TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Automatische Codegenerierung für Proxies in MOST-System

18. Mai 2004

Themensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Bearbeiter:	Jie Tang
Betreuer:	Stefan Wagner (TU Muenchen)
	Michael Ehrmann (BMW CarIT)

Zusammenfassung

Im Rahmen der MMI (Man Machine Interface) Entwicklung bei der Firma BMW Car IT wird eine Proxy- Softwareschicht, die eine Abbildung auf konkrete MOST (Media Oriented System Transport) Funktionsblöcke darstellt, benötigt, um es zu ermöglichen, die Steuergeräte im MOST-Verbund programmatisch anzusteuern. Aufgrund des großen Umfangs im aktuellen MOST Bussystem ist es für diese Schicht möglichst automatisch Sourcecode zu generieren, um einerseits leichte Programmierfehler zu minimieren und andererseits Redundanz zu vermeiden.

In dieser Ausarbeitung wird rund um das Thema Codegenerierung diskutiert. Nach einer Einführung ins Themengebiet folgt ein Kapitel, das die grundlegenden Begriffe wie Model und Template systematisch erläutert. Basierend auf dem theoretischen Ansatz werden bestehende Konzepte jeweils mit Hilfe konkreter Implementierungen vorgestellt. Im Anschluß daran wird die praktische Arbeit dieses SEP beschrieben. Dabei handelt es sich um die Vorstellung der Rahmenbedingungen, die mit der Durchführung des SEP eng zusammenhängen und damit die Gestaltung des Endprodukts in gewisser Weise mitprägen. Des weiteren will man den eingesetzten Code-Generator erfahren, dessen Aufbau und Funktionsweise ebenfalls detailliert erläutert wird. Abschließend gibt es noch eine Zusammenfassung und einen Ausblick, der den aktuellen Stand der Code-Generierungstechnik beschreibt und deren Entwicklungstrend aufweist.

Inhaltsverzeichnis

1 Einführung **4**

2 Grundlegende Begriffe **4**
2.1 Typische Einsatzgebiete 5
2.1.1 Compilerbau 5
2.1.2 Framework 5
2.2 Das zugrunde liegende Schema 5

3 Bestehende Konzepte und deren Implementierung **6**

4 Implementierung **8**
4.1 Rahmenbedingungen der Projektdurchführung 8
4.1.1 MOST - Media Oriented Systems Transport 8
4.1.2 Komponenten von MOST 9
4.1.3 Framework J2MOST 9
4.1.4 MMI System 10
4.2 Codegenerator im Einsatz 11
4.3 Struktur der Treiber bzw. Templates 12
4.4 Vorstellung des Endproduktes 13

5 Zusammenfassung und Ausblick **14**

1 Einführung

In der Geschichte der Softwareentwicklung hat man bereits mehrfach davon geträumt, den Programmcode automatisch durch Tools generieren zu lassen. Mit Hilfe eines Compilers ist nun die Umsetzung einer höheren Programmiersprache in eine niederigere eine Sache von einer Befehlsausführung innerhalb weniger Sekunden. Das hat zum einen erheblichen menschlichen und sachlichen Aufwand erspart und zum anderen zu einer Minimierung der Fehler geführt und damit auch in bedeutendem Maße zur Produktivitätssteigerung beigetragen.

Einen solchen Automatismus hat man sich natürlich immer gewünscht. Aber bei näherem Hinsehen ist leicht festzustellen, dass sich dies auf einer Ebene befindet, wo die Beziehung zwischen Eingabe und Ausgabe in Hinsicht auf ihre Struktur auf einer stabilen und wohldefinierten Basis verankert ist. Aber genau das ist in der Informatik im allgemeinen nicht der Fall: Informatik lässt sich dadurch kennzeichnen, dass sie sich gerade auf einem raschen Kurs der Entwicklung befindet und in vielen Aspekten sich kein Bezug auf bewährte Standards oder Normen nehmen lässt.

Man stellt sich dann die Frage, wo die Grenzen für die automatische Codegenerierung sind? Wenn man sie wohl nicht im allgemeinen Fall einsetzen kann, lässt sich trotzdem eine methodische Aussage dafür treffen, in welchen Fällen sie den Softwareentwicklern am günstigsten gegenübersteht und damit es praktikabel ist, sie in den Entwicklungsablauf zu integrieren?

Im Rahmen eines Codegenerierungseinsatzes versucht der vorliegende Praktikumsbericht einerseits eigene Erfahrungen, die in der Praxis gesammelt worden sind, aufzuzeigen, und andererseits den Entwicklungszustand derzeitiger Codegenerierungstechnik systematisch so vorzustellen, dass daraus für jedes typischen Einsatzgebiet ein geeingenter Codegenerator hervorgeht.

2 Grundlegende Begriffe

Was steht nun hinter dem Begriff der Codegenerierung? Im Allgemeinen läßt sich die Frage ziemlich einfach beantworten: Codegenerierung ist eine Tätigkeit, die sich darauf bezieht, eine Art von Code zu schreiben, die wiederum neue Codes generieren(meistens automatisch) kann, kurz gefasst, man schreibt code, der code schreibt.

Hier ist der Begriff des Codes aber auch sehr allgemein gefasst. Wir können zum einen von direkt ausführbaren Codes ausgehen, die unmittelbar eingesetzt werden können, zum anderen könnten die Codes auch als Zwischenergebnis eine Art

vom Modell darstellen, die wiederum als Input zur Codegenerierung dient. Diese Gestaltung hat eigentlich damit zu tun, dass das Softwareengineering sich immer darauf konzentriert, die Ebene der Abstraktionen zu erhöhen. Damit verbunden wird der Einsatz der Codegenerierung in zwei Gebiete untergliedert.

2.1 Typische Einsatzgebiete

2.1.1 Compilerbau

Als ein klassisches Beispiel der Codegenerierung ist wohl der Compilerbau zu nennen. Dabei stellt die Codegenerierung eine wichtige Komponente dar, worin unter anderem sowohl Zwischencode als auch Maschinencode erzeugt werden, die alle wiederum aus einer Quellsprache, z.B Programmiersprachen wie C++, Pascal, hervorgehen. Im Grunde genommen handelt es sich dabei als Input für die Codegenerierung um einen aus der Analysephase stammenden Ableitungsbaum. Die Umsetzung in den Zwischencode erfolgt dann natürlich durch ein festgelegtes Schema, mit dessen Hilfe z.B eine Zuweisung in einen MOVE Befehl umgewandelt wird.

2.1.2 Framework

Das *Framework* ist insofern ein interessantes Objekt für die Codegenerierung, weil es meistens viele vordefinierte Artefakten den Benutzern vorschreibt. In [1] wird Framework wie folgt definiert:

Ein Framework ist ein partiell vollständiges Software-(Sub-) System, das nur noch instanziiert werden muß. Es definiert die Architektur für eine Familie von (Sub-) System und stellt die Grundbausteine für ihre Erzeugung dar. Es reglementiert die Punkte, an denen Anpassungen für spezielle Zwecke vorgenommen werden müssen.

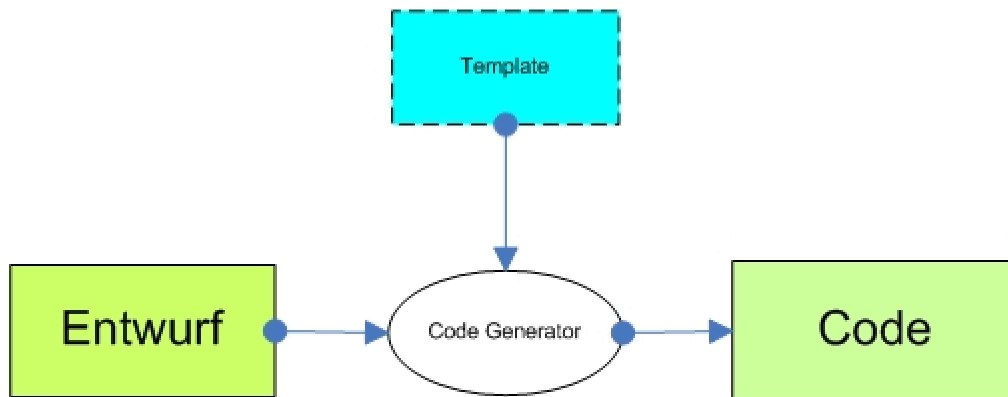
J2EE ist ein gutes Beispiel, um zu erklären, warum hier die Codegenerierung notwendig ist. Für eine Persistenzverwaltung einer Datenbanktabelle sind insgesamt 3 Klassen und 2 Schnittstellen niederzuschreiben, wenn EJB zum Einsatz kommt. Fast alle dieser Klassen haben vordefinierte Strukturen, weshalb eine automatische Generierung durch Tools sicher solche Aspekte ausnutzen kann, und damit die Vorteile in Richtung Fehlerminimierung und schnelle Änderungsanpassung einbringen kann. Im Übrigen fällt auch das SEP unter dieser Kategorie, worüber in späteren Kapiteln detaillierter eingegangeng wird.

2.2 Das zugrunde liegende Schema

Zum Abschluß dieses Kapitels will ich noch das Schema, das allen Codegenerierungstechniken zugrunde liegt, einführen. Wie vorhin bereits mehrfach erwähnt

wurde, um seine Arbeit abzuleisten, müssen Informationen als Input vorliegen, die strukturiert sind. Der Codegenerator liest dann diese Datei ein und bereitet diese je nach Bedarf auf, z.B wäre hier eine Umwandlung ins Objekt denkbar. Um die Informationen auszunutzen braucht der Codegenerator noch ein sogenanntes Template (deu. Vorlage), das ein Skelett für den zu erzeugenden Zielcode darstellt. Die Aufgabe des Codegenerators besteht dann darin, Inputinformationen auf geeigneter Weise in das Template einfließen zu lassen, damit am Ende der Zielcode als Produkt entsteht.

Abbild 1 zeigt das Schema des state-of-the-art Codegenerators:



3 Bestehende Konzepte und deren Implementierung

In diesem Kapitel wird der Stand der Technik zum Thema Codegenerierung vorgestellt. Des weiteren werden auch die jeweiligen Implementierungen in Hinsicht auf deren Vorteile und Nachteile erläutert.

Nach der Verwendungsart kann sich die Codegenerierung in zwei Kategorien untergliedern:

- Ein *passiver Code Generator* wird typischerweise nur einmal ausgeführt. Von dem Zeitpunkt an wird das Ergebnis freistehend, d.h. Es trennt sich vom Codegenerator, und wird im Laufe der Entwicklung von den Programmierern weiter gepflegt. Die Wizards und einige CASE Tools sind typische Beispiele dafür. Ein passiver Code Generator erspart das Eintippen. Sie sind grundsätzlich parametrisierte Templates. Sobald das Ergebnis produziert wird, wandelt es sich zu Quelldateien in Projekten, die weiter editiert, kompiliert, und unter Source Control gesetzt werden genau wie andere Dateien. Ihre Herkunft wird sozusagen vergessen.

- Ein *aktiver Code Generator* wird in den Einsatz gebracht wenn sein Ergebnis jedes Mal benötigt wird. Das Ergebnis ist so wie ein Handzettel weil es immer vom Code Generator reproduziert werden kann. Im Gegensatz zu seinem passiven Verwandten bringt der aktive Code-Generator mehr Nutzen als reine Eintippensparnisse ein. Wenn man das Prinzip DRY[1] streng befolgen will, ist der Einsatz von aktiven Code Generatoren sogar ein Muss. Ein Bespiel aus dem Datenbankbereich wird einleuchten: Hier haben wir es normalerweise mit zwei Gebieten zu tun: Die Datenbank auf einer Seite und die Programmiersprache auf der anderen Seite, die eingesetzt wird, um auf die Datenbank zuzugreifen. Mit einem Datenbankschema in der Hand braucht man nur noch eine Datenbankzugriffsschicht zu definieren. Man könnte natürlich diese Schicht von Hand codieren, was aber das DRY Prinzip rücksichtslos verletzen würde. Im Klartext ausgedrückt wird das Wissen über das Schema an zwei Orten gespeichert, was natürlich zur Folge hat, wenn sich das Schema ändert, dass auch der Zugriffscode von Hand modifiziert werden muss. Wenn dies aber in Vergessenheit geraten würde, dann würde man durch unvorhersehbare Laufzeitfehler bestraft werden . Wenn hier aber der aktive Code Generator in die Entwicklung integriert wird, heißt es immer, die Schemaänderung wird unverzüglich die Modifizierung des Zugriffcodes veranlassen. Somit bleibt man immer auf der sicheren Seite.

Nach der Vorgehensweise des Code Generators

- *Code-Driven Ansatz* erhält seinen Namen, dadurch dass der Code Generator den Sourcecode als seine Inputsource (Man erinnert sich noch an das aufgeführte Schema) benutzt. Ein Beispiel in dieser Kategorie stellt XDoclet[3] dar: Die Metadaten, die im Kommentarbereich einer normalen Sourcedatei attributiert werden, werden vom Code Generator eingelesen, und je nach deren Bedeutung die Targetdatei erzeugt. Ein gutes Beispiel ist wieder mal EJB: Aus einer attributierten Entity Bean Klasse generiert XDoclet alle Klassen rundherum, wie Home, Remote Interface, und Datenzugriffklasse, etc. Sein Vorteil liegt darin, dass der Codegenerierungsprozess nun eng mit in den alltäglichen Entwicklungsprozess integriert wird. Sein einziger Nachteil läßt sich so argumentieren, dass sich der Entwurf in den Implementierungscode vermischt, was aber zur Folge hat, dass die Software stark am Abstraktionsgrad verliert.

- *Model-Driven Ansatz* Die Alternative dazu weist der Model-Driven Ansatz aus, der den Code üblicherweise aus einem abstrakten Entwurfsmodell zusammenstellt. Viele gängigen Codegeneratoren folgen diesem Weg, wie z.B XSLT und Velocity. Das Input stellt normalerweise eine XML Datei, oder

[1] Don't Repeat Yourself

UML Diagram dar, die oft strukturierte Informationen enthalten. Was aber das Template angeht, sind es hier ziemlich variantenreich: Jeder Codegenerator definiert eigenständige Templateformat, was aber von der Idee hier nicht weit auseinander liegen. Seine Stärke läßt sich durch seine Flexibilität zeigen: Eine Umstellung von einer Programmiersprache in eine andere ist nun die Arbeit der Templateverfassung, aber der Entwurf bleibt dabei als eigentständiges Artefakt unberührt.

- *MDA (Model Driven Architecture)* Model-Driven Architecture ist eine Initiative von Object Management Group(OMG) für Codegenerierung. Die Grundidee dabei ist zu versuchen, die hoch abstrakten Modelle aus der frühren Phase der Softwareentwicklung schrittweise durch systematisches Mapping in konkrete Modelle und sogar einsetzbare Code zu verwandeln. Genauer gesagt, man beginnt mit einem so genannten PIM(Platform Independet Model) Modell, das wie der Name schon andeutet, nur die Eigenschaften und Verhalten spezifiziert aber keinerlei Informationen über drunter stehende Platform oder Technologie angibt. Danach läßt sich das Modell in eine XML Datei(XMI interchanged Format) exportieren läßt zur weiteren Verarbeitung. Im nächsten Schritt schraubt man nun den Abstraktionsgrad herunter, und transformiert das PIM in PSM(Platform Specific Model) mit Hilfe vom so genannten Mapping, das in spezieller Sprache verfasst wird und oft die in PIM spezifizierten Marks ausnutzt.

4 Implementierung

In diesem Kapitel handelt es sich um die Vorstellung der praktischen Arbeit. Am Anfang die Rahmenbedingungen, unter denen das SEP beschrieben wird. Danach folgt eine Vorstellung des Codegenerators, der im SEP zum Einsatz kommt. Anschließend erfahren Sie die Artefakte, genauer gesagt, die so genannten Treiber und Templates, die während des SEPs entstanden sind. Abschließend kommt noch die Vorstellung des Endproduktes, das auch intern im Projekt dem MMI System die Möglichkeit für den MOST Zugang eröffnet.

4.1 Rahmenbedingungen der Projektdurchführung

4.1.1 MOST - Media Oriented Systems Transport

In modernen Fahrzeugen insbesondere im Premium Segment werden immer stärker Komponenten aus dem Bereich Kommunikation, Information und Multimedia miteinander vernetzt. Bei diesen Anwendungen werden zusätzlich zu den Steuerdaten Multimediadaten übertragen und dafür muß entsprechende Bandbreite auf dem verwendeten Bus zur Verfügung stehen.

Zu diesem Zweck wird das Multimedia Bussystem MOST (Media Oriented Systems Transport, [2]) eingesetzt. Dieser Bus stellt eine Bandbreite von bis zu 24.8 Mbit/s zur Verfügung. Die Busteilnehmer werden über ein optisches Übertragungsmedium zu einem Ring verbunden.

4.1.2 Komponenten von MOST

MOST-Devices

Ein MOST-Device ist eine physikalische Einheit, die mit mindestens einem MOST-Transceiver ausgestattet ist. Auf der Applikationsebene enthält ein MOST-Device mehrere Komponenten, die Funktionsblöcke genannt werden, z.B CDPlayer oder Telefon.

MOST-Funktionen

Ein Funktionsblock ist eine Sammlung verschiedener MOST-Funktionen. Dabei gibt es zwei Arten von MOST-Funktionen:

- **Methoden**: Methoden sind Funktionen, die gestartet werden können und nach einer bestimmten Zeit das Ergebnis zurückliefern.
- **Properties**: Properties sind Funktionen, über die bestimmte Eigenschaften des MOST-Device verändert oder ausgelesen werden können.

Funktionen werden folgendmaßen addressiert:
DeviceID.FBlockID.InstID.FktID.OpType.(Data)

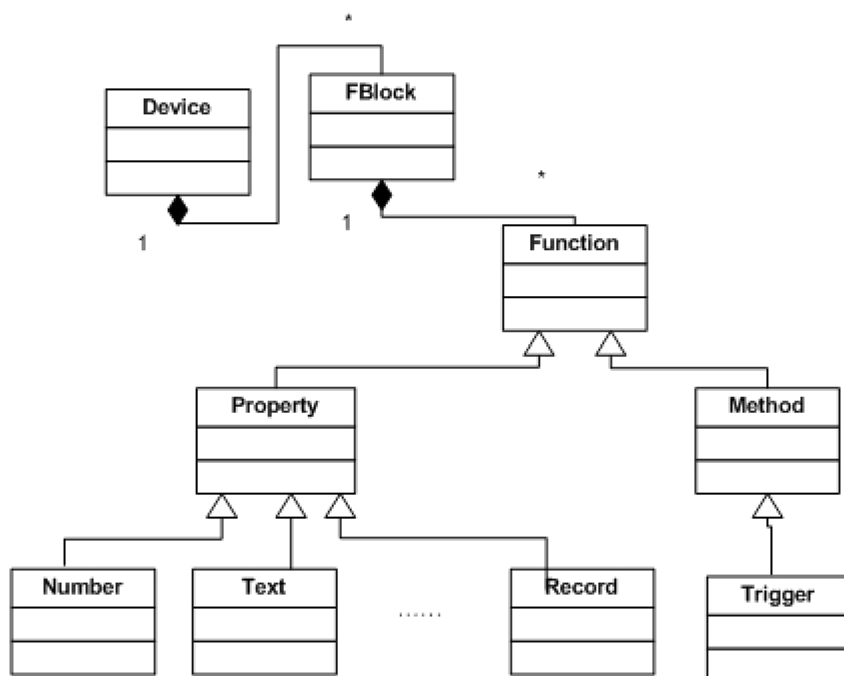
Dabei kennzeichnen die *DeviceID* und *FBlockID* den jeweiligen Device- und Funktionsblock, wo die Funktion enthalten ist. Das *InstID* wird benutzt, um mehrere identische Funktionsblöcke innerhalb eines MOST-Device zu unterscheiden. *FktID* ist der Identifikator der MOST-Funktionen und *OpType* der Identifikator des Operationstyps. *Data* beinhaltet in der Regel die Parameter des Funktionsaufrufs.

4.1.3 Framework J2MOST

Aufgrund des großen Umfangs (Im aktuellen E60 sind etwa 14 MOST-Devices, 63 MOST-Funktionsblöcke und über 800 MOST-Funktionen vorhanden) stellt dies das ideale Ziel für Codegenerierung dar. Dadurch will man vor allen Dingen die manuellen Fehler, die sich während üblicher Programmiertätigkeit einschleichen, dramatisch reduzieren. Auf einer anderen Seite würde auch der Wartungsaufwand viel zu hoch scheinen aufgrund geänderter Fahrzeugplatform, was logischerweise eine unvermeidbare Anforderung vor allem vor dem Hintergrund der Vorentwicklungstätigkeit bei BMW CarIT darstellt, wenn Codegenerierung nicht praktiziert

würde.

Um die Codegenerierung in Gang zu bringen, wurde zunächst ein Framework namens J2MOST geschaffen, das einerseits die Strukturen von MOST, die oben aufgeführt wurde, widerspiegelt, und andererseits die Laufsteuerungen und die Protokolle beinhaltet. Dabei gibt es Hotspots, die in Form von abstrakten Klassen vorliegen und von Benutzern des Frameworks, in diesem Fall den erzeugten Code, abgeleitet werden müssen.
Abbild 2 zeigt die Struktur von J2MOST.



4.1.4 MMI System

Das MMI (Man Machine Interface) System eines Fahrzeugs stellt eine zentrale Interaktionsschnittstelle dar zwischen dem Benutzer und den softwarebasierten Funktionen des Fahrzeugs. BMW hat hierzu das iDrive als MMI entwickelt, das bei aktuellen 7er und neuen 5er zum Einsatz kommt. Das MMI System stützt sich in großen Teilen auf J2MOST, weil viele ihrer Funktionalitäten direkt von MOST oder über einen CAN-MOST Gateway damit indirekt unterstützt werden.

4.2 Codegenerator im Einsatz

Der im SEP eingesetzte Codegenerator basiert auf der Programmiersprache Java, d.h. Alles, was zur Codegenerierung erzeugt wird, wird in Java-Syntax geschrieben. Deshalb werden die ganzen Artefakte zur Laufzeit der Generierung interpretiert und ausgeführt. Um dies zu bewerkstelligen, verwendet der Codegenerator intern eine Skriptsprache, die sich BeanShell[4] nennt, das wiederum mit Java kompatibel ist und zugleich auch typische Skriptcharakter wie Typfreiheit vorweist.

Um seine Arbeit zu erledigen, braucht der Codegenerator grundsätzlich zwei Dinge: Das eine ist Treiber, der als Schnittstelle zwischen dem Input und dem Generator dient und das Input in den für Generator verwertbaren Informationsträger, z.B HashMap oder XML Element, verwandelt. Das andere ist Template, das als Skelett für das Generierungsprodukt eingesetzt wird und im Grunde genommen sich aus drei Bereichen zusammensetzt: Textbereich, wo der Text unverändert in das Endprodukt übertragen wird, und Skriptbereich, wo der Codegenerator seine Arbeit verrichtet und zwar aus dem vorhin erwähnten Informationsträger die für das Template interessanten Informationen herausnimmt. Der dritte ist der Ersetzungsbereich, wo der Platzhalter im Template durch gewonnene Informationen im Skriptbereich ersetzt wird.

Abbild 3 veranschaulicht den geschilderten Generierungsprozess.

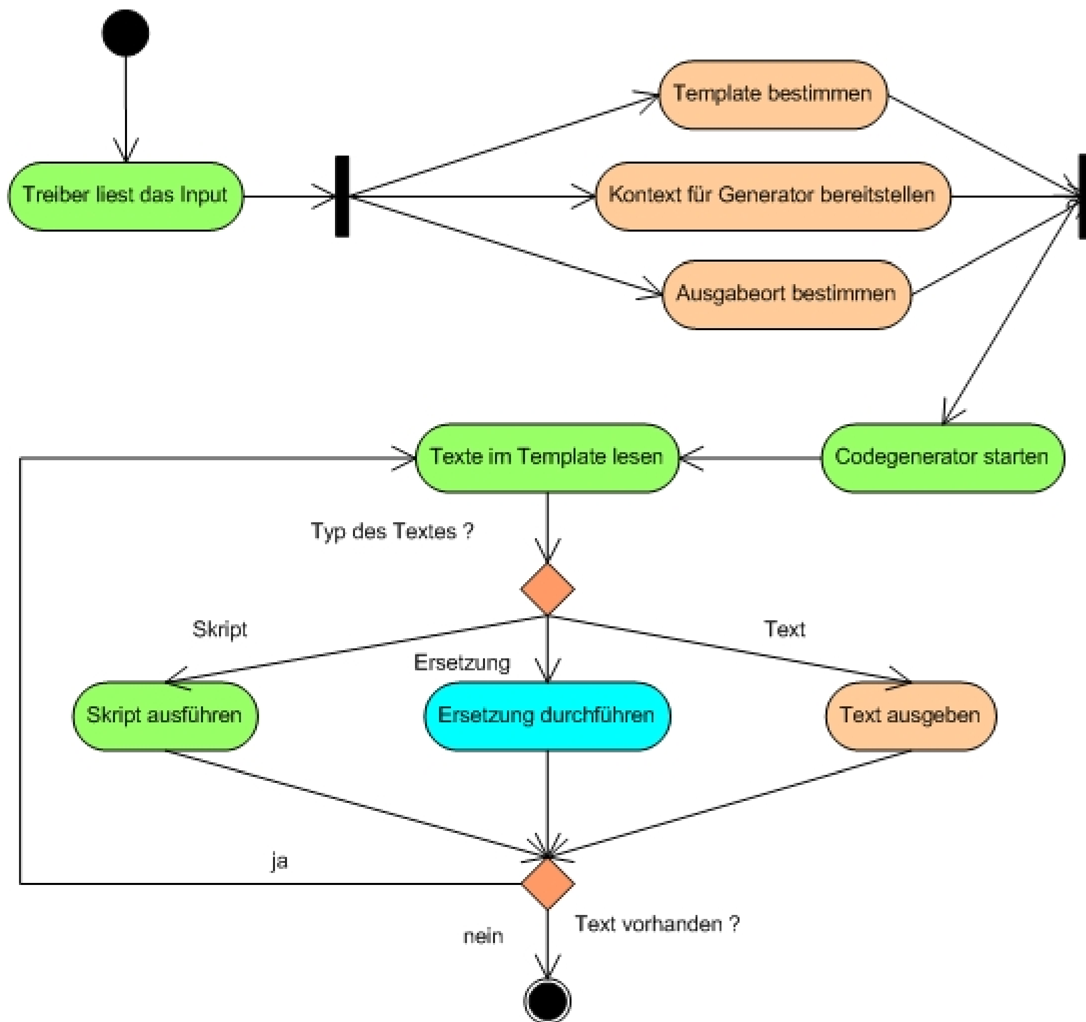


Abbildung 1: Generierungsprozess

4.3 Struktur der Treiber bzw. Templates

Basierend auf der Beschreibung im letzten Kapitel, wird in diesem Kapitel vor allem über die Struktur der Treiber und Templates erzählt, die einen großen Teil dieses SEP ausmachen. Weil die ganzen generierten Artefakte zum Schluß mit dem Framework J2Most arbeiten müssen, ist es selbstverständlich, dass die Struktur der Treiber und Templates ganz eng an der Struktur von J2Most gekoppelt ist. Wie es schon aus der Mosteinführung und Vorstellung von J2Most zu erfahren ist, existiert in der MOST Welt eine hierarchische Struktur mit der Modellierung von Device, Funktionblock und Funktion, welche auch hier wiederum zu finden ist. Für Device und Functionblock gibt es jeweils eigene Treiber und Templates. Der Grund für diese Trennung liegt an der verwendeten XML

Datei, die eine umgekehrte Relation zwischen Functionblock und Device besitzt, d.h. das Device Element ist dem Funktionblock Element untergeordnet. Deshalb ist es angebracht, die Informationen zuerst zu sammeln, welche Funktionsblöcke ein Device enthält, und da direkt Device Proxy zu generieren. Was aber die Beziehung zwischen Funktionsblock Element und Funktion Element in der XML Datei angeht, ist es wieder hierarchisch, deshalb fällt bei Funktion der Treiber weg und wird deren Generierungsprozess direkt im Treiber von Funktionsblock eingeleitet. Neben den funktionalen Teilen gibt es auch ein paar Dateien für die Einstellungen, die z.B den Ausgabeort bestimmt, für Namenskonflikt bei der Generierung verschiedene Strategie vorschreibt.
Abbild 4 zeigt die Struktur in hierarchischer Form.

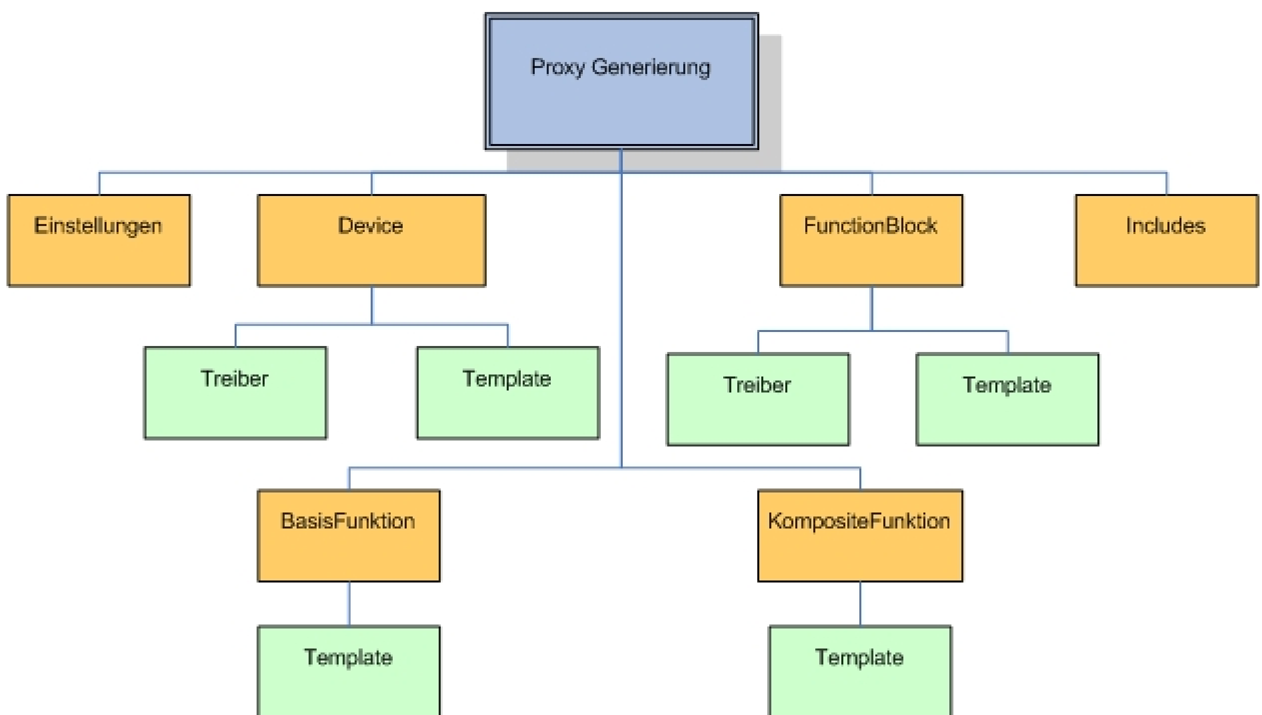


Abbildung 2: Struktur der Proxygenerierung

4.4 Vorstellung des Endproduktes

Die generierten Dateien zusammen mit dem Framework werden eingesetzt in den MMI Entwicklungsaktivitäten der Firma BMW CarIT, was einen erleichterten Zugang auf MOST für die Applikationsprogrammierer ermöglicht hat.
Abbild 4 zeigt die Struktur in hierarchischer Form.

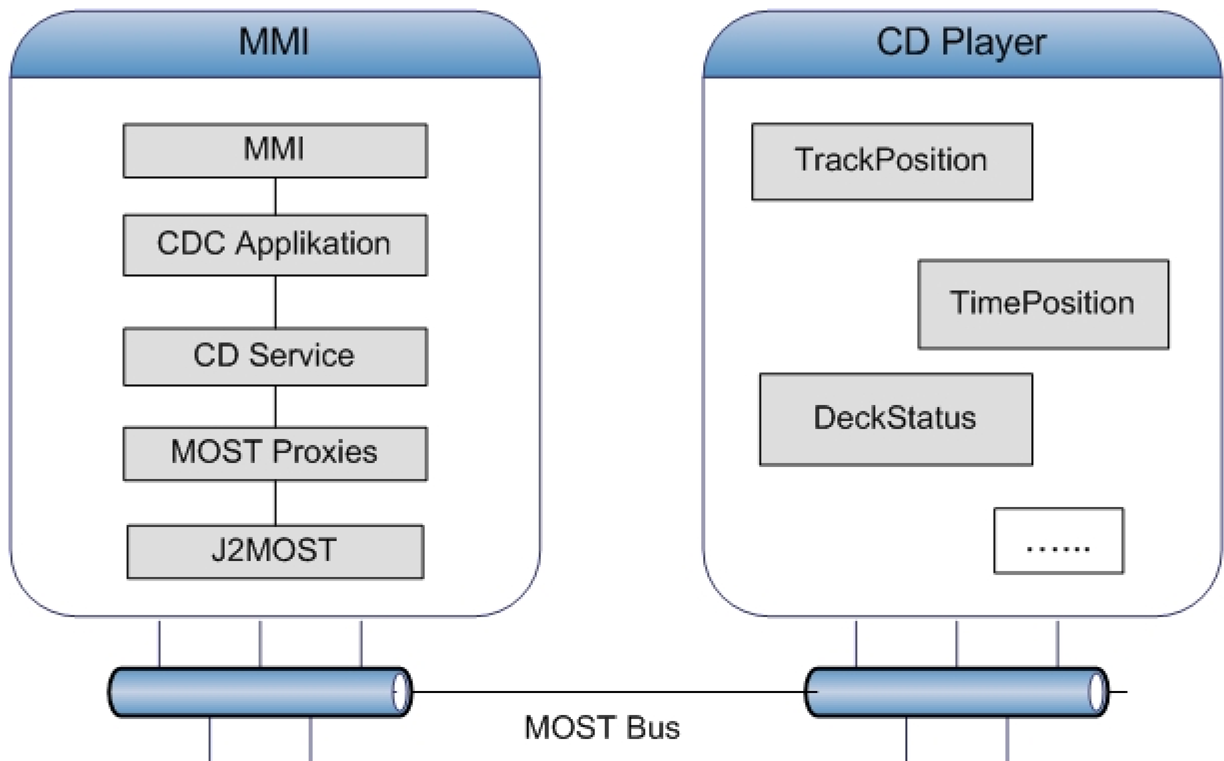


Abbildung 3: produkteinsatz

5 Zusammenfassung und Ausblick

In dieser Ausarbeitung wird zuerst den Lesern in die Landschaft der Codegenerierungstechnik eingeführt. Man ist immer noch auf dem Weg, diese Technik in weiteren Bereichen einzusetzen. Im Grunde genommen wird die bislang aber vor allem in den Bereichen eingesetzt, wo es besonders Sinn macht, z.B. in Compilerbau oder in Framework. Es existiert natürlich auch eine Menge dedizierte Codegeneratoren, die sich gezielt auf einen Bereich konzentrieren. Während der Bearbeitungszeit bin ich immer wieder im Netz auf solche interessanten Codegeneratoren gestoßen, wenn es dabei um strukturierte Eingangsinformation geht, ein Beispiel ist ein Sourceforge Projekt[5], dessen Programm eine vom Benutzer spezifizierte Zustandsmaschine in ausführbare Codes umsetzen kann.
Im Großen und Ganzen liegt die große Herausforderung für die Codegenerierung meiner Meinung nach darin, Eingangsinformationen strukturiert so aufzubereiten, dass die später eindeutig zu interpretieren ist. In manchen Fällen stellt dies kein großes Problem dar, z.B der Zustandsmaschinegenerator und Compilerbau. In anderen Fällen wirkt sich die Sache nicht mehr so einfach und ist meistens mit komplexen Ablaufsteuerungen und menschlichem Wissen, das schwer zu strukturieren ist, z.B trickreiche Optimierungen.

Die Frage, ob die Codegenerierung in konkreten Arbeiten umgesetzt wird, hängt stark von der Art der durchgeführten Tätigkeit ab. Wichtige Faktoren sind Grad der Wiederholung und Aufwand der Tätigkeit, wenn die Codegenerierung nicht umgesetzt würde. Der letzte aber auch ganz wichtige Faktor ist der Aufwand bzw. die Machbarkeit der Codegenerierung.

Literatur

[1] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad; Stal, Michael: *Pattern-Oriented Software Architecture - A System of Patterns.* Chichester England, John Wiley & Son 1996

[2] MOST Cooperation: *Most Specification Rev 2.1.* http://www.mostcooperation.com/downloads/specifications

[3] XDoclet: *Using XDoclet.* http://xdoclet.sourceforge.net/using.html

[4] Bean Shell: *BeanShell-Lightweight Scripting for Java.* http://www.beanshell.org

[5] State Machine Compiler: *SMC: The State Machine Compiler.* http://smc.sourceforge.net