2).

Alle Artefakte werden vor ihrer Veröffentlichung wechselseitig durch die Projektpartner qualitätsgesichert. 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“ geprüft.
Eine Freigabe von Artefakten erfolgt nach positiver Evaluation. In allen anderen Fällen geht die Überarbeitung auf den entwickelnden Projektpartner über.

Im Folgenden wird für alle Artefakte, die im MOHITO Projekt entwickelt werden, definiert, welche Qualitätssicherungsmechanismen sich eignen.

4.1 Code-Entwicklungsartefakte

Für die Qualitätssicherung von Code-Artefakten wird auf die etablierten Standards aus der Software-Entwicklung zurückgegriffen.

Bereits im frühen Stadium der Entwicklung gilt es ein gutes Design der Software sicherzustellen und Fehlentscheidungen, die später teuer korrigiert werden müssen zu verhindern. Hierzu werden frühzeitig Reviews von Architektur und Entwurf durchgeführt und diese gegebenenfalls angepasst. Solche Reviews werden manuell und in Teamarbeit, auch Projektpartner-übergreifend abgehalten.

Unit Tests werden genutzt um auf feingranularer Ebene, wie zum Beispiel Klassen und Methoden den Code automatisiert und reproduzierbar zu prüfen. Eine wichtige Metrik hierbei ist die Code-Coverage, die eine Rückmeldung dazu gibt, wie gut der Code durch automatisierte Unit-Tests abgedeckt ist. In der Regel ist eine Abdeckung von ca. 80% anzustreben. Die

Erfahrung hat gezeigt, dass für eine höhere Abdeckung der Entwicklungsaufwand für die Tests nicht im Verhältnis zur Automatisierung steht.

Zusätzlich zu den Unit-Tests werden Coding-Guidelines geprüft um einen einheitlichen Code Stil und damit die Lesbarkeit des Codes sicherzustellen. Zusätzlich werden Metriken beispielsweise über die Komplexität des Codes erhoben um hier gegebenenfalls komplexe Codebereiche erkennen und überarbeiten zu können. Hilfreiche Tools hierzu sind Checkstyle [CS], Google Code Pro Analytix [CP] und FindBugs [FB].

Einen Sonderfall bei Code-Entwicklungsartefakten stellen generierte Artefakte dar. Hierzu zählen zum einen Code, der durch den Code Generator erzeugt wurde, zum anderen aber auch Deskriptor- und Konfigurationsdateien, die durch eine Modell-zu-Text-Transformation erstellt wurden.

Für den generierten Code sollten geeignete Qualitätssicherungsmaßnahmen wie bereits im letzten Abschnitt beschrieben gewählt werden.

Für Deskriptor- und Konfigurationsdateien sollte, sofern vorhanden, gegen eine entsprechende Formatspezifikation geprüft werden. Handelt es sich beispielsweise um ein bestimmtes XML Format, für das eine XML Schema Definition vorliegt, so sollte im Rahmen der Tests eine Datei für ein Referenzmodell generiert und gegen diese Schema Definition geprüft werden.

Zusätzlich zu den automatischen Prüfungen von Implementierungen wird es bei ca. 80% Fertigstellung manuelle Reviews der Code-Artefakte geben. Bei diesen Reviews werden nochmals manuell kritische Stellen der Implementierung gemeinsam gesichtet und geprüft. Hierbei ist neben der Korrektheit auch die Verständlichkeit des Codes im Fokus.

4.2 Modell-Entwicklungsartefakte

Aufgrund des modellgetriebenen Vorgehens im Projekt werden neben den klassischen Code- auch Modell-Entwicklungsartefakte entstehen. In diesem Gebiet gibt es noch deutlich weniger, etablierte Möglichkeiten zur automatischen Prüfung. Daher wird die Qualität von Modell-Entwicklungsartefakten primär durch manuelle Reviews sichergestellt. Bei diesen Reviews werden sowohl das übergreifende Modellkonzept, als auch detailliertere Aspekte, wie Referenzen zwischen Elementen und Elementattributen gesichtet.

Ergänzend zu den manuellen Reviews werden im Metamodell soweit wie möglich und sinnvoll Modellbedingungen (Constraints) eingebracht, die dann von der Modellierungsumgebung validiert werden können.

Der Bereich der Qualitätsmetriken für Modell-Entwicklungsartefakte ist noch relativ jung und immer noch Forschungsschwerpunkt. Aus Sicht des MOHITO Projekts wird dieser Bereich Projektbegleitend beobachtet und wenn möglich werden Qualitätsmetriken für Modelle, die

während der Projektlaufzeit publiziert und als sinnvoll anerkannt werden ins das Projekt übernommen.

4.3 Dokumente

Die im Projekt entstehenden Dokumente werden vor allem die auf ihre Verständlichkeit, Vollständigkeit und Richtigkeit geprüft. Dies geschieht primär durch gemeinsame Templates, ein gemeinsames, kontinuierlich weiterzuentwickelndes Glossar und manuelle Reviews der Dokumente selbst. Generell gilt, dass jedes Dokument von mindestens einem Vertreter jedes Projektpartners geprüft werden soll.

5 Integration

Um die Projektergebnisse, vor allem den Prototyp, in die bestehenden Applikationen integrieren zu können, müssen bei der Generierung die Schnittstellen des Zielsystems bekannt sein. Es muss gegen die Schnittstellen (API) programmiert werden, d.h. es müssen auch Gegebenheiten der Architektur und bestehende Constraints beachtet werden. Die entwickelten Libraries müssen integrierbar, erweiterbar und wartungsfreundlich sein, um eine Weiterentwicklung und langfristige Nutzung zu gewährleisten. Zu diesem Zweck werden von CMMI in [CM] die folgenden Qualitätssicherungsmechanismen vorgeschlagen:

1. Eine Tabelle mit Relationen zwischen den Produkt Komponenten und dem externen System
2. Eine Tabelle mit Relationen zwischen den Produkt Komponenten
3. Liste mit vereinbarten Interfaces für jedes Paar von Produkt Komponenten (sofern dies sinnvoll ist)
4. Berichte der Infertace-Kontroll-Arbeitsgruppen
5. Aufgaben Definition für Interface Updates
6. API (Application Program Interface)
7. Interface Beschreibungen immer auf dem neuesten Stand

Aufgrund der Größe des Projektes werden nicht alle dieser von CMMI vorgeschlagenen Qualitätssicherungsmechanismen eingehalten – z.B. nicht Punkt 4 – wenn möglich sollen aber insbesondere 1., 2., 3., 6. Und 7. erstellt werden.

Im Rahmen einer iterativen Vorgehensweise sollen verschiedene Entwicklungsstufen eingeplant werden, so dass Erkenntnisse aus frühen Entwicklungsstufen in die weitere Entwicklung einfließen können. Die Ergebnisse der einzelnen Entwicklungsstufen sollen ausführlich getestet werden (auch manuell).

Wenn ein Teil einer Software in ein bestehendes Produkt übernommen werden soll, so müssen bestehende Strukturen und Zuständigkeiten beachtet werden. Entsprechend müssen Verantwortlichkeiten definiert und die Ausführung der notwendigen Arbeiten geplant werden.

Bei der Erweiterung existierender Produkte liegt der Schwerpunkt auf der guten Integrationsfähigkeit der neuen Software-Komponenten auf technischer Ebene. Dies gilt nicht nur für die technische Kompatibilität mit existierenden Software-Komponenten, sondern auch für ihre Einbindung in existierende Infrastrukturen und Tools wie Versionsverwaltungssystem, Build-Server und Testumgebungen.

Beim Ersetzen existierender Produkte oder Produktbestandteile ist es zusätzlich notwendig, die Übereinstimmung mit funktionalen und nicht-funktionalen Spezifikationen der ersetzten Komponenten zu prüfen. Dies kann beispielsweise durch die Übernahme existierender Tests geschehen. Ferner müssen bestehende Lizenzierungsmechanismen eingehalten werden.

5.1 Integration in OPEN

Ziel ist die prototypische Umsetzung für die OPEN Plattform der CAS Software AG. Damit befinden wir uns im Umfeld einer klassischen Client-Server-Architektur. Um die zu entwickelnde Client-Anwendung testen zu können, muss schon während der Entwicklung ein Server zur Verfügung stehen, auf dem ein Branch der aktuellen OPEN Plattform liegt. Diese stellt die API zur Verfügung, gegen die entwickelt wird, bzw. die der mobile xRM Client benutzt. Serverseitig werden u. A. die Technologien OSGI und Spring eingesetzt, die eine stufenweise Entwicklung und Funktionsüberprüfung unterstützen.

5.2 Integration in den MML

Die prototypische Integration in den „Mobile Mediation Layer“ (MML) der B2M Software AG wird stufenweise erfolgen. Der MML ist hierbei die Serverkomponente, welche Dienstanfragen geeigneter mobiler Klienten bearbeitet. Clientseitig wird zur Modellintegration die „HereIAm“ App eingesetzt, eine Android Applikation, welche mit dem MML interagiert und im Projekt schon als Referenzimplementierung dient.

Serverseitig kommen Technologien wie Hibernate und Spring zum Einsatz. Clientseitig wird der Datenhaltungsstack durch ORMLight realisiert. Damit stellen beide Software Komponenten eine geeignete Plattform für die stufenweise Integration des Mohito Frameworks dar und bieten zudem jederzeit die Möglichkeit die Funktionalität des entstehenden Frameworks zu verifizieren.

6 State-of-the-art QM

6.1 Architektur

Die allgemeine Analyse von Software Architekturen findet heutzutage vor allem durch manuelle Assessments und Reviews statt.

In der Vergangenheit wurden standardisierte Vorgehensweisen entwickelt, von denen sich einige wenige etabliert haben. Zu den prominentesten Vertretern gehört die „Architecture-Level Modifiability Analysis“ (ALMA) [ALMA], die einen Workshop-orientierten Charakter hat und die Anpassbarkeit von einer Software-Architektur unter Berücksichtigung von als wahrscheinlich eingestuften Änderungsszenarien analysiert. Hierbei werden unterschiedliche Projektbeteiligte zur Erhebung der Änderungsszenarien interviewt. Nachfolgend werden Softwarearchitekten und -Entwickler, zu ihrer Einschätzung der Anpassbarkeit für diese Szenarien interviewt.

Automatische Analysen von Softwarearchitektur sind heutzutage noch wenig verfügbar. Sie lassen sich grob in die zwei Kategorien Architekturkonformität und Architekturqualität aufteilen.

Analysen der Architekturkonformität prüfen die Implementierung einer Software gegen vorher definierte Regeln. Hierzu gehört beispielsweise das Werkzeug JDepend [JDepend], das es erlaubt die Abhängigkeiten zwischen Paketen automatisiert, auch im Rahmen eine automatisierten Build-Prozesses zu prüfen. Zum anderen gibt es Architekturanalysen, die manuell auf einem Architekturentwurf durchgeführt werden und oftmals einen speziellen Qualitätsaspekt untersuchen.

Analysen zur Architekturqualität dienen primär zur Unterstützung des Softwarearchitekten bei seiner Entwurfstätigkeit. Werkzeuge in diesem Bereich fokussieren sich in der Regel auf einen Qualitätsaspekt. So dient zum Beispiel das Werkzeug Palladio zur Analyse von Performanceeigenschaften einer Softwarearchitektur [Palladio]. Auf Basis eines Komponentenentwurfs, einer Ausführungsumgebung, und eines Nutzungsprofils simuliert Palladio das Verhalten der entworfenen Software und gibt eine Rückmeldung zu Ressourcen- und Zeitverbräuchen.

6.2 Code

Das Qualitätsmanagement von Programmcode lässt sich heutzutage in vier Bereiche aufteilen: Code Konventionen, Tests, Metriken, und Continuous Integration.

Code Konventionen

Code Konventionen dienen wie in Abschnitt 4.1 beschrieben der leichteren Zusammenarbeit von mehreren Entwicklern. Sie lassen sich durch regelbasierte System prüfen, die den Code analysieren und Verletzungen der Regeln direkt in der Entwicklungsumgebung oder als zusammenfassenden Bericht zurückliefern. Im Umfeld der Java-Entwicklung mittels Eclipse ist Checkstyle der bekannteste Vertreter solcher Werkzeuge [CS]. Es ist als OpenSource Projekt frei verfügbar.

Tests

Das Testen von Software wird heutzutage primär durch Unit Tests realisiert. Sie testen Software als kleine Einheiten und stellen durch ihre Automatisierung auch langfristig bei der Weiterentwicklung das bisher erwartete und geprüfte Verhalten sicher. Im Java-Umfeld hat sich hier JUnit als de facto Standard-Testframework etabliert. Um leichter mit Abhängigkeiten zu anderen Software-Artefakten umgehen zu können, beispielsweise wenn diese noch nicht realisiert wurden, existieren sogenannte Mockup-Frameworks. Mit einem Mock kann eine Objektinstanz inklusive ihres Verhaltens definiert und als Eingabe für einen Test verwendet werden. Dies ist auch dann möglich, wenn bisher nur eine Schnittstelle definiert wurde, aber noch keine passende Implementierung dieser existiert. Mockito ist eines der inzwischen am weitesten Entwickelten Mockup-Frameworks im Java Umfeld [Mockito].

Als Metrik dafür, wie gut die existierenden Tests für eine Software sind, wird heutzutage die sogenannte Testabdeckung des Codes (Code Coverage) verwendet. Hierbei wird während der Testdurchführung protokolliert welcher Code tatsächlich im Rahmen des Tests ausgeführt wurde. Das Ergebnis wird dann als sogenannte Testabdeckung in Form eines Berichts zusammengefasst oder der Code wird entsprechend direkt in der Entwicklungsumgebung markiert. Hierdurch können Entwickler relativ leicht sehen, welche Teile der Software noch ungenügend von Tests überprüft werden. Im Java Umfeld stehen heutzutage die Werkzeuge Emma [EM] und Cobertura [CO] als bekannteste Vertreter zur Verfügung.

Code-Analysen

Qualitätsanalysen von Code sind heutzutage ein etabliertes Mittel um Hinweise auf dessen Wartbarkeit zu erhalten. Hierbei werden zum einen Metriken in Bezug auf Komplexität, Kopplung und Vererbung erhoben [Marinescu06]. Sie geben einen Hinweis welche Teile des Codes vor allem schwer verständlich sind oder bei Änderungen ein hohes Risiko zu Nebeneffekten mit sich führen. Metriken an sich geben nur Charakteristika des Programmcodes wieder. Sie geben keine konkreten Änderungshinweise und ihr Nutzen hängt von den Schwellwerten ab, ab denen der Wert einer Metrik als kritisch angesehen wird. Google Code Pro Analytix [CP] stellt viele etablierte Metriken für Java Code innerhalb der Eclipse IDE zur Verfügung.

Die zweite Kategorie an Code-Analysen beschäftigt sich mit Problemmustern. Hierbei wird die Implementierung einer Software nach strukturellen Anomalien untersucht. Hierzu gehören beispielsweise auffällige Abhängigkeiten zwischen zwei Programmelementen. Untersucht werden solche Problemmuster im Kontext der Wartbarkeit einer Software und zur Erkennung möglicher Programmfehler, wie zum Beispiel nicht-initialisierte Datenfelder.
Im Java Umfeld existiert hier zum Beispiel das Werkzeug Findbugs [FB]. Es ist in der Lage ca. 200 Fehlermustern zu erkennen und kann sowohl direkt in der Entwicklungsumgebung als auch in einem automatisierten Build-Prozess verwendet werden.
Ebenfalls zu den Code-Analysen gehört die Erkennung von Code-Klonen. Hierbei handelt es sich um Code der kopiert wurde, um Funktionalität an einer anderen Stelle einer Software wiederzuverwenden, anstatt sie für beide Stellen gleichermaßen zugänglich zu machen. Solche Code-Kopien haben zum einen den Nachteil, dass der gleiche Code an mehreren Stellen gewartet werden muss, zum anderen wird die Code Basis unnötig vergrößert und verlangt daher auch insgesamt mehr Verwaltungs- und Wartungsaufwand.
Zur Erkennung solcher Code-Klone kann ebenfalls das Werkzeug Google Code Pro Analytix [CP] verwendet werden.

Continuous Integration

Um kontinuierlich sicherzustellen, dass eine Software in einem guten Zustand ist und dennoch den Analyse und Testaufwand für die Entwickler gering zu halten wird heutzutage mit Continuous Integration Systemen gearbeitet. Hierbei handelt es sich um eine Software, die automatisch in regelmäßigen Abständen die gemeinsame Code Verwaltung auf Änderungen prüft. Im Falle einer Änderung wird ein vorgegebener Build-Prozesse durchgeführt. Hierbei können neben dem Kompilieren der Software und dem automatischen Erzeugen eines auslieferbaren Produktes, auch alle zuvor genannten Analysen und Tests durchgeführt werden. So wird von einem Build Durchlauf neben der auslieferbaren Software auch ein Bericht erstellt, der die Qualitätsmerkmale des aktuellen Codes und die Ergebnisse der Testdurchführung wiedergibt. Bei kritischen Problemen können zusätzlich die verantwortlichen Personen automatisch informiert werden.

Im OpenSource Umfeld gehören Jenkins [JK] und Hudson [HU] zu den bekanntesten Vertretern, wobei Jenkins als Kopie des Hudson Servers entstanden ist.

6.3 Modelle

Qualitätsanalysen und –Bewertungen entstehen vor allem aus der langfristigen Erfahrung im Umgang mit Artefakten (Code, Dokumente, etc.). Da die modellgetriebene Entwicklung im Vergleich zur klassischen Programmierung noch relativ jung und damit die langfristige Erfahrung noch entsprechend geringer ist, existieren heute auch noch nicht so viele Verfahren und Werkzeuge zum Qualitätsmanagement von Modellen.

Eine standardisierte Qualitätsdefinition wie es der ISO 9126 / ISO 25000 Standard für die Softwareimplementierung liefert, existiert für Metamodelle und Modellinstanzen noch nicht. Mohagheghi et. al. [Moh09] haben jedoch 6 Qualitätskriterien für Modelle definiert, die den Stand der Technik wiederspiegeln, und die bei der modellgetriebenen Entwicklung berücksichtigt werden sollten:

- Changeability (Änderbarkeit)
- Completeness (Vollständigkeit)
- Comprehensibility (Verständlichkeit)
- Confinement (Angemessenheit)
- Consistency (Konsistenz)
- Correctness (Korrektheit)

Die verbreiteteste Art Modelle und ihre Qualität zu bewerten sind manuelle Reviews. Hier werden sowohl Metamodelle als auch Modellinstanzen von Hand geprüft und ggf. überarbeitet.

Für automatisierte Qualitätsanalysen stehen ebenfalls bereits Werkzeuge zu Verfügung. Hierbei werden vor allem allgemeine Komplexitätsmetriken, wie zum Beispiel der Grad der Kopplung zwischen zwei Modellklassen, erhoben. Genauso wie bei den Code Analysen lassen sich aus solchen Metriken auch Problemmuster erkennen.

Einen schnellen Überblick über die Thematik liefert auch der Artikel [EMFRefactor1].

EMF Refactor

Das EMF Refactor Projekt ist Teil des Eclipse Modelling Framework Projektes und befindet sich derzeit noch in einem Incubation Status ([EMFRefactor1], [EMFRefactor2]). Ziel des Projektes ist die Unterstützung von Refactorings zur Qualitätsverbesserung von Modellen. Hierzu wird zum einen die Erhebung von Metriken unterstützt, zum anderen aber auch die Transformation von Modellen zur Verbesserung.
EMF Refactor liefert bereits Standard Metriken und Refactorings speziell für UML2 Modelle mit. Darüber hinaus können diese Projekt- und Domänenspezifisch erweitert werden. Es bietet sowohl Erweiterungspunkte für eigene Metriken als auch Refactorings. Als Sprachen zur Erweiterung stehen OCL, Java, und Henshin Transformationssprache zur Verfügung.
Da die bisher mitgelieferten Metriken speziell für UML2 Modelle ausgelegt sind, ist eine Erweiterung für ein eigenes Metamodell notwendig.

Modagil Mobil Model Analyse

Im Rahmen des Modagile Mobile Projektes [ModagileMobil] wurde ein Werkzeug zur Analyse von Metamodellen und Modellinstanzen entwickelt. Es basiert ebenfalls auf dem Eclipse Modelling Framework. Aktuell ist es als Plugin für die Eclipse IDE verfügbar. Zusätzlich entsteht derzeit auch eine Integration in den Jenkins Continuous Integration Server zur kontinuierlichen Überwachung von Modellen.

Dieses Analyse-Werkzeug bietet, ebenso wie EMF-Refactor, die Möglichkeit Metriken auf Metamodellen und Modellinstanzen zu erheben. Es können Schwellwerte für die existierenden Metriken, sowie eigene Metriken eingestellt werden.

Das Werkzeug ist im Kontext von Mobilen Anwendungen entstanden. Hierdurch existieren auch erste Erfahrungen aus dieser Domäne, wenn auch noch nicht aus der langfristigen Anwendung des Werkzeugs.

7 Literaturverzeichnis

[ALMA] Architecture-level modifiability analysis (ALMA) by PerOlof Bengtsson, Nico Lassing, Jan Bosch, Hans van Vliet, The Journal of Systems and Software, 2004
[CS] http://checkstyle.sourceforge.net/
[CO] http://cobertura.sourceforge.net/
[CP] https://developers.google.com/java-dev-tools/codepro/doc/
[EM] http://emma.sourceforge.net/
[EMFRefactor1]http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2012/06/arendt_taentzer_OS_06_12_lo66.pdf
[EMFRefactor2] http://www.eclipse.org/emf-refactor/
[FB] http://findbugs.sourceforge.net/
[CM] CMMI Product Team: CMMI® *for Development - Improving processes for developing better products and services, Version 1.3*, 2010, http://www.sei.cmu.edu/reports/10tr033.pdf
[HU] http://www.hudson-ci.org/
[JDepend] http://clarkware.com/software/JDepend.html
[JK] http://jenkins-ci.org/
[Marinescu06] Michele Lanza, Radu marinescu. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, 2006
[Mockito] http://code.google.com/p/mockito/
[ModagileMobil] http://www.modagile-mobile.de
[Moh09] P. Mohagheghi, V. Dehlen, T. Neple, Definitions and Approaches to Model Quality in Model-Based Software Development – A Review of Literature, Information and Software Technology, 12/2009
[Palladio] http://www.palladio-simulator.com

