Brüder im Geiste

Konsistenz im SOA-Modellrepository bei T-Mobile

Frank Kimmlingen

*Seit 2006 wird bei T-Mobile eine große SOA-Infrastruktur (**SOA B**ack**p**lane bzw. abgekürzt SOABP) betrieben. Zur Entwurfszeit der SOA-Infrastruktur wird das SOA-Modellrepository CEISeR verwendet, welches im Wesentlichen eine konsolidierte konsistente Sicht auf die SOA-Services und die Kommunikationsbeziehungen zwischen Service-Anbieter und -Nutzer bietet. In CEISeR importieren einerseits viele unterschiedliche Personen bzw. Rollen ihren Modellanteil für die SOABP-Umgebung (Import) und andererseits wird zur Laufzeit aus CEISeR heraus die SOABP konfiguriert (Export). Daher ist eine zentrale Anforderung an CEISeR, die Konsistenz der Daten zu verschiedenen Zeitpunkten vom Anwendungsfall abhängig durch flexibel konfigurierbare Constraint-Konfigurationen überprüfen zu können.*

Das T-Mobile-SOA-Modellrepository

CEISeR (**C**entral **E**nterprise **I**ntegration **Se**rvice **R**epository) ist das SOA-Modellrepository bei T-Mobile. Ein Modellrepository im Allgemeinen dient der Beschreibung von Objektmodellen, von Objekten und deren Beziehungen zueinander, die einem bestimmten fachlichen Domänenmodell entsprechen. Bei CEISeR ist dieses Domänenmodell die SOABP-Laufzeitumgebung, die in einem MagicDraw UML-Klassendiagramm modelliert ist. Das UML-Klassendiagramm ist das CEISeR-Metamodell, aus dem mit Hilfe von openArchitectureWare [oAW], einem MDSD-Generatorframework [MDSD], das Modellrepository generiert wird. Insbesondere wird der Kern von CEISeR, der „CEISeR Core" (`core.eiCommon`), vollständig aus dem CEISeR-Metamodell erzeugt (s. Abb. 1).

Eine wichtige Eigenschaft von Modellrepositorys ist die Fähigkeit, Objekte persistieren zu können. Darüber hinausgehende Anforderungen [SeKaMä09] sind etwa die Versionierung von Objektmodellen oder die fachliche Konsistenz in dem Objektmodell. Mit der fachlichen Konsistenz in Objektmodellen setzt sich dieser Beitrag auseinander.

Um eine hierfür ausreichende Güte der Daten zu erhalten, reicht es nicht aus, Constraints zu prüfen, die sich aus dem CEISeR-Metamodell automatisch ableiten lassen, sondern es ist notwendig, teilweise komplexe fachliche Constraints zu entwickeln, die dann automatisiert ausgeführt werden. Diese fachlichen Constraints sind in der Komponente `core.eiCommon.constraints` ausgelagert (s. Abb. 1).

Für die Konfiguration und Ausführung der Constraints wurde im CEISeR-Kern ein Constraint-Framework entwickelt, welches – wie bei allen Komponenten in diesem Kern – aus einem Teil in der Plattform (`core.platform`) besteht. Die Plattform ist unabhängig von dem fachlichen Domänenmodell. Darauf aufbauend gibt es eine Menge von Constraints, die aus dem CEISeR-Metamodell generiert werden (`core.eiCommon`).

Die Anforderungen an das Constraint-Framework

Im Folgenden werden die Anforderungen an das Constraint-Framework dargestellt, deren konkrete Umsetzung in CEISeR im weiteren Verlauf beschrieben wird:

- *1. Anwendungsfallbezogene Definition von Constraint-Mengen:* Zu unterschiedlichen Zeitpunkten müssen unterschiedliche Constraints geprüft werden. Durch die Definition von Constraint-Mengen wird die Menge der zu prüfenden Constraints pro Entität festgelegt.
- *2. Anwendungsfallbezogene Definition von Metamodellfragmenten:* Metamodellfragmente werden auf der Basis des CEISeR-Metamodells definiert und entsprechen Ausschnitten des CEISeR-Metamodells. Durch die Definition von Metamodellfragmenten wird die Menge der zu prüfenden Entitäts-Objekte gesteuert. Die Menge der Entitäts-Objekte, die zur Laufzeit geprüft werden, wird im Folgenden auch Modellfragment genannt und ergibt sich durch die Anwendung eines Metamodellfragments auf die zu prüfenden Entitäts-Objekte.
- *3. Unterscheidung der Constraint-Verletzungen nach „error" und „warning":* Ein „error" soll dazu führen, dass der jeweilige Anwendungsfall abgebrochen wird. Demgegenüber darf eine „warning" den Benutzer lediglich darauf aufmerksam machen, dass in dem Modellfragment, welches überprüft wurde, etwas noch nicht vollständig ist.
- *4. Prüfung einzelner Entitäts-Objekte:* Die Prüfung einzelner Entitäts-Objekte soll möglich sein.
- *5. Prüfung einer Menge von Entitäts-Objekten:* Im Kontext einer einzelnen Prüfung soll eine Menge von Entitäts-Objekten prüfbar sein.
- *6. Implizite oder explizite Constraint-Ausführung:* Die Prüfung von Constraints soll explizit durch den Nutzer oder implizit durch die Anwendung erfolgen können.
- *7. Leichte Wartbarkeit und flexible Erweiterbarkeit:* Da häufig Änderungen am CEISeR-Metamodell vorgenommen werden, ist es wichtig, dass die Pflege der Constraint-Konfigurationen einfach und flexibel ist.

Constraints und Constraint-Konfigurationen

Abbildung 2 stellt das Konzept des Constraint-Frameworks dar. Die zentrale Klasse ist `ConstraintConfiguration`, welche dafür zuständig ist, Constraint-Konfigurationsdateien zu lesen.

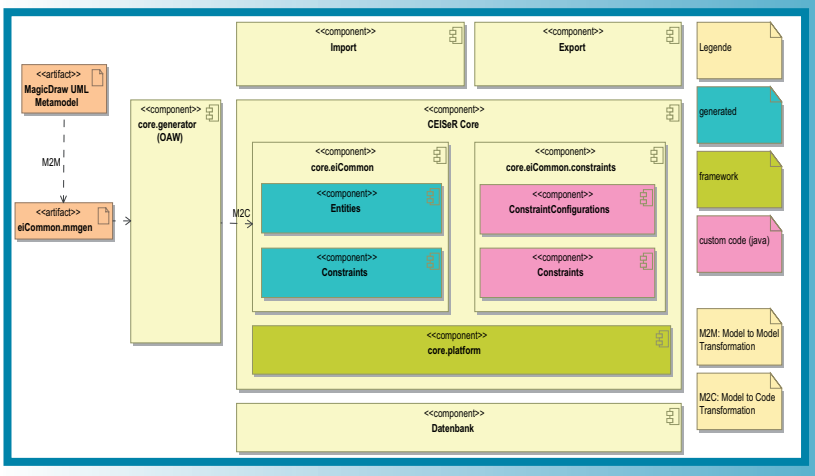

Abb. 1: Architektur von CEISeR

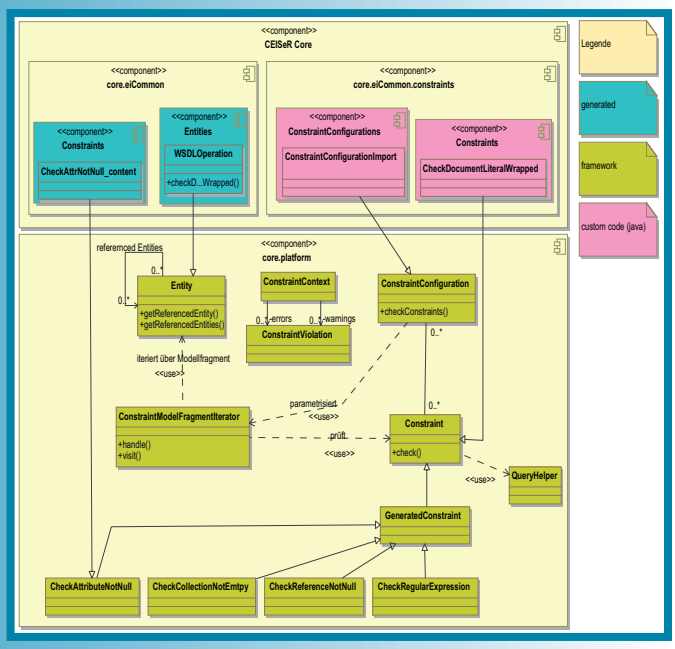

Abb. 2: Konzept des Constraint-Frameworks

Die Constraint-Konfigurationsdateien beschreiben basierend auf dem CEISeR-Metamodell:

- Constraint-Mengen für Entitäten (s. o. Anforderung 1),
- Metamodellfragmente durch Navigationsregeln basierend auf Entitäten und deren Referenzen (Anforderung 2),
- Unterscheidung der Constraints nach „error“ und „warning“ (Anforderung 3).

Das folgende Beispiel stellt exemplarisch einen Ausschnitt einer solchen Datei dar, die dem Metamodellfragment aus Abbildung 3 entspricht:

```
# Metamodellfragment
+ nav WSDLMessageReference.message
+ nav XMLSchemaComponent.usedComps

# Constraint-Mengen
errors:
+ chk XMLSchemaComponent attrNotNull_content
+ chk WSDLOperation documentLiteralWrapped

warnings:
+ckk ….
```

Die Constraint-Mengen werden über „+ chk“-Regeln konfiguriert. Die Regel

```
+ chk WSDLOperation documentLiteralWrapped
```

aus obigem Beispiel bedeutet beispielsweise, dass für die Entität **`WSDLOperation`** der Constraint mit dem Namen **`documentLiteralWrapped`** aufgerufen werden soll.

Über die Schlüsselwörter **`errors:`** bzw. **`warnings:`** werden die nachfolgenden Regeln als „error“ bzw. „warning“ interpretiert (**`documentLiteralWrapped`** ist z. B. ein „error“). Über die Navigationsregeln werden die Metamodellfragmente definiert. Die Regel

```
+ nav WSDLMessageReference.message
```

bedeutet beispielsweise, dass von der Entität **`WSDLMessageReference`** die Assoziation mit dem Namen **`message`** zu dem Metamodellfragment gehört.

Bei der Sprache der Constraint-Konfigurationsdateien handelt es sich um eine textuelle domänenspezifische Sprache für die Definition von Metamodellfragmenten und den dazugehörenden Constraint-Mengen [DSL]. Dieser Ansatz wurde gewählt, um die Constraint-Konfigurationen möglichst einfach, ohne notwendiges Generieren bzw. Kompilieren, ändern zu können. Für jeden in der Domäne relevanten Anwendungsfall gibt es eine Spezialisierung dieser Klasse (z. B. **`ConstrainConfigurationImport`** für den Anwendungsfall „Import“), die eine entsprechende Constraint-Konfigurationsdatei einliest. Die Klasse **`ConstrainConfiguration`** parametrisiert die Klasse **`ConstraintModelFragmentIterator`**, welche zusammen mit den Klassen **`Entity`** und **`Constraint`** das Visitor-Muster [Visitor] implementiert. Die ersten drei Anforderungen sind wie oben beschrieben gemäß Abbildung 2 realisiert.

Generierte versus implementierte Constraints

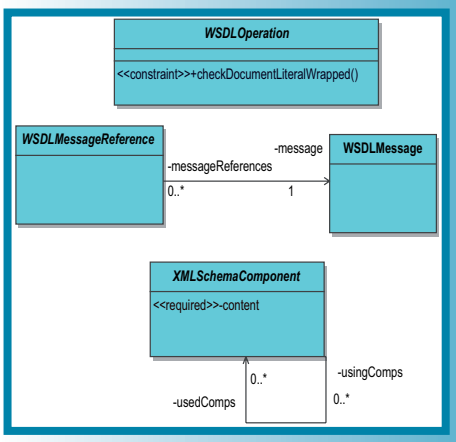

Abb. 3: Ausschnitt eines Metamodellfragments

In Abbildung 2 ist der Unterschied zwischen generierten und implementierten Constraints dargestellt. In einem modellgetriebenen Entwicklungsprojekt [MDSD], wie CEISeR, ist ein Ziel, möglichst viele Informationen im Metamodell unterzubringen. Für die Realisierung von komplexen Constraints bietet sich die Object Constraint Language [OCL] an. CEISeR nutzt lediglich zur Generierungszeit (**`core.generator`**) das Eclipse Modeling Framework [EMF]. Die Laufzeitkomponente (**`core.platform`**) basiert auf einer proprietären Hibernate[HIB]-Persistenzschicht. Daher haben wir uns aus Aufwandsgründen dazu entschieden, komplexe Constraints nicht im Metamodell zu beschreiben, sondern diese lediglich durch den Stereotyp **`<<constraint>>`** auszuzeichnen und in Java zu implementieren. Die Implementierung der Constraints basiert hierbei im Wesentlichen auf den beiden CEISeR-Kern-Klassen **`Entity`** und **`QueryHelper`**. Einfache Constraints, wie z. B. **`AttributeNotNull`** (Stereotyp **`<<required>>`**) und **`ReferenceNotNull`**, werden aus dem Metamodell generiert. Ein Beispiel eines komplexen implementierten Constraints ist **`CheckDocumentLiteralWrapped`** (s. Abb. 2 und 3).

Anwendung einer ConstraintConfiguration

Bei der Anwendung einer „ConstraintConfiguration“ muss der Nutzer sich einen sogenannten „ConstraintContext“ erzeugen. Er wird einerseits im oben erwähnten Visitor-Muster des **`ConstraintModelFragmentIterator`** für das Aufsammeln der bereits besuchten Entitäten genutzt. Der Nutzer des Frameworks andererseits kann vom „ConstraintContext“ „errors“ und „warnings“ erfragen, die bei der Ausführung der Constraints aufgetreten sind. Details des Ablaufs sind dem Sequenzdiagramm in Abbildung 4 zu entnehmen.

Durch das Konzept des „ConstraintContext“ werden die 4. und 5. Anforderung erfüllt. Die 6. Anforderung wird er-

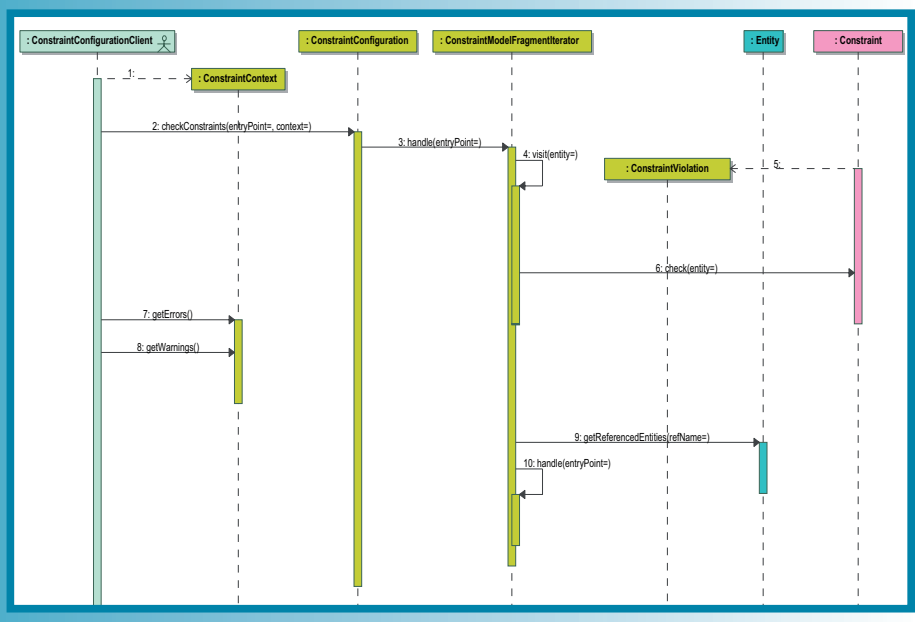

Abb. 4: Nutzung einer ConstraintConfiguration

füllt, indem der Nutzer einerseits explizit die Steuerung der Constraint-Prüfung, wie gerade beschrieben, vornehmen kann. Die implizite Constraint-Prüfung wird direkt vom CEISeR-Kern übernommen, indem die `core.platform`-Klassen die Steuerung der Constraint-Prüfung mandatorisch in ihrer `commit`-Methode ausführen. Sobald ein „error" auftritt, wird die gesamte Datenbanktransaktion zurückgerollt.

Management von Constraint-Konfigurationen

Die 7. Anforderung erfordert das einfache Management der Constraint-Konfigurationen. Dies wird erreicht, indem alle Constraint-Konfigurationen Spezialisierungen von `ConstraintConfigurationBase` sind (s. Abb. 5).

`ConstraintConfigurationBase` enthält dabei sämtliche generierten Constraints und Navigationsregeln, die sich durch Kompositionsbeziehungen im CEISeR-Metamodell ergeben. Die dazugehörige Constraint-Konfigurationsdatei wird generiert. Dann muss sie explizit in einem vorgegebenen Verzeichnis abgelegt und unter Versionskontrolle gestellt werden. Hierdurch wird die Nachvollziehbarkeit und explizite Steuerung der Constraint-Konfigurationen in einem MDSD-Ansatz bewahrt und gleichzeitig die manuelle Erstellung großer Dateien vermieden.

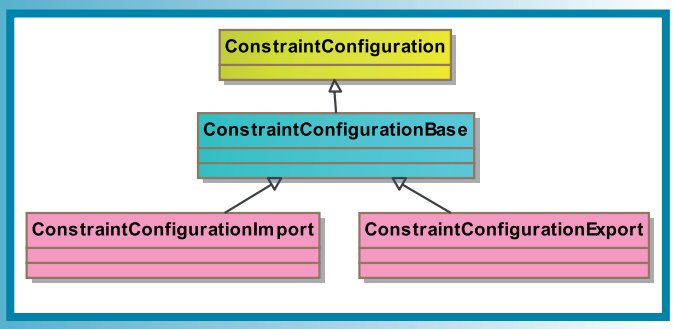

Abb. 5: Vererbung von Constraint-Konfigurationen

Grafische Darstellung von Constraint-Konfigurationsgraphen

Bei Constraint-Konfigurationen handelt es sich um komplexe Graphstrukturen, weshalb die grafische Darstellung zu „Debug"-Zwecken sehr nützlich ist. Die Realisierung ist mit der Hilfe von GraphML [GRAPHML] und dem Toolkit prefuse [PREFUSE] auf der Basis der zur Laufzeit vorliegenden Information der `ConstraintConfiguration`-Instanz (Metamodellfragmente und Constraint-Mengen) erfolgt. In Abbildung 6 ist der Graph der Constraint-Konfiguration `ConstraintConfigurationImport` für die Entität `Environment` dargestellt.

Fazit

In CEISeR wird eine verteilte Modellierung durch mehrere Personen bzw. Rollen betrieben, aus deren Ergebnis zur Laufzeit die Konfiguration der SOABP erfolgt. Daher bestehen Anforderungen bezüglich der Konsistenz des SOA-Modellrepositorys bei T-Mobile, die über das hinausgehen, was bei mir bekannten SOA-Repository-Produkten am Markt zu finden ist. Die hier beschriebene Umsetzung der Anforderungen integriert sich zum einen sehr gut in die vorhandene MDSD-basierte Entwicklung von CEISeR. Ein pragmatischer Ansatz einer textuellen DSL wird zum anderen für die Definition von Anwendungsfall-abhängigen Constraint-Konfigurationen verfolgt, die sich einfach und flexibel anpassen lassen.

Links und Literatur

[DSL] Domänenspezifische Sprache,
`http://de.wikipedia.org/wiki/Domänenspezifische_Sprache`

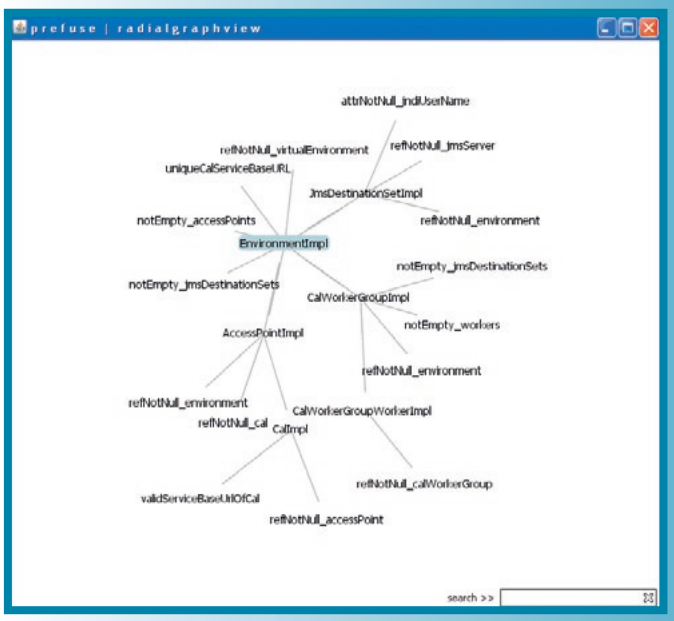

Abb. 6: Constraint-Konfiguration des SOABP-Environment beim Import

[EMF] Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/
[HIB] Hibernate, http://www.hibernate.org
[GRAPHML] GraphML, http://graphml.graphdrawing.org/
[MDSD] Model-Driven Software Development, http://de.wikipedia.org/wiki/Modellgetriebene_Softwareentwicklung
[oAW] openArchitectureWare, http://www.openarchitectureware.org/
[OCL] Object Constraint Language, http://de.wikipedia.org/wiki/Object_Constraint_Language
[PREFUSE] Prefuse, http://prefuse.org/
[SeKa08] C. Sensler, A. Karalus, SOA@T-Mobile – Vollautomatische Service Provisionierung auf dem ESB – Teil 1-3, in: Java Magazin, 10.2008 – 12.2008, http://www.sensler.de/public.html
[SeKaMä09] C. Sensler, A. Karalus, M. Märtens, Modellrepository @ T-Mobile, in: JavaSPEKTRUM, 1/2009
[Visitor] Visitor-Muster, http://en.wikipedia.org/wiki/Visitor_pattern

Frank Kimmlingen arbeitet seit über 10 Jahren bei der T-Mobile Deutschland als Softwareentwickler, Architekt und Projektleiter im Framework- und Enterprise-Integration-Umfeld. Seit etwa 2003 bringt er die modellgetriebene Softwareentwicklung in verschiedenen SOA-Umgebungen der T-Mobile voran. Nach seinem Abschluss 1995 an der Universität Bonn arbeitete er bei der IDS Prof. Scheer GmbH in Saarbrücken und bei der SAP AG in Walldorf an der Umsetzung modellbasierter Softwareentwicklungsansätze. E-Mail: Frank.Kimmlingen@t-mobile.de.