# MobCon

Ein Framework für Mobile Container

**Diplomarbeit**

von Oliver Liemert
Matrikel-Nummer 861210
Vorgelegt am 01.03.2004

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Softwaretechnik

Prof. Dr.-Ing. Mira Mezini

Betreuer Dipl. Ing. Vasian Cepa

# Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 1. März 2004

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Listings

# Abstract

The automation of well-known programming tasks or constructs was already successfully applied to server-side container technologies like EJB [3] and COM+ [4].

MobCon, the abbreviation for the mobile container-framework and the name of this thesis, is the attempt to use the container-technology for applications on mobile devices like mobiles or palms. To achieve this goal, the repetitive programming tasks (technical concerns) must be adapted and the container-framework has to change in a way, that it meets the demands of mobile applications.

MobCon uses code generation techniques to automate repetitive programming tasks. These are encapsulated in so-called transformers, which will be used by application-developers to build their applications. The application-developer can decorate his application with tags, which apply the, already implemented, technical concerns to the application-code.

To complete the framework MobCon offers services like communication or context-information of an application, transparent to the transformers and applications.

The principal motivation of MobCon is the cost-reduction of developing applications. This is possible, because technical concerns don’t have to be readdressed all the time, therefore error analysis and design of a mobile application is simplified. And of course the application code will be reduced, because the developer doesn’t need to write code for the technical concerns that MobCon implements already.

# Kapitel 1

# Einführung

Das Automatisieren von sich wiederholenden und wohlbekannten Programmieraufgaben wurde mit den serverseitigen Container-Technologien wie EJB [3] und COM+ [4] schon erfolgreich praktiziert.

MobCon, die Abkürzung für das mobile Container-Framework und der Name dieser Diplomarbeit, ist der Ansatz die Container-Technologie für Anwendungen mobiler Endgeräte wie Handys oder Palms zu nutzen. Dafür müssen die wiederkehrenden Programmieraufgaben (technische Concerns) angepasst und das Container-Framework so verändert werden, dass es den Ansprüchen mobiler Anwendungen gerecht wird.

MobCon benutzt dabei Code-Generierungs-Techniken um wiederkehrende Programmieraufgaben zu automatisieren. Diese werden in MobCon in so genannten Transformern gekapselt, welche wiederum von Anwendungs-Entwicklern benutzt werden können. Dieser kann, durch dekorieren seiner Anwendung mit Tags, die Transformer und die in ihnen gekapselten technischen Concerns für seine Zwecke nutzen.

Zur Komplettierung des Frameworks bietet MobCon Dienste, wie Kommunikation oder Informationen über den Kontext einer Anwendung, transparent den Transformern und der Anwendung an.

Die Motivation hinter MobCon ist die Reduzierung der Kosten der Anwendungs-Entwicklung. Dies wird ermöglicht, indem technische Concerns nicht wiederholt implementiert werden müssen und somit die Fehlerbeseitigung und das Design einer mobilen Anwendung vereinfacht werden. Allgemein wird natürlich der vom Anwendungs-Entwickler zu schreibende Programmcode reduziert, da dieser den

Code für die, von MobCon schon implementierten, technischen Concerns nicht mehr selbst schreiben muss.

Übersicht über die Gliederung dieser Diplomarbeit:

- **Kapitel 1: Einführung**
  In diesem Kapitel wird erklärt was MobCon ist, warum man und wer es benutzen sollte. Die in dieser Diplomarbeit verwendeten Konventionen werden genauso eingeführt, wie die von der Diplomarbeit zu erfüllenden Ziele.

- **Kapitel 2: Aufbau von MobCon**
  Hier soll der generelle Aufbau von MobCon dargestellt und an einigen Beispielen dokumentiert werden. Es soll eine Übersicht über MobCon und dessen Möglichkeiten gegeben werden.

- **Kapitel 3: Dokumentation für Framework-Entwickler**
  Kurze Beschreibungen über die Funktionsweise und Bedeutung aller Klassen des Frameworks. In diesem Kapitel kann der Entwickler alle Informationen finden, um einen guten Überblick über die Framework-Struktur zu bekommen.

- **Kapitel 4: Dokumentation für Transformer-Entwickler**
  Einführung in die wichtigsten Bestandteil der Transformer-Entwicklung und der Zusammensetzung der Transformer-Module.

- **Kapitel 5: Dokumentation für Anwendungs-Entwickler**
  Hier wird erklärt wie eine mobile Anwendung erstellt und ausgeführt werden kann. Es wird auf die Schnittstellen zu den Transformern hingewiesen und wie diese benutzt werden sollten.

- **Anhang A: Transformer im MobCon-Package**
  In diesem Kapitel werden die im Zuge dieser Diplomarbeit entwickelten Transformer beschrieben.

- **Anhang B: Anwendungs-Code des „Hello World“-Beispiels**
  Der von MobCon generierte Code des in dieser Diplomarbeit eingeführten „Hello World“-Beispiels.

- **Anhang C: Anwendungen der MobCon-Auslieferung**
  Eine Übersicht über die mit MobCon ausgelieferten Beispielanwendungen zur Orientierung oder als Vorlage für eigene Anwendungen.

## 1.1 Konventionen

Zur besseren Lesbarkeit dieser Arbeit wurden verschiedene Schriftarten verwendet. Deren Bedeutung ist folgende:

- `Schrift in Maschinenschrift` kennzeichnet Schlüsselwörter, die mit einer Programmiersprache in Verbindung stehen, wie zum Beispiel Klassennamen, Paketnamen, Methodennamen oder Quelltext selbst.

- *`Kursive Maschinenschrift`,* wird für Tags benutzt.

- ***Fette und Kursive Schrift*** wird verwendet um wichtige Punkte herauszuheben und um Beispiele einzuführen.

## 1.2 Motivation

Die Vorteile die durch die Benutzung von MobCon, oder einem Container-Framework im Allgemeinen, entstehen sind vielfältig. Allen voran soll durch sie der Entwicklungsaufwand einer Anwendung reduziert werden. Dies wird durch verschiedene Eigenschaften von MobCon erreicht:

- Die ***Rollen von Programmierern werden klar verteilt*** und jeder kann in „seinem“ Gebiet entwickeln ohne, im Idealfall, einem anderen Programmierer zu konsultieren.

- Neue Transformer können in einer standardisierten Art und Weise zu MobCon hinzugefügt oder ausgetauscht werden. Somit ist ***MobCon flexibel*** und kann beliebig, den Ansprüchen entsprechend, erweitert oder angepasst werden.

- Der Container- und Transformer-Code ist ***getestet und gedebugt***. Durch seine Wiederverwendung wird die Programmieraufgabe des Anwendungs-Entwicklers signifikant erleichtert. Der Anwendungs-Entwickler muss nur noch die Anwendungslogik schreiben und die schon implementierten Transformer benutzen.

- Der ***Container verwaltet die Kommunikation*** zwischen der Client- und der Serverseite der Anwendung. Das heißt, auch hier kann der Anwendungs-Entwickler einfach das MobCon-Framework benutzen, um Nachrichten zwischen den beiden Seiten auszutauschen.

## 1.3 Rollen in MobCon

Entwickler können in MobCon in drei verschiedene Rollen eingeteilt werden. Diese unterscheiden sich durch ihren Tätigkeitsbereich und ihrer benötigten technischen

Kenntnisse. Zwischen diesen Rollen definiert MobCon Schnittstellen, um die Arbeiten aufteilen zu können. Im Einzelnen werden die Rollen in MobCon folgendermaßen verteilt:

- Der ***Framework-Entwickler*** beschäftigt sich vor allem mit dem Workflow der Code-Generierung von mobilen Anwendungen. Er kann sich aber auch für eine Änderung des Kommunikations-Models verantwortlich zeichnen oder allgemein zugängliche Dienste bereitstellen.

- Der ***Transformer-Entwickler*** kapselt die wiederkehrenden Programmieraufgaben (technische Concerns) in so genannten Transformern. Außerdem stellt er dem Anwendungs-Entwickler die Funktionalität seiner Transformer durch das Definieren von Tags zur Verfügung. Er implementiert die Schnittstelle zwischen Framework-Entwickler und Anwendungs-Entwickler.

- Der ***Anwendungs-Entwickler*** ist auf das Umsetzen der Anwendungslogik mobiler Anwendungen spezialisiert. Dazu benutzt er die vom Transformer-Entwickler erstellten Transformer. Er dekoriert seinen Code mit den vom Transformer-Entwickler definierten Tags um die technischen Concerns zu adressieren, muss aber keine nähere Kenntnis über die Implementierung der technischen Concerns an sich haben.

## 1.4 Ziele

Ziel der Diplomarbeit is einen ersten Prototyp eines mobilen Container-Frameworks zu entwerfen. Er soll Daten persistent halten können und die dafür benötigte Container-Struktur besitzen. Der mobile Teil (Client) des Containers soll mit Hilfe eines oder mehrerer Transformer in Java’s J2ME Mobile Information Device Profile (MIDP) [6] Code generiert werden. Die Tools vDoclet [8], QDox [7] und Velocity [9] sollten benutzt und entsprechend den Bedürfnissen des Container-Frameworks angepasst werden. Die Kommunikationsstruktur von Client und Server soll entwickelt werden, um Daten auszutauschen und Clientdaten persistent auf dem Server zu speichern.

Später wurde das Erstellen zusätzlicher Transformer, die weitere technische Concerns implementieren, und das Entwickeln einer aussagekräftigen Beispielanwendung (MobRay, dazu mehr im Anhang C.4) den Zielen hinzugefügt.

# Kapitel 2

# Aufbau von MobCon

In diesem Kapitel soll der generelle Aufbau von MobCon dargestellt und an einigen Beispielen dokumentiert werden. Hier soll keinesfalls ein tiefer Einblick, sondern vielmehr eine Übersicht über MobCon und dessen Möglichkeiten gegeben werden. Nähere Informationen über die Benutzung von MobCon oder über die von den einzelnen Entwicklern benötigte Dokumentation kann im Kapitel 3 für Framework-Entwickler, im Kapitel 4 für Transformer-Entwickler und im Kapitel 5 für Anwendungs-Entwickler nachgelesen werden.

Übersicht über dieses Kapitel:

- **Definitionen**
  Übersicht und Nachschlagewerk über die im Zuge dieser Diplomarbeit benutzten Definitionen.

- **Technische Concerns in MobCon**
  Einführung der zur Zeit von MobCon adressierten technischen Concerns.

- **Architektur**
  Erläuterung der MobCon zugrunde liegenden Architektur.

- **Umgebungen und Werkzeuge**
  Vorstellung der zur Implementierung von MobCon benötigten oder benutzten Programmier-Werkzeuge und -Umgebungen.

- **Template-Objekte**
  In diesem Abschnitt werden die für MobCon essentiellen Template-Objekte vorgestellt. Es wird gezeigt wie diese vereinigt und mit Tags dekoriert werden können.

- **Transformer**
  Überblick über einen zentralen Teil von MobCon, den Transformern. Es werden der Aufbau des Transformer-Moduls und dessen einzelne Komponenten beschrieben. Außerdem wird auf den Ablauf der Verarbeitung der Transformer näher eingegangen.

- **MobCon-Code**
  Die grundlegende Funktionsweise des MobCon-Codes und ein einführendes „Hello World“-Beispiel werden in diesem Abschnitt erläutert.

- **Template-Pattern in MobCon**
  Es wird gezeigt wie das Template-Pattern in MobCon zur Anwendung gebracht wird. Das heißt wie der Code des Anwendungs-Entwicklers von dem von MobCon generierten Code getrennt wird.

- **Kommunikation**
  Der Ablauf der Kommunikation zwischen Client und Server über das MobCon-Framework wird in diesem Abschnitt vorgestellt.

- **Deployment Descriptor**
  Einführung des Deployment Descriptor von MobCon und kurze Beschreibung dessen Zwecks.

- **Ausblick**
  Eventuelle Verbesserungsvorschläge oder Ideen zur Weiterentwicklung des MobCon-Frameworks.

## 2.1 Definitionen

Es folgt eine Übersicht über die wichtigsten Definitionen bezüglich MobCon. Diese Definitionen werden im Laufe dieser Arbeit benutzt und teilweise nochmals eingehender betrachtet. Dieser Abschnitt soll als Übersicht und Nachschlagewerk zur besseren Verständlichkeit dieser Arbeit dienen. Die Schlagwörter sind alphabetisch geordnet, damit sie beim Nachschlagen besser gefunden werden können

**Anwendungs-Code**

Der Java-Source-Code für die Clientseite, der aus dem MobCon-Code und durch den Transformationsprozess der Transformer von MobCon generiert wird.

### Client

Mit Client wird im Folgenden der Teil des mobilen Containers bezeichnet, der auf dem mobilen Endgerät ausgeführt wird. Zur Emulation dieses Endgeräts wurde Javas J2ME Wireless Toolkit 2.0 [12] benutzt.

### MobCon-Code

Der MobCon-Code ist der mit Tags dekorierte Code, aus dem später die gesamte mobile Anwendung entsteht. Hier befindet sich neben den dekorierten Objekten auch die eigentliche Anwendungslogik.

### Mobile Anwendung

Eine Anwendung besteht aus einer Menge von Transformern, also der implementierten technischen Concerns, und der Anwendungslogik. Mit dem Begriff „mobile Anwendung“ werden der Teil der Client- und der Serverseite der Anwendung zusammengefasst.

*CID (Container-ID)*: Kennzeichnet die Anwendung durch eine globale, eindeutige, 128-Bit lange Zahl.

### Server

Der serverseitige Teil des mobilen Containers wird kurz als Server bezeichnet. Hier sollte der größte Teil der Anwendungslogik zu finden sein. Für dessen Realisierung wurde Javas SDK 1.4.1[17] verwendet.

### Tag

Das englische Wort „Tag“ könnte in Deutsch mit dem Wort „Textmarke“ oder „Auszeichnung“ übersetzt werden. Tags werden immer eine Arbeit auf höherer Ebene bewirken, da die konkrete Programmierung in Java-Code teilweise durch eine Programmierung mit Tags ersetzt wird. Diese Tags sind in ihrer Bedeutung mit Eigenschaften gleichzusetzen, dass heißt dem Tag dekorierten Code wird eine gewisse Eigenschaft zugeschrieben, welche später durch MobCon implementiert wird. In MobCon bewirken Tags meistens eine, von der Eigenschaft des Tags abhängige, Generierung von Anwendungs-Code.

Mehrere unterschiedliche Arten von Tags kommen in MobCon vor:

- Tags, mit denen die Klasse des MobCon-Codes dekoriert werden kann. Mit diesen Tags wird festgelegt welche Transformer auf den MobCon-Code angewandt werden.

- Tags, mit denen die Objekte des MobCon-Codes dekoriert werden. Diese Tags stellen Eigenschaften des jeweiligen Objekts da, wie zum Beispiel sein Label oder seine Größe.

- Die Tags, welche in den so genannten TagDic-Dateien eines Transformers verwendet werden, dienen dem Zweck der Dokumentation und beschreiben welche Wirkung dieser Transformer auf den Anwendungs-Code hat.

- Auch die Template-Objekte können mit Tags dekoriert werden. Dies kann zum Beispiel für das Auffinden spezieller Template-Objekte oder hinzufügen zusätzlicher Semantik zu diesen Objekten benutzt werden.

**Template-Objekte**

Sie ermöglichen die Struktur des generierten Anwendungs-Codes in einem Abstract Syntax Tree (AST) zu speichern und diesen dann später als Java-Source auszugeben

**Transformer**

Transformer implementierten die technischen Concerns. Diese können durch den Anwendungs-Entwickler benutzt beziehungsweise adressiert werden, indem der MobCon-Code mit Tags dekoriert wird.

**Transformer-Modul**

Ein Transformer-Modul stellt das gesamte Paket dar, das zur Verwendung eines Transformers in einer mobilen Anwendung benötigt wird. Es besteht aus der TagDic-Datei, der Manifest-Datei, dem Transformer-Code und eventuell dem serverseitigen Transformer-Code.

**Transformer-Code**

Transformer-Code wird der im Transformer-Modul befindliche Code des Velocity-Templates genannt, der den technischen Concern des Transformers implementiert.

**Transformer-Typ**

Das ist der technische Concern, den ein Transformer implementiert.

*TID (Transformer-Typ-ID)*: Jeder durch einen Transformer implementierte technische Concern hat eine eindeutige Kennung. Diese wird vom Transformer-Entwickler bei der Erstellung des Transformers diesem zugeteilt. Die Kennung ist für alle Transformer, die den gleichen technischen Concern implementieren, die Selbe.

**Transformer-Instanz**

Eine Transformer-Instanz ist die konkrete Implementierung eines Transformer-Typs durch einen Transformer. Dies muss eingeführt werden, da es mehr als eine konkrete Implementierung für ein und denselben technischen Concern in einer Anwendung geben kann.

*IID (Transformer-Instanz-ID)*: Eindeutige Zahl, die jeder Transformer Instanz, bei 1 angefangen, zugeteilt wird.

**Transformer-Gruppe**

Ein oder mehrere Transformer zwischen denen keine Abhängigkeiten bestehen.

**Transformationsprozess**

In diesem Vorgang werden die Transformer gemäß der Tags im MobCon-Code aufgerufen und abgearbeitet. Resultat ist der generierte Anwendungs-Code sein.

## 2.2 Implementierte technische Concerns

Technische Concerns in MobCon sind ***kapselbare Programmieraufgaben oder Eigenschaften einer Anwendung***, die immer wieder von verschiedenen Anwendungen adressiert werden.

Die Module, die diese technischen Concerns kapseln und implementieren werden in MobCon Transformer genannt. Diese können vom Anwendungs-Entwickler benutzt werden, um nicht wiederholt den technischen Concern implementieren zu müssen. Auf deren Aufbau wird in Abschnitt 2.6 näher eingegangen.

Folgend ein Überblick über die zur Zeit von MobCon adressierten technischen Concerns, beziehungsweise der Transformer. Da die Transformer in MobCon in englischer Sprache benannt wurden, steht der englische Name in Klammern hinter den deutschen Bezeichnungen. Die genaue Funktionsweise der einzelnen Transformer und wie diese benutzt werden können, kann in Anhang A nachgelesen werden.

**Persistenz der Daten (Data persistence):**

Der Transformer verwaltet die persistente Speicherung von Feldern oder Objekten im nichtflüchtigen Speicher des mobilen Endgeräts oder auf dem Server.

**Darstellungs-Manager (Screen Manager, User Interface):**

Die Anzahl der Möglichkeiten in Javas J2ME MIDP Textboxen, Listen, Displays, oder Anzeigeobjekte im Allgemeinen zu erstellen sind beschränkt. Dadurch können diese leicht in einem Transformer gekapselt werden, der dem Entwickler bei deren Erstellung hilft und daraus resultierende Standardstrukturen entwickelt.

**Kontext- und Sessioninformationen (Session):**

Der Transformer speichert Kontext- und Sessioninformation auf dem Server und stellt, wenn erforderlich, sie auf dem mobilen Endgerät wieder her. Diese Informationen können zum Beispiel der letzte besuchte Bildschirm einer Anwendung, Benutzername oder Passwort sein.

**Bildanpassung (Image, Data Adaptation):**

Der Transformer kann Bilder auf der Serverseite an die Bedürfnisse des mobilen Endgeräts anpassen (Farbe, Größe, Auflösung…) und diese angepassten Daten zu ihm senden. Dadurch wird knappe Rechenzeit auf dem Endgerät eingespart und auf den Server ausgelagert.

**Loggen (Logging):**

Alle Methodenaufrufe oder/und Aufrufe von Befehlen können geloggt werden. Dabei werden bei einem Methodenaufruf die Transformer auf der Konsole ausgegeben, die den Code dieser Methode verändert haben. Dadurch können später bei der Fehlersuche Hinweise erhalten werden, an welcher Stelle nach dem Fehler gesucht werden sollte.

**Verschlüsselung (Encryption):**

Dieser Transformer ermöglicht vom Client zum Server geschickte Daten mit den Algorithmen von Bouncy Castle [14] zu verschlüsseln.

## 2.3 Architektur

Die Architektur von MobCon kann in vier Teile zerlegt werden. Diese werden in Abbildung 2.1 dargestellt und folgend kurz vorgestellt:

- Das ***MobCon-Framework***, dessen Komponenten in Abbildung 2.1 durch einen schwarzen Kasten zusammengefasst werden. Das Framework besteht aus den Template-Objekten, der Kommunikationseinheit, der Steuereinheit und den von MobCon benutzten Tools QDox und vDoclet (gestrichelt dargestellt, weil externe Werkzeuge). Das Framework parst den MobCon-Code mit Hilfe von QDox und vDoclet, das Ergebnis wird an die Transformer übergeben. Es liest die Konfigurationsdateien, verwaltet die Transformer und steuert den Workflow des Transformationsprozesses, sowie die Kommunikation zwischen Server- und Clientteil der Anwendung. Die Template-Objekte werden vom Framework zur Verfügung gestellt und von den Transformern zur Speicherung des generierten Codes benutzt.

- Dem ***MobCon-Code***, aus dem die mobile Anwendung generiert und die verschiedenen Transformer adressiert werden.

- Die ***Transformer***, beziehungsweise die implementierten technischen Concerns. Diese werden aufgrund der im MobCon-Code verwendeten Tags aktiviert und führen Veränderungen an den Template-Objekten durch, gemäß ihrer implementierten Concerns. Aus diesen Objekten wird später der Anwendungs-Code generiert. Die Transformer sind in der Template-Sprache Velocity-Script geschrieben.

- Die ***mobile Anwendung*** selbst. Diese kann nochmals in eine Client- und eine Serverseite eingeteilt werden. Sie entsteht aus dem MobCon-Code, der die Anwendungslogik definiert und mit Hilfe des MobCon-Frameworks von den vom MobCon-Code adressierten Transformern in ausführbaren Code für die Client- und Serverseite umgewandelt wird.

Die mobile Anwendung wird nicht nur alleine auf dem mobilen Endgerät ausgeführt, die Architektur von MobCon kann vielmehr als Erweiterung des Client-Server-Modells angesehen werden. Das bedeutet, dass ein Teil des Containers auf der Serverseite (Server) ausgeführt wird und ein anderer Teil des Containers auf dem mobilen Endgerät (Client). Die zwei Teile ergeben zusammen eine einzelne mobile Anwendung. Sie müssen miteinander kommunizieren und brauchen daher beide Kommunikationslogik, welche das Container-Framework zur Verfügung stellt und das Senden der Daten zwischen den beiden Teilen übernimmt. Der Großteil der Anwendungs- und Containerlogik sollte auf der Serverseite sein, da die Ressourcen des mobilen Endgeräts oft beschränkt sind und mit ihnen sparsam umgegangen werden sollte.

In den folgenden Abschnitten wird auf die einzelnen Komponenten näher eingegangen, um so einen globalen Überblick über MobCon zu schaffen.

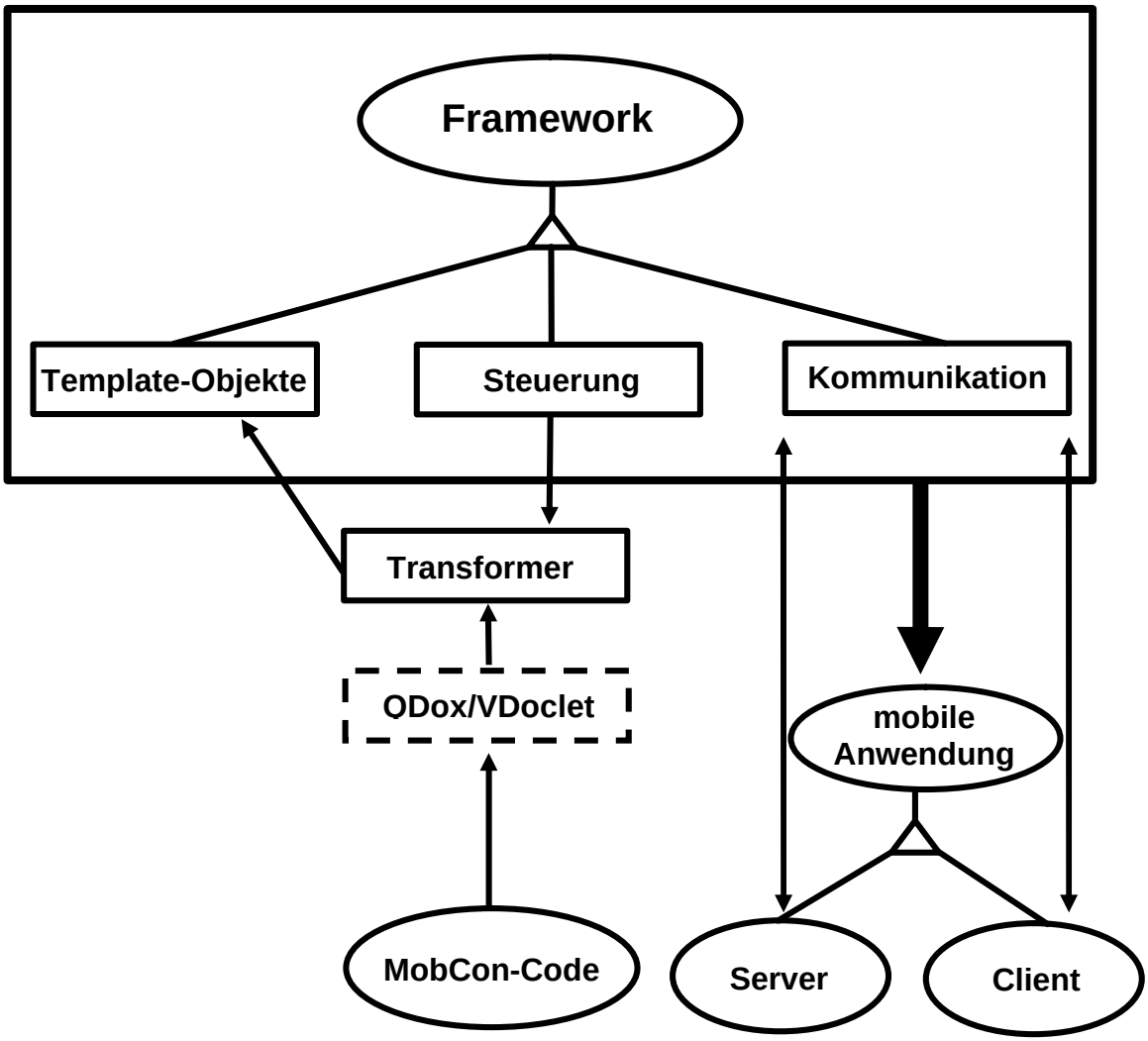


*Abbildung 2.1: Architektur von MobCon*

## 2.4 Umgebung und Werkzeuge

Im Folgenden werden die zur Implementierung von MobCon benötigten Werkzeuge und verwendeten Umgebungen für Framework, Client, Server und Transformer vorgestellt.

Das MobCon-Framework und die serverseitigen Anwendungen wurden in Suns **Java 2 SDK** Standard Edition Version 1.4.1 [17] verfasst.

Der Anwendungs-Code wird in **Javas J2ME** [5] **MIDP** (Mobile Information Device Profile) [6] entwickelt beziehungsweise generiert. Diese Sprache ist ein, an mobile

Endgeräte angepasster, Dialekt von Java und relativ hardwareunabhängig, so dass der Entwickler bei der Erstellung der Anwendung die Hardwaredetails der virtuellen Maschine überlassen kann. Des Weiteren kann diese Sprache von vielen mobilen Endgeräten verstanden und benutzt werden.

Java's **JavaDoc** [10] ist ein Werkzeug, welches API-Dokumentation im HTML-Format aus Java-Source-Code generiert. Mit seiner Hilfe können die „@-Tags" gelesen werden, die im MobCon-Code zur Dekorierung der Objekte benutzt werden.

Thoughtworks' **QDox** [7] ist ein Parser, der mit JavaDoc „@-Tags" dekorierte Klassen-, Interface- und Methoden-Informationen aus Java-Sourcen extrahiert und zur Verfügung stellt.

Apache's **Velocity** [9] ist eine auf Java basierende Template-Sprache. Sie wurde gewählt, weil es eine einfache Skriptsprache ist, die aber dennoch erlaubt jegliche Java-Objekte in Velocity-Scripts zu benutzen. In ihr wird der Transformer-Code formuliert.

**vDoclet** [8] benutzt JavaDoc um Java-Sourcen zu parsen und ein abstraktes Klassen-Modell der Java-Sourcen zur Verfügung zu stellen (`DocInfo`-Objekt). Dieses Modell wird an Velocity weitergegeben in dem es dann durch Velocity-Templates, im Falle von MobCon Transformer-Code genannt, benutzt werden kann.

## 2.5 Template-Objekte

Die Template-Objekte sind ein wichtiger Bestandteil von MobCon. Sie ermöglichen die Klassenstruktur des generierten Anwendungs-Codes in einem Abstract Syntax Tree (AST) zu speichern, welcher von mehreren Template-Objekten dargestellt wird. Die Template-Objekte können ihrerseits mit Tags dekoriert werden, mit deren Hilfe Eigenschaften dieser Objekte festgelegt werden können.

Nachfolgend soll ein Überblick über die verwendeten Template-Objekte und deren Funktionsweise gegeben werden. Das UML- Diagramm dieser Objekte ist der Abbildung 2.2 zu entnehmen.

Die `FieldTemplate`-Objekte speichern alle Informationen die notwendig sind, ein Feld genau zu beschreiben. Das sind der Name, Zugriffsmodifikatoren, Typ und Wert welche mit entsprechenden get/set-Methoden verändert werden können. Diese Objekte können zusätzlich mit Tags dekoriert werden.

`TaggedCode` ist eine Klasse, die eine Code-Zeile als String speichert. Ein `TaggedCode`-Objekt kann auch zusätzlich mit einem oder mehreren Tags versehen

werden. Dies ist hilfreich um später nach bestimmten Code-Zeilen zu suchen oder diese mit Eigenschaften zu versehen.

Die `MethodTemplate`-Objekte beschreiben eine Java-Methode. Diese Objekte bestehen neben Namen, Zugriffsmodifikator und Typ auch aus den Exceptions und den Parametern der Methode als `FieldTemplate`-Objekte. Der eigentliche Code der beschriebenen Methode wird durch die drei Methoden `addBegin(TaggedCode)`, `addBody(TaggedCode)` und `addEnd(TaggedCode)` hinzugefügt. Die Methodennamen implizieren schon, dass Code grundsätzlich an drei verschiedenen Stellen einer Methode eingefügt werden kann: Dem Anfang, in der Mitte oder am Ende. Der in diesen drei Stellen gespeicherte Code wird in der Reihenfolge Anfang, Mitte, Ende ausgegeben. Wobei ein Code-Element, das vor einem anderen Code-Element innerhalb dieser drei Stellen eingefügt wurde, auch vor dem später eingefügtem Element steht („First In – First Out“). Wie die `FieldTemplate`- und `TaggedCode`-Objekte kann ein `MethodTemplate`-Objekt auch mit zusätzlichen Tags dekoriert werden.

Es können zwei `MethodTemplate`-Objekte mit gleichen Namen, Rückgabewert und Parametern vereinigt werden. Hierbei werden standardmäßig die drei Teile einer Methode (Begin, Body, End) der zwei beteiligten `MethodTemplate`-Objekte miteinander vereinigt. Der Ablauf der Vereinigung wird durch ein Objekt welches das `MixTemplate`-Interface implementiert, bestimmt. Dieses wird von dem `ClassTemplate`-Objekt zu den, zu der Klasse gehörenden, `MethodTemplate`-Objekten propagiert.

Den `ClassTemplate`-Objekten können neben dem Namen, dem Zugriffsmodifikator, der von dieser Klasse erweiterten Klasse, der importierten Klassen und der implementierten Interfaces auch die `MethodTemplate`- und `FieldTemplate`-Objekte hinzugefügt werden. Diese repräsentieren die Methoden und Felder der Klasse. Um das Verhalten der Vereinigung zweier Klassen zu bestimmen, kann hierzu ein `MixTemplate`-Objekt definiert werden, welches an alle eingefügten `MethodTemplate`-Objekte weiter propagiert wird.

***Beispiel: Benutzung von Template-Objekten in Transformern***

```
1  ClassTemplate CT1 = new ClassTemplate();
2  CT1.setAccess(“public”);
3  CT1.setName(“TestClass”);
4
5  FieldTemplate FT1 = new FieldTemplate();
6  FT1.setType(“String”);
7  FT1.setName(“name”);
8  CT1.addField(FT1);
```

```
9
10 MethodTemplate MT1 = new MethodTemplate();
11 MT1.setName(“init”);
12 MT1.setAccess(“public”);
13 MT1.setName(“void”);
14 MT1.addBody(“name = \”Mike\”;”);
15 CT1.addMethod(MT1);
```

*Listing 2.1: Benutzung von Template-Objekten in Transformern*

Erläuterungen zu Listing 2.1:

- *Zeile 1-3*: Das `ClassTemplate`-Objekt wird instanziert, welches eine Klasse mit Namen `TestClass` und Zugriff `public` modelliert.

- *Zeile 8*: Das in Zeile 5-7 geschaffene `FieldTemplate`-Objekt wird dem `ClassTemplate`-Objekt hinzugefügt. Somit wird es später im generierten Anwendungs-Code als Codezeile `String name;` erscheinen.

- *Zeile 14*: Die Code-Zeile `name = „Mike“;` wird dem `MethodTemplate`-Objekt in der mittleren Region seines Methoden-Körpers hinzugefügt.

### 2.5.1 Vereinigen von ClassTemplate- oder MethodTemplate-Objekten

Eine Vereinigung ist in MobCon notwendig, da jeder Transformer ein eigenes `ClassTemplate`-Objekt produziert und alle diese Objekte im Zuge des Transformationsprozesses zu einem einzigen, den gesamten Anwendungs-Code repräsentierenden, `ClassTemplate`-Objekt zusammengefügt werden müssen. Aus diesem Objekt wird der Anwendungs-Code generiert. Es wird somit ermöglicht, die Methoden der Klasse nach und nach mit Code von jedem Transformer zu füllen.

Die Vereinigung zweier `MethodTemplate`-Objekte mit gleicher Signatur, wird in MobCon normalerweise durch die Vereinigung der Inhalte der drei Teile einer Methode (Begin, Body, End) und deren Tags vorgenommen. Zwei `ClassTemplate`-Objekte mit gleichen Namen werden in MobCon durch die Vereinigung ihrer `MethodTemplate`-Objekte und `FieldTemplate`-Objekte zusammengefasst.

Wird ein anderes Verhalten als das Standardverhalten bei dem Vereinigungsvorgang erwünscht, so kann dieses verändert werden indem das `MixTemplate`-Interface implementiert und die resultierende Klasse dem zu vereinigenden `ClassTemplate`-Objekt übergeben wird, welches diese Klasse wiederum an seine `MethodTemplate`-Objekte weiter propagiert. Dieses Interface besteht aus zwei Methoden: `mixClassTemplates(ClassTemplate ct1, ClassTemplate ct2)` und `mixMethodTemplates(MethodTemplate mt1, MethodTemplate`

`mt2`). Die Logik dieser zwei Methoden bestimmt wie die Objekte miteinander vermischt werden.

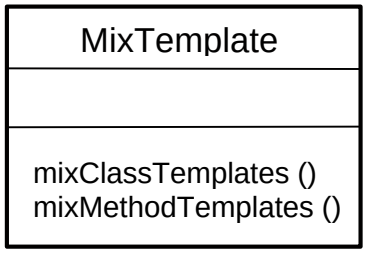


*Abbildung 2.3: Interface MixTemplate*

***Beispiel: Vereinigung von MethodTemplate-Objekten***

Der Code der `init`-Methode soll durch eine Ausgabe auf der Konsole erweitert werden. Der Code von Listing 2.1 wird durch folgenden Code ergänzt:

```
16 MethodTemplate MT2 = new MethodTemplate();
17 MT2.setName(“init”);
18 MT2.setAccess(“public”);
19 MT2.setName(“void”);
20 MT2.addBody(“System.out.println(name);”);
21 CT1.addMethod(MT2);
```

*Listing 2.2: Vereinigung von MethodTemplate-Objekten*

Die Umwandlung des, sich aus Listing 2.1 und Listing 2.2 ergebenden, `ClassTemplate`-Objekts in Anwendungs-Code würde folgendes Ergebnis liefern:

```
1 public Class TestClass{
2   String name;
3   public void init(){
4     name = “Mike”;
5     System.out.println(name);
6   }
7 }
```

*Listing 2.3: Resultierender Anwendungs-Code*

### 2.5.2 Tagging von Template-Objekten

Die `TaggedCode`-, `MethodTemplate`- und `FieldTemplate`-Objekte sind alle von einer Klasse abgeleitet, wie in Abbildung 2.2 zu sehen ist. Diese Klasse nennt sich `TaggedElement` und bewirkt, dass Tags den Objekten hinzugefügt werden können.

Durch das Dekorieren von `TaggedCode`-, `MethodTemplate`- oder `FieldTemplate`-Objekten wird es ermöglicht diesen Konstrukten noch mehr Semantik zu geben. Die Nutzungsmöglichkeiten dieser Tags ist vielseitig.

Ein Beispiel wäre die Arbeitsweise des Transformers welcher den Logging-Concern implementiert. Die mit MobCon entwickelten Transformer dekorieren erstellte `MethodTemplate`-Objekte zusätzlich mit einem Tag, der ihrem Präfix aus dem TagDic entspricht. Bei Benutzung des Logging-Transformers werden dann alle Tags eines `MethodTemplate`-Objekts bei dem Aufruf der generierten Methode ausgegeben. Dadurch wird das debugen einer Anwendung erleichtert, da alle Transformer die diese Methode manipuliert haben, ein Zeichen dafür, anhand eines Tags entsprechend ihres Namens hinterlassen. Auf diese Weise können leichter die Transformer gefunden werden, die diese Methode verändert haben und für einen eventuellen Fehler verantwortlich sind.

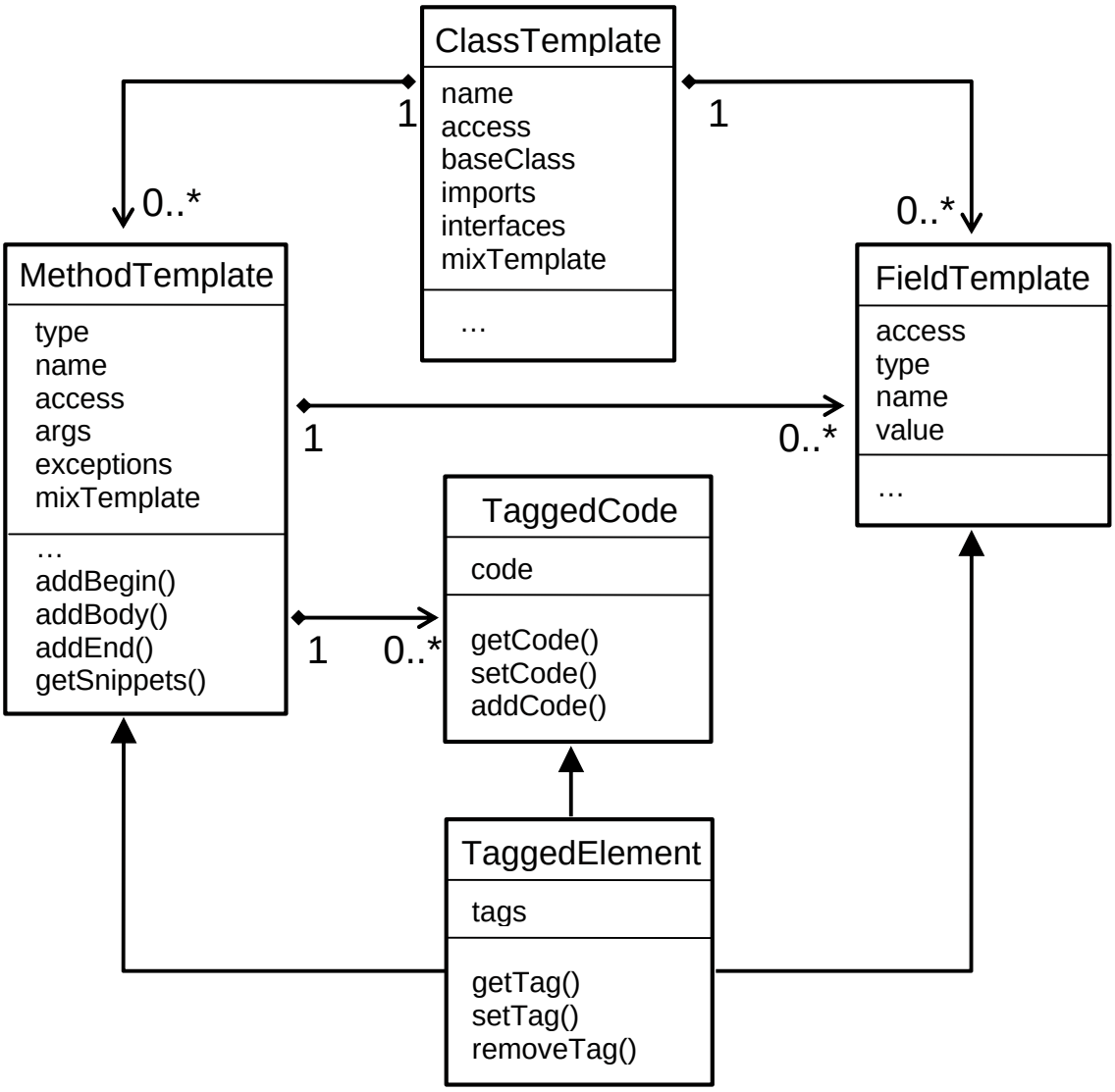


*Abbildung 2.2: UML-Diagramm der Template-Objekte*

***Beispiel: Benutzung des TaggedCode-Objekts***

Dieses Beispiel soll zeigen, wie Abhängigkeiten zwischen Transformern durch das `TaggedCode`-Objekt formuliert werden können. Hierbei soll ein Transformer `TaggedCode`-Objekte mit einen Tag *`encrypt`* versehen, falls gewünscht wird, dass das Objekt der rechten Seite der Zuweisung verschlüsselt wird. Der Encryption-Transformer durchsucht anschließend alle generierten `MethodTemplate`-Objekte nach `TaggedCode`-Objekten die mit dem Tag *`encrypt`* dekoriert sind und verändert deren gespeicherten Code entsprechend der Verschlüsselungslogik.

Code in einem Transformer:

```
1  ClassTemplate CT1 = new ClassTemplate();
2  CT1.setAccess(“public”);
3  CT1.setName(“TestClass”);
4
5  MethodTemplate MT1 = new MethodTemplate();
6  MT1.setName(“init”);
7  MT1.setAccess(“public”);
8  MT1.setName(“void”);
9  MT1.addBody(“encrypted = secureCode;”, “encrypt”);
10 CT1.addMethod(MT1);
```

*Listing 2.4: Hinzufügen des TaggedCode-Objekts im Transformer*

In Zeile 9 werden die Parameter der `addBody()`-Methode in ein `TaggedCode`-Objekt umgewandelt und der Methode hinzugefügt. Dieses Objekt besteht aus der Code-Zeile `encrypted = secureCode;` und dem Tag *`encrypt`*.

Code im Encryption-Tranformer:

```
1  MethodTemplate MTagged = CT1.getMethod(“init”);
2  TaggedCode tc = MTagged.getCode(“encrypt”);
3  #if(tc)
4  ... doSomething ...
```

*Listing 2.5: Suchen der TaggeCode-Objekte im MethodTemplate*

Der Encryption-Transformer sucht nach `TaggedCode`-Objekten in dem Code der Methode `init` (Zeile 2) und verschlüsselt diese falls sie gefunden werden.

## 2.6 Transformer

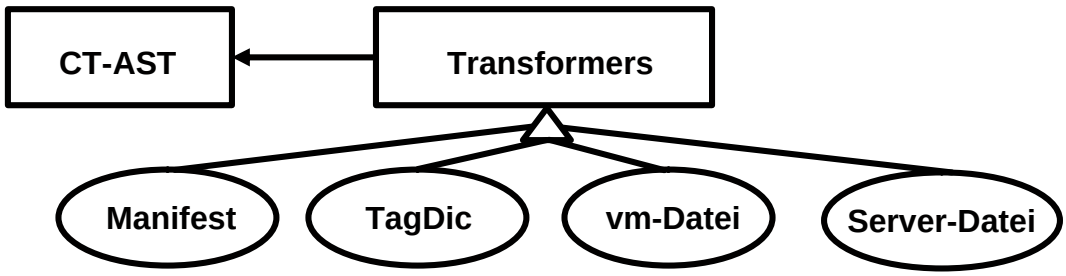


*Abbildung 2.4: Aufbau eines Transformer-Moduls*

Ein Transformer stellt in MobCon einen technischen Concern dar, so wie dieser in Abschnitt 2.2 beschrieben wurde. Ein Transformer besteht hauptsächlich aus einer Velocity-Script-Datei, welche den technischen Concern implementiert. Als Input werden dem Velocity-Script der von vDoclet geparste Objekt-Baum des mit Tags dekorierten MobCon-Codes und ein `ClassTemplate`-Objekt übergeben. Der Transformer verarbeitet den Objekt-Baum und nimmt Änderungen an dem `ClassTemplate`-Objekt vor, so dass die Logik des technischen Concerns sich in diesem widerspiegelt. Zum Beispiel könnte ein Transformer welcher Zugriffsmethoden auf Felder generiert get/set-Methoden als `MethodTemplate`-Objekte für alle entsprechend dekorierten Felder dem `ClassTemplate`-Objekt hinzufügen.

Ein Transformer wird durch eine Java JAR-Datei repräsentiert, sie wird im folgenden Transformer-Modul genannt. Das Transformer-Modul setzt sich aus vier Komponenten zusammen auf die noch einmal im Detail eingegangen wird:

- **Manifest-Datei**
  Enthält für den Transformationsprozess benötigte Informationen über den Transformer.

- **TagDic**
  Hier werden die Tags, die später im Source-Code verwendet werden, definiert.

- **Server-Datei**
  Serverseitige Transformerlogik.

- **vm-Datei**
  Velocity-Script, das einen technischen Concern implementiert.

### 2.6.1 Manifest-Datei

In dem Manifest des Transformers werden folgende Einträge erwartet:

- **Transformer-Name**: Der Name des Transformers
- **TID**: Transformer-Type-ID
- **IID**: Transformer-Type-Instance-ID
- **File-Name**: Name der Transformer-Datei im JAR
- **Tag-File**: Name der TagDic-Datei im JAR
- **Use-Before**: Eine durch "/" getrennte Liste von Transformer-IDs (TID.IID), die vor diesem Transformer benutzt werden müssen
- **Use-After**: Eine durch "/" getrennte Liste von Transformer-IDs (TID.IID), die erst nach diesem Transformer benutzt werden dürfen
- **Server-File**: Der Name der Datei im JAR, welche die serverseitige Transformer-Logik hält

***Beispiel: Manifest des Session-Transformers***

```
1  Manifest-Version: 1.0
2
3  name: Transformer
4  Transformer-Name: Session
5  TID: 04
6  IID: 01
7  File-Name: session.vm
8  Tag-File: session.tag
9  Server-File: SessionTrans
10
11 name: Dependencies
12 Use-Before: 02.01
13 Use-After:
```

*Listing 2.6: Manifest des Session-Transformers*

### 2.6.2 TagDic

Tags werden im MobCon-Code verwendet, um zu bestimmen welcher Transformer von der Anwendung benutzt wird und welche Veränderungen am Anwendungs-Code vorgenommen werden sollen. Sie könnten als Eigenschaften der dekorierten Objekte

angesehen werden. Zum Beispiel könnte ein Feld im MobCon-Code mit einem Tag *access* versehen sein, was bedeutet, dass für dieses Feld get/set-Methoden generiert werden sollen. Also hat dieses Feld die Eigenschaft, die im englischen „accessible" lauten würde.

Das TagDic-Objekt des Transformers hält Schlüssel-/Wertpaare. Die Schlüssel sind Eigenschaften (wie „accessible"), die den Objekten zugewiesen werden können, und die Werte die tatsächlich benutzten Tags (wie *access*) in der Anwendung. Das bedeutet, dass die in der Anwendung benutzten Tags frei definiert werden können. Ein spezieller Schlüssel ist „PREFIX", dessen Wert das Präfix zu allen im TagDic-Objekt vorhandenen Tags ist.

Wenn ein TagDic-Objekt erstellt wird, liest es als erstes seine mit ihm verbundene TagDic-Datei ein. Dies ist eine Datei, in der durch „=" getrennte Schlüssel-/Wertpaare gespeichert sind. Die wichtigste Methode eines TagDic-Objektes ist die Methode `String getTag(String key)`. Diese Methode liefert zu einer Eigenschaft den Tag, mit dem im MobCon-Code Objekte dekoriert werden können.

***Beispiel: Benutzung des TagDics***

Um zu verdeutlichen wie das TagDic-Objekt benutzt wird, ist hier einmal der typische Code eines Transforners aufgelistet. Mit `$tagDic.getTag("accessible")` (Zeile 2, Listing 2.7) wird der Tag zurückgegeben, der die Eigenschaft „accessible" widerspiegelt. Also sucht das TagDic-Objekt in der TagDic-Datei nach dem Schlüssel „accessible" (Zeile 2, Listing 2.8) und gibt, zusammen mit dem Präfix (Zeile 1, Listing 2.8) der allen Tags dieses TagDic-Objekts vorangestellt wird, *@dp.access* zurück. Die Methode `$field.getTag(String tag)` (Zeile 2, Listing 2.7) liefert `true`, falls das Feld `$field` mit dem Tag *tag* dekoriert ist. Das heißt, dass im Transformer-Code nur nach den Eigenschaften, nicht aber direkt nach den Tags gesucht wird. Somit bleiben die Tags im MobCon-Code frei definierbar. Die Methode liefert `true` für das in Zeile 3 des Listings 2.9 dekorierte `String`-Objekt `id`. Daher wird die Methode `dp_gen_field ($field)` (Zeile 4, Listing 2.7) im Transformer aufgerufen.

Transformer-Code:

```
1  #set($tag = false)
2  #set($tag = $field.getTag($tagDic.getTag("accessible")))
3  #if ($tag)
4    #dp_gen_field ($field)
5  #end
```

*Listing 2.7: Benutzung des TagDics in einem Transformer*

Inhalt der TagDic-Datei des Transformers:

```
1  PREFIX = @dp
2  accessible = access
3  storeable = store
```

*Listing 2.8: Inhalt des TagDics*

Benutzung des frei definierten Tags im MobCon-Code:

```
1  public class Test{
2    /**
3      * @dp.access
4     /*
5     String id;
```

*Listing 2.9: Benutzung des frei definierten Tags im MobCon-Code*

### 2.6.3 Transformer-Code

Der Transformer-Code ist das Velocity-Script, welches den technischen Concern dieses Transformers implementiert. Aus diesem Code heraus können mehrere Variabeln benutzt werden, die schon vom MobCon-Framework bereitgestellt werden, dazu mehr in Abschnitt 4.1.

***Beispiel: Code des Velocity-Scripts des Session-Transformers***

```
1  ##
2  ##
3  #****** Macro dp_meth_get******
4  # Function: Generates the get-Method of the Field
5  *#
6  #macro(dp_meth_get $type $name)
7  #set($MT = $vdoclet.makeBean("mobcon.ct.MethodTemplate"))
8  $MT.setAccess("public")
9  $MT.setType("$type")
10 $MT.setName("get$StringUtils.capitalizeFirstLetter($name)")
11 $MT.addEnd("return $name;")
12 $STOREOBJECT.addMethod($MT, $tagDic.getPrefix())
13 #end
```

*Listing 2.10: Code des Velocity-Scripts des Session-Transformers*

Erläuterungen zu Listing 2.10:

- *Zeile 6*: Hiermit wird ein neues Velocity-Makro angelegt, welches von jeder Stelle aus dem Transformer-Code aus aufrufbar ist.

- *Zeile 7*: Mit der Methode `$vdoclet.makeBean(…)` kann ein beliebiges Objekt in Velocity instanziert werden. Alle Variabeln in Velocity haben als Präfix ein ’$’-Zeichen.

- *Zeilen 8-11*: Hier wird das `MethodTemplate`-Objekt `$MT` mit Semantik gefüllt.

- *Zeile 12*: Schließlich wird das erstellte `MethodTemplate`-Objekt `$MT` dem `ClassTemplate`-Objekt `$STOREOBJECT` zugewiesen, dass heißt diese Methode ist jetzt Teil der von `$STOREOBJECT` repräsentierten Klasse.

### 2.6.4 Transformationsprozess

Die Reihenfolge der Anwendung der Transformer auf den MobCon-Code wird durch die Use-Before-/ Use-After-Listen in den Manifest-Dateien der Transformer bestimmt. Nachdem die Reihenfolge festgelegt wurde, werden voneinander unabhängige Transformer zu Gruppen zusammengefasst. Diese Transformer-Gruppen werden nacheinander abgearbeitet.

Die einzelnen Transformer einer Gruppe nehmen Manipulationen an jeweils leeren `ClassTemplate`-Objekten vor, welche allesamt parallel ausgeführt werden könnten, da keine Abhängigkeiten zwischen den Transformern einer Gruppe bestehen. Wenn alle Transformer einer Gruppe ihre Arbeit beendet haben, werden die entstanden `ClassTemplate`-Objekte gemäß ihrem Vereinigungsverhalten zu einem einzigen `ClassTemplate`-Objekt zusammengefügt. Dieses Vereinigungsverhalten kann entweder das Standardvereinigungsverhalten oder ein spezielles Vereinigungsverhalten sein, welches in einem von `MixTemplate` abgeleiteten Objekt definiert wird. Das entstandene `ClassTemplate`-Objekt wird an die nächste Transformer-Gruppe als Input weitergegeben und wiederum weiterverarbeitet und so weiter und so fort.

Der Transformationsprozess wird in Abbildung 2.5 veranschaulich. Die AST-CT-Objekte sind die `ClassTemplate`-Objekte zu Begin und am Ende des Transformationsprozesses. TG sind die einzelnen Transformer-Gruppen, die aus den umkreisten Transformern bestehen. Alle Transformer bekommen als Input den AST des von vDoclet aufbereiteten MobCon-Codes (App-Source). Als Output liefert jede Transformer-Gruppe ein `ClassTemplate`-Objekt (CT), welches an die nächste Gruppe weitergegeben wird.

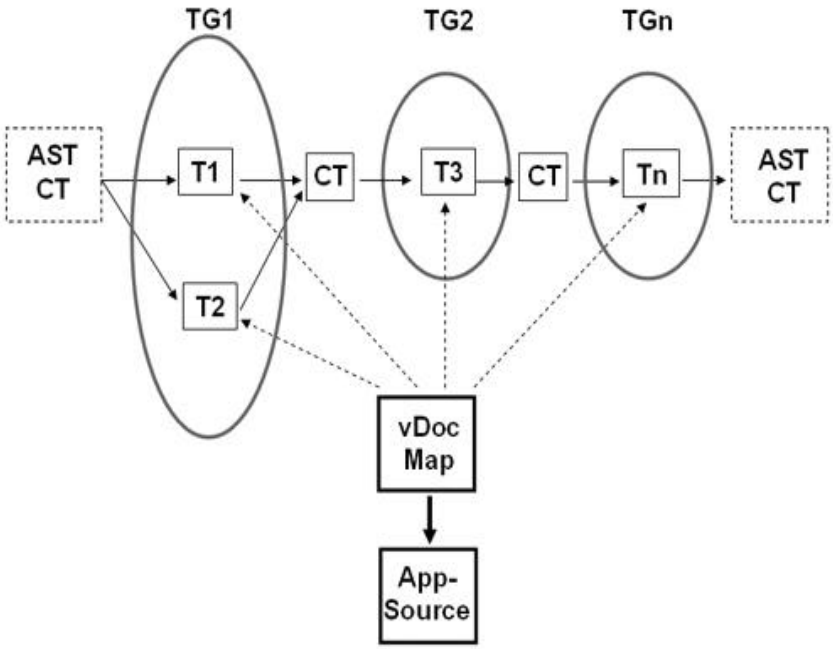


*Abbildung 2.5: Transformer Workflow*

### 2.6.5 Abhängigkeits-Datei (depend.xml)

Falls ein anderer Workflow des Transformationsprozesses bevorzugt wird, als der der sich aus den Manifest-Dateien ergibt, dann kann dies in einer separaten Datei festgelegt werden. In dieser Datei kann nicht nur der Workflow, sondern auch die Gruppierung von Transformern und das Vereinigungsverhalten der `ClassTemplate`-Objekte festgelegt werden. Näheres dazu in Abschnitt 5.4.

### 2.6.6 Server-Datei (Serverseitige Transformerlogik)

Diese Java-Source-Datei beinhaltet die serverseitige Transformerlogik dieses Transformers. Sie ist in Javas SDK 1.4.1 verfasst und wird auf die Serverseite kopiert, in welcher sie dann auch kompiliert wird. Hier sollte der größte Teil der speicher- oder rechenaufwendigen Aufgaben des Transformers zu finden sein.

## 2.7 MobCon-Code

Der MobCon-Code ist der mit Tags dekorierte Code, aus dem später die gesamte mobile Anwendung entsteht. Hier findet sich neben den dekorierten Objekten auch die eigentliche Anwendungslogik. Die Anwendungslogik sind meistens undekorierte Methoden im MobCon-Code.

Am besten kann der Aufbau des MobCon-Codes an einem Beispiel beschrieben werden. Dieser MobCon-Code würde „Hello World“ auf den Bildschirm eines mobilen Endgeräts ausgeben. Um zu verdeutlichen wie MobCon die Arbeit erleichtert, soll

hier darauf hingewiesen werden, dass der generierte Anwendungs-Code (Im Anhang B zu finden) über 100 Zeilen lang ist. Das heißt fünf Mal umfangreicher als der MobCon-Code.

***Beispiel: Code für „Hello World“***

```
1  import javax.microedition.midlet.*;
2  import javax.microedition.lcdui.*;
3
4  /**
5   *@scr
6   */
7  public class Test{
8     /**
9       * @scr.label "Test"
10      * @scr.firstDisplay
11      * @scr.exitButton
12      * @scr.textField textField
13      */
14     private Form form;
15
16     /**
17      * @scr.label "First Application"
18      * @scr.string "Hello World"
19      */
20     private TextField textField;
21 }
```

*Listing 2.11: Code für „Hello World“*

Welche Wirkung die Tags in diesem Beispiel haben, soll hier kurz unter Angabe der Zeilennummern aufgezeigt werden:

- *Zeile 5*: Dem Framework wird mitgeteilt, dass für diese Anwendung der Transformer mit dem Präfix `@scr` benutzt werden soll, also der Darstellungs-Manager. Dieser durchsucht den MobCon-Code nach Objekten, welche mit Tags aus seinem TagDic dekoriert sind. Sie kann man leicht erkennen, weil sie alle mit dem Präfix `@scr` beginnen. Falls der Transformer ein Tag-dekoriertes Objekt erkennt, wird er, abhängig von dem Tag, Veränderungen an seinem `ClassTemplate`-Objekt vornehmen.

- *Zeile 9*: Dem Tag `@scr.label` wird in einem `Form`-Objekt der String „Test“ als Parameter übergeben. Dies hat zur Folge, dass das `Form`-Objekt den Label „Test“ hat.

- *Zeile 10*: Der Tag `@scr.firstDisplay` bedeutet, dass dieses `Form`-Objekt der erste angezeigte Bildschirm in dieser Anwendung sein wird.

- *Zeile 11*: Ein Button zum Beenden der mobilen Anwendung wird diesem Bildschirm hinzugefügt.

- *Zeile 12*: Es können auch Objekte als Parameter der Tags übergeben werden, wie hier geschehen. Zu diesem Bildschirm wird ein `TextField`-Objekt mit Namen `textField` hinzugefügt.

- *Zeile 18*: Der lesbare Text dieses `TextField`-Objekts ist „Hello World“. In Zeile 12 wird dieses Objekt dem `Form`-Objekt zugewiesen. Dies bewirk, dass „Hello World“ auf dem Bildschirm zu lesen.

## 2.8 Ablauf der Verarbeitung des MobCon-Codes

Ausgehend von der Annahme, dass nur ein Transformer an der Verarbeitung des MobCon-Codes beteiligt ist, sollen hier die einzelnen Schritte der Transformation des MobCon-Codes erläutert werden. Diese Schritte können in Abbildung 2.6 nachvollzogen werden.

1. Der Anwendungs-Entwickler schreibt den MobCon-Code und dekoriert diesen mit Tags, gemäß den Angaben aus den TagDics der von ihm ausgewählten Transformer.

2. QDox parst den mit Tags dekorierten MobCon-Code und extrahiert die Klassen- / Methoden- und Feld-Informationen.

3. Diese Informationen werden an vDoclet übergeben, welches diese Struktur mit den vorhanden Tags in einem AST darstellt.

4. Der AST wird zusammen mit einem leeren `ClassTemplate`-Objekt in den Transformer (Velocity-Script) gemappt. Abhängig von den benutzten Tags werden verschiedene Veränderungen durch das Velocity-Script an dem `ClassTemplate`-Objekt vorgenommen, in welchem der Transformer seine generierten Methoden und Felder speichert.

5. Das resultierende `ClassTemplate`-Objekt wird als Java-Source-Code ausgegeben.

Die Transformation des MobCon-Codes durch mehrere Transformer unterscheidet sich dadurch, dass der Schritt 4. für jeden Transformer wiederholt wird, alle `ClassTemplate`-Objekte der Transformer vereinigt werden und erst nach Abarbeitung aller Transformer der 5. Schritt erfolgt.

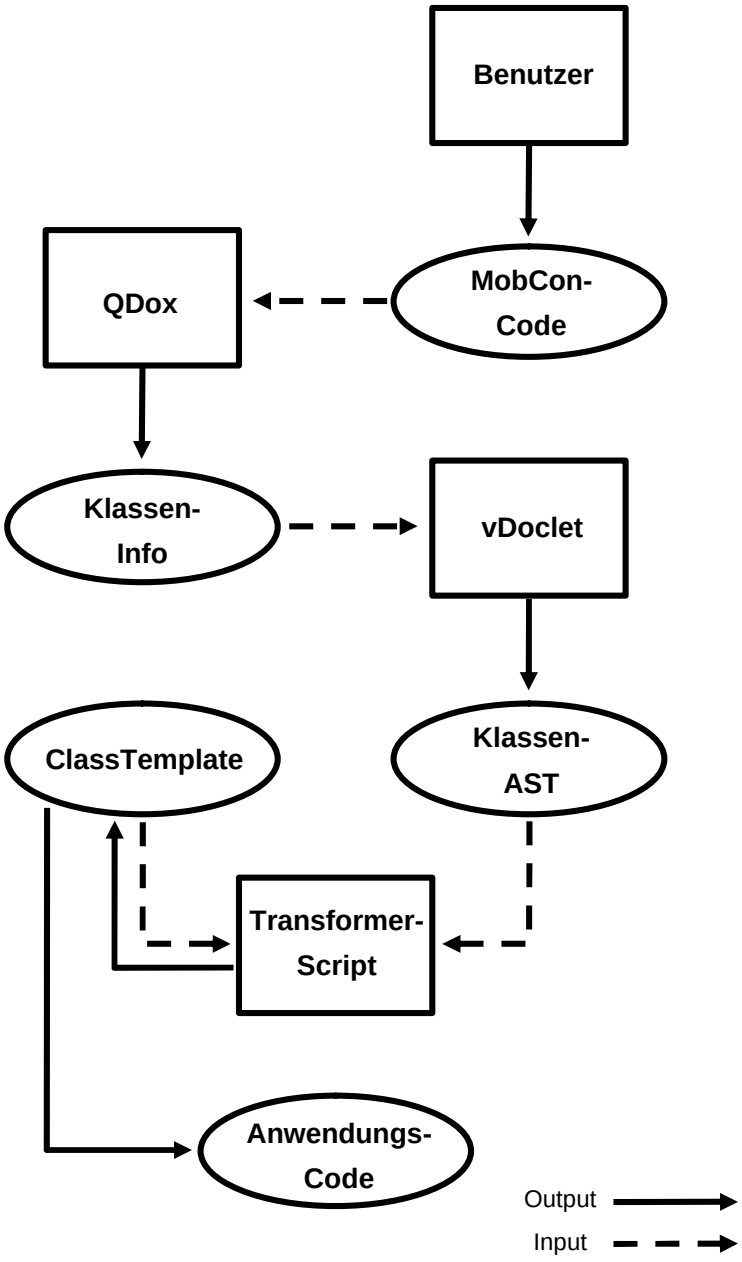


*Abbildung 2.6: Verarbeitung des MobCon-Codes*

***Beispiel: Parsen des MobCon-Codes und erstellen des Anwendungs-Codes***

Hier soll einmal exemplarisch der Weg eines dekorierten Objekts vom MobCon-Code, über einen Transformer zum Anwendungs-Code nachvollzogen werden.

In Zeile 8 des Listings 2.12 wird der `String id` mit einem Tag `@dp.access` dekoriert, welches die Eigenschaft „accessible“ modelliert. In Zeile 3 des Listings 2.13 durchsucht der Transformer alle Objekte des MobCon-Codes nach der Eigenschaft „accessible“. Diese Eigenschaft findet er bei dem `String id`, der dann an das Makro `dp_meth_get` (Listing 2.13, Zeile 5) übergeben wird.

Der Transformer generiert ein `MethodTemplate`-Objekt dessen Name sich aus der Code-Zeile `get$StringUtils.capitalizeFirstLetter($name)` ergibt. Der Funktionsaufruf `$StringUtils.capitalizeFirstLetter($name)` gibt einen String zurück, bei dem der erste Buchstabe des Namens des Feldes groß geschrieben wird. Zusammen mit dem Wort `get` resultiert der Methodenname `getId()`. Der Methoden-Körper wird mit der Code-Zeile `return id;` gefüllt. Dieses `MethodTemplate`-Objekt wird dann dem `ClassTemplate`-Objekt `$CT` hinzugefügt, welches am Ende des Transformationsprozesses ausgegeben wird. Der entstandene Anwendungs-Code ist in Listing 2.14 zu sehen.

MobCon-Code:

```
/**
 *@scr
 *@dp
 */
public class Test{
   . . .
   /**
     * @dp.access
     * @scr.store
     */
    private String id;
```

*Listing 2.12: MobCon-Code*

Transformer-Code:

```
#foreach ($field in $class.fields)
  #set($tag = false)
  #set($tag = $field.getTag($tagDic.getTag("accessible")))
  #if ($tag)
    #dp_meth_get ($field.type $field.name)
  #end
. . .
#macro(dp_meth_get $type $name)
  #set($MT = $vdoclet.makeBean("mobcon.ct.MethodTemplate"))
  $MT.setAccess("public")
  $MT.setType("$type")
  $MT.setName("get$StringUtils.capitalizeFirstLetter($name)")
  $MT.addEnd("return $name;")
  $CT.addMethod($MT, $tagDic.getPrefix())
#end
```

*Listing 2.13: Transformer-Code*

Generierter Anwendungs-Code:

```
1  . . .
2  public String getId(){
3  return id;
4  }
5  . . .
```

*Listing 2.14: Generierter Anwendungs-Code*

## 2.9 Template-Pattern in MobCon

In MobCon kann der generierte Code grob in zwei Teile eingeteilt werden. Zum einen gibt es den von den Transformern generierte Code und zum anderen den Code der vom Anwendungs-Entwickler undekoriert in den MobCon-Code eingefügt wurde und der in den Anwendungs-Code integriert wird.

In MobCon werden diese zwei Teile durch Anwendung des Template-Patterns voneinander getrennt. Dies geschieht indem der ausschließlich vom Transformer generierte Code in einer abstrakten Klasse gekapselt wird. In einer anderen Klasse wird der gesamte vom Anwendungs-Entwickler eingefügte Code gekapselt und diese Klasse von der abstrakten Klasse abgeleitet. Dadurch können eventuelle Compilermeldungen besser deren Ursprung nach, dem von MobCon generierten Code des Transformer-Entwicklers oder dem vom Anwendungs-Entwickler eingefügten Code zugeordnet werden.

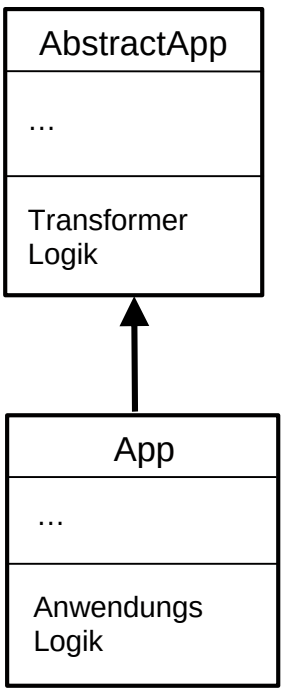


*Abbildung 2.7: Template-Pattern in MobCon*

## 2.10 Kommunikation

Mobile Endgeräte sind in ihrer Leistungsfähigkeit oft stark beschränkt. Darum ist die Kommunikation zwischen Server und Endgerät umso wichtiger, da dadurch der Großteil der Anwendungslogik auf den Server übertragen werden kann und dem Endgerät nur noch Resultate übermittelt werden. Somit wird das mobile Endgerät von unnötigem Rechenaufwand und Speicherverbrauch entlastet.

Deshalb ist das MobCon-Framework mit einer einfachen, auf Sockets basierenden, Kommunikations-Logik versehen. Durch sie können Informationen zwischen Server- und Clientseite ausgetauscht werden und sie ist flexibel genug, um den Ansprüchen einer mobilen Anwendung gerecht zu werden.

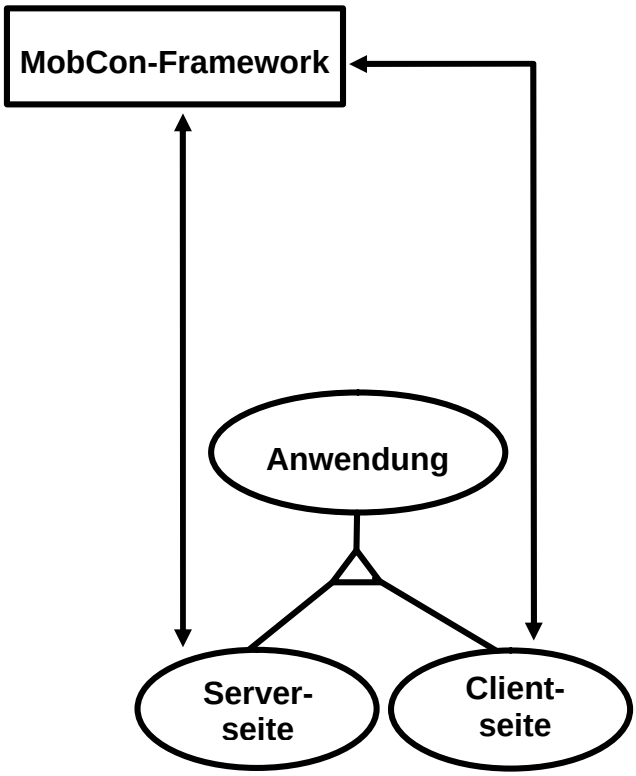


*Abbildung 2.8: Kommunikation über das MobCon-Framework*

MobCon hat einen Knotenpunkt der das Routen von Nachrichten zwischen der Server- und Clientseite übernimmt. Dieser Punkt ist der Connector Router, er empfängt die Nachrichten von mobilen Endgeräten und sendet sie zu dem serverseitigen Container für den die Nachricht bestimmt ist. Umgekehrt sendet der Connector Router die Antwort-Nachrichten des serverseitigen Transformers wieder zu dem Client zurück.

Die Anzahl der Kommunikationsmöglichkeiten für das mobile Endgerät wird dadurch sehr stark reduziert und ist nur über den Connector Router möglich. Dies hat zum Vorteil, dass der Client nicht direkt die Ports der serverseitigen Transformer ansprechen muss. Würde dies der Fall sein, so müsste schon bei der Änderung eines Ports die gesamte mobile Anwendung neu erstellt werden.

In Abbildung 2.9 wird das Routen der Nachricht an einem Beispiel vorgeführt. Die vom Client gesendete Nachricht spricht die Anwendung mit CID = 01 und dessen serverseitigen Transformer mit TID = 10, IID = 01 an. Auf der Serverseite ist nicht nur ein serverseitiger Transformer vorhanden, sondern mehrere, vielleicht auch von anderen Anwendungen. Die gesendete Nachricht wird vom Connector Router empfangen und an den für sie bestimmten serverseitigen Transformer weitergeleitet. Dieser antwortet dem Client wiederum über den Connector Router.

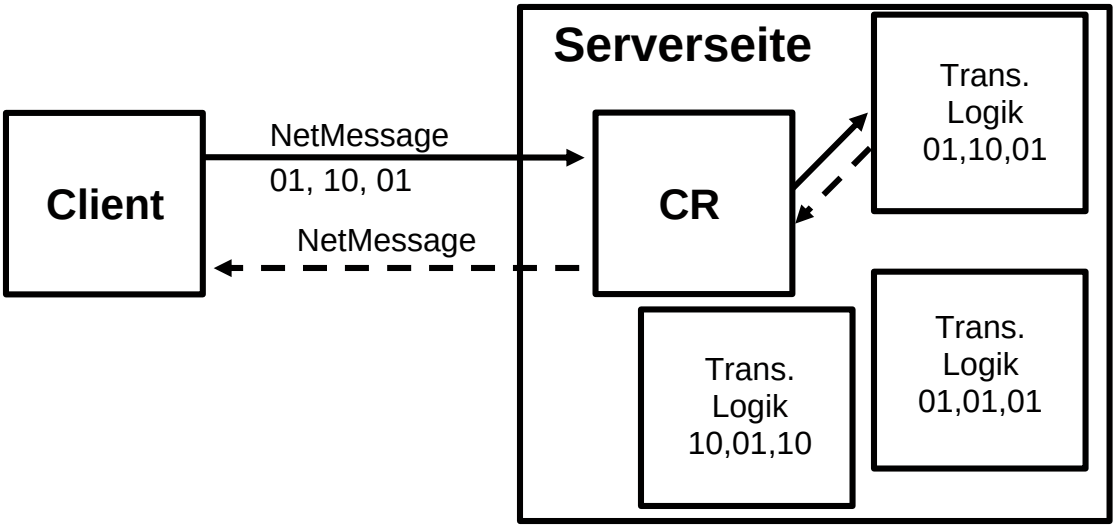


*Abbildung 2.9: Der Connector Router als Knotenpunkt*

Eine Nachricht wird in MobCon durch ein `NetMessage`-Objekt dargestellt. Nachrichten werden als Byte-Nachricht versendet und bestehen aus einem Kopf, der den Empfänger eindeutig Kennzeichnet, einem Datenteil und der Länge dieser Daten. In Abbildung 2.10 wird schematisch ein `NetMessage`-Objekt dargestellt.

Der Kopf des `NetMessage`-Objekts besteht aus vier Teilen:

- **CID**: Container ID, eindeutige 128-bit lange ID
- **TID**: Transformer-Typ ID, der vom Transformer implementierte technische Concern

- **IID**: Transformer-Instanz ID, Instanz des verwendeten Concerns
- **OID**: Operation ID, bestimmt welcher Code im serverseitigen Transformer ausgeführt wird

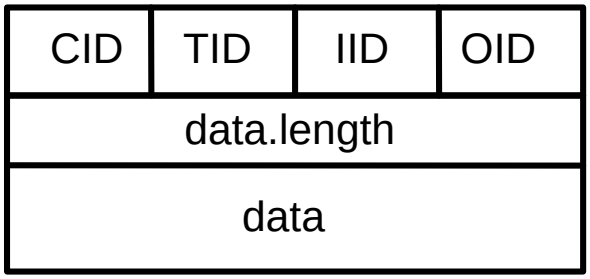


*Abbildung 2.10: Das NetMessage-Objekt*

Der serverseitige Code der Anwendungslogik wird als Transformer mit der Adresse „00.00“, also TID = IID = 00, angesehen und kann auf diese Weise adressiert werden.

Daten können in MobCon nur als Bytearrays versendet werden. Dies hat vor allem den Grund, dass in MIDP Daten nur in Bytearray-Form im nichtflüchtigen Speicher des mobilen Endgeräts abgelegt werden können. Durch diese Form wird erreicht, dass Nachrichten nach dem Empfangen und ohne vorherige Transformation gespeichert werden können.

Das Nachrichten dieser Form entsprechen, wird in MobCon dadurch gewährleistet, indem über das Netzwerk zu sendende oder im nichtflüchtigen Speicher abzulegende Objekte alle vom `Storeable`-Interface abgeleitet sein müssen. Dieses Interface besteht nur aus zwei Methoden, die eine wandelt das zugrunde liegende Objekt in ein Bytearray um (`object2record()`), während die andere aus diesem Bytearray wieder ein Objekt dieser Klasse rekonstruiert (`record2object(Byte[])`).

***Beispiel: Verwendung des NetMessage-Objekts***

Im folgenden Code wird eine Nachricht an den Transformer mit der Kennung 00.00 (also der serverseitige Teil der Anwendungslogik) in Anwendung CID gesendet. Der serverseitige Transformer soll die Operation mit OID = 1 ausführen und die Daten aus dem Objekt `DataBaseEntry`, welches das `Storeable`-Interface implementiert, verarbeiten.

```
1  public void storeEntry( DataBaseEntry dbe)
2  {
3    NetMessage message = new NetMessage();
4    message.setCID(CID);
5    message.setTID("00");
6    message.setIID("00");
7    message.setOID("1");
8    message.setData(dbe.object2record());
9
10   CTX.sendNetMessage(message);
11 }
```

*Listing 2.15: Verwendung des NetMessage-Objekts im Anwendungs-Code*

## 2.11 Deployment Descriptor

Der Deployment Descriptor ist eine XML-Datei, die ähnlich wie in EJB, zur Konfiguration der Anwendung verwendet wird. Hier können Angaben über sich häufig veränderte Eigenschaften der Anwendung gemacht werden, wie zum Beispiel über die IP-Adresse des Servers. Somit muss nicht der Quellcode der Anwendung nach diesen Eigenschaften durchsucht werden, weil sie an einer Stelle zusammengefasst werden.

***Beispiel: Deployment Descriptor***

Dieser Deployment Descriptor beschreibt zwei Eigenschaften der Anwendung. Zum einen wird in Zeile 4 festgelegt, dass ein im Transformer-Code verwendbarer Schlüssel generiert wird und zum anderen wird in Zeile 8 die Adresse des Servers angegeben auf dem sich der serverseitige Transformer der mobilen Anwendung befindet.

```
1  <application>
2    <transformer>
3      <encryption>
4        <generatekey>true</generatekey>
5      </encryption>
6      <persistence>
7        <server>
8          <ip>socket://localhost:1079</ip>
9        </server>
10     </persistence>
11   </transformer>
12 </application>
```

*Listing 2.16: Deployment Descriptor*

## 2.12 Ausblick

Im Folgenden soll ein Ausblick auf eventuelle Verbesserungsvorschläge oder Ideen zur Weiterentwicklung des Frameworks gegeben werden, die während der Bearbeitung dieser Diplomarbeit entstanden sind.

MobCon ist ein lauffähiger Prototyp eines Container-Frameworks für mobile Anwendungen. Es wurde gezeigt, dass eine solche Strukturierung der Anwendungs-Entwicklung mit Erfolg durchgeführt werden kann und ein viel versprechender Ansatz ist. Um diesen Ansatz zu komplettieren sollten aber noch weitere Transformer entwickelt werden, wie zum Beispiel Transformer die Sicherheit oder Authentifizierung implementieren. Die zur Zeit primitive Kommunikation zwischen Client- und Serverseite müsste auch nochmals überdacht und verbessert werden, um eventuell „sichere“ Nachrichten versenden zu können oder Nachrichten in einer Queue auf der Serverseite abzuarbeiten. Im Allgemeinen kann an eine Portierung des MobCon-Frameworks nach einer anderen Programmierumgebung als MIDP gedacht werden. Dies hätte zur Folge, dass neue Transformer für diese neue Sprache entwickelt werden und Teile des Frameworks umgeschrieben werden müssten.

Eine andere Idee wäre die Aufteilung der Rolle des Transformer-Entwicklers in zwei neue Rollen, die des TagDic-Entwicklers und die des Transformerlogik-Entwicklers. Der TagDic-Entwickler würde sich Gedanken über die vom Anwendungs-Entwickler benötigten Eigenschaften des Concerns machen und diese in einer Plattformunabhängigen Sprache entwickeln, während der Transformerlogik-Entwickler die Transformerlogik für die verschiedenen Platformern implementiert. Dies hätte den Vorteil, dass die Eigenschaften des Transformers somit für alle Plattformen die gleichen wären und der MobCon-Code beim Wechseln zu einer anderen Plattform nicht geändert werden müsste.

# Kapitel 3

# Dokumentation für Framework-Entwickler

Folgendes Kapitel beinhaltet die Dokumentation, die ein Entwickler benötigt um Veränderungen an dem Framework durchzuführen. Neben einer kurzen Beschreibung der Funktionsweise der einzelnen Klassen des Frameworks, wird auch deren Bedeutung erläutert. Die Abschnitte sind nach den einzelnen Packages des MobCon-Frameworks gegliedert und alphabetisch geordnet.

Um effizienter ihre Aufgaben erfüllen zu können ist es auch für Entwickler anderer Bereiche hilfreich einen Überblick über die Arbeitsweise des Frameworks zu bekommen.

Übersicht über dieses Kapitel:

- **Klassen in mobcon.app.***
  In diesem Package sind alle Klassen enthalten, die für den Ablauf des Transformationsprozesses benötigt werden.
- **Klassen in mobcon.ct.***
  Alle Klassen zur Erstellung von Template-Objekten sind hier zu finden.
- **Klassen in mobcon.message.***
  Die Klassen dieses Packages werden zur Kommunikation auf der Clientseite genutzt.
- **Klassen in mobcon.server.***
  Dieses Klassen werden zur Strukturierung und Adressierung des serverseitigen Containers benötigt.

- **Klassen in mobcon.storeables.***
  Hier sind einige, von MobCon genutzte, Beispielimplementierungen des `Storeable`-Interfaces zusammengefasst.

- **Klassen in mobcon.util.***
  Ausschließlich von MobCon benötigte Hilfsklassen werden in diesem Abschnitt vorgestellt.

- **Von MobCon erzeugte Klassen**
  Diese Klassen werden im Zuge des Transformationsprozesses von MobCon erstellt.

- **Änderungen an QDox und vDoclet**
  Fasst kurz die Änderungen an den Tools QDox und vDoclet zusammen, die nötig waren um sie auf MobCon abzustimmen.

- **Verzeichnisstruktur**
  In diesem Abschnitt wird die Verzeichnisstruktur des Frameworks vorgestellt.

- **Ant-Aufrufe**
  Diese Aufrufe werden zur Erstellung des Frameworks benutzt.

## 3.1 Klassen in mobcon.app.*

### Control

In dieser Klasse wird der gesamte Ablauf des Transformationsprozesses kontrolliert. Der Workflow, wie in 2.8 beschrieben, wird von dieser Klasse aus gesteuert.

Die einzelnen Schritte vom MobCon-Code bis zum Anwendungs-Code sollen hier einmal aus der Sicht des Framework-Entwicklers wiedergegeben werden:

- Die vDoclet-Engine wird gestartet.

- Die Velocity-Engine wird gestartet und deren Kontext initialisiert.

- Die Container-ID (CID) der Anwendung wird mit Hilfe der `RandomGUID`-Klasse generiert.

- Der Transformations-Manager TransMan wird initialisiert, alle Transformer im Transformer-Verzeichnis der Anwendung werden ausgelesen und die Reihenfolge deren Benutzung festgelegt.

- Der Deployment Descriptor wird ausgelesen und dessen Eigenschaften an den Velocity-Context weitergegeben.
- Vorhandene zusätzliche Client- und/oder Serverdateien der Anwendung werden in die jeweiligen Verzeichnisse kopiert.
- Die drei Klassen `MU, CTX` und `MobConMain` werden generiert.
- Der MobCon-Code wird eingelesen und die adressierten Transformer werden gestartet.
- Der Source-Code der generierten Klassen wird in die jeweiligen Anwendungsverzeichnisse ausgegeben.

**DepBuilder**

Wird von der Klasse `Control` aufgerufen um den Abhängigkeitsbaum der Transformer zu erstellen. Diese Klasse liest die Use-Before- und Use-After-Listen aus den Transformer-Manifest-Dateien ein und legt die daraus resultierende Aufrufreihenfolge der Transformer fest. Transformer werden zu Gruppen zusammengefasst und der Abhängigkeitsbaum an `Control` zurückgegeben.

**DepDescriptorReader**

Parst den Deployment Descriptor und erlaubt den Zugriff auf dessen Elemente.

**DepFileReader**

Falls die Datei `depend.xml` im plugin-Verzeichnis der mobilen Anwendung zu finden ist, werden die Abhängigkeiten zwischen den Transformern nicht von der Klasse `DepBuilder` aus den Manifest-Dateien der Transformer erstellt, sondern aus dieser Datei mit Hilfe dieser Klasse gelesen. Der Aufbau der Datei `depend.xml` ist dem Abschnitt 5.4 zu entnehmen.

**TagDic**

Ist schon im Abschnitt 2.6.2 ausführlich erklärt worden. Diese Klasse wurde mit Hilfe von `java.util.PropertyResourceBundle` erstellt.

**TagDicParser**

Liest den Inhalt eines TagDics ein und generiert automatisch Dokumentation aus den dort eingefügten Tags. Details über den Aufbau der TagDic-Datei und welche Tags eingefügt werden können, ist in Abschnitt 4.2 nachzulesen.

**Transformer**

Liest die JAR-Datei des Transformers aus und speichert alle notwendigen Informationen aus der Manifest-Datei, wie die UseBefore / UseAfter-Listen, die TID, die IID, den Namen des Transformers, den Dateinamen der Velocity-Script-Datei, die TagDic-Datei des Transformers und den Dateinamen der serverseitige Transformer-Logik. Diese Klasse stellt gleichzeitig den Zugriff auf die Veloctiy-Script-Datei des Transformers zur Verfügung. Außerdem wird in dieser Klasse auch das, aus der Anwendung dieses Transformers auf den MobCon-Code resultierende, `ClassTemplate`-Objekt gehalten.

**TransformerGroup**

Klasse um eine Transformer-Gruppe in MobCon zu modellieren.

**TransMan**

Erzeugt und verwaltet den Workflow des Transformationsprozesses. Das heißt diese Klasse erstellt den Abhängigkeitsbaum anhand der Manifest-Dateien der Transformer oder, falls vorhanden, anhand der `depend.xml`-Datei. Sie iteriert in der richtigen Reihenfolge über die Transformer-Gruppen beziehungsweise Transformer und gibt den Inhalt des resultierenden `ClassTemplate`-Objekts als eine Java-Source-Datei aus.

**VelocityNullLogEvent**

Diese Klasse wird benötigt um in Velocity unnötiges loggen des NullLogEvents zu verhindern.

## 3.2 Klassen in mobcon.ct.*

Die Klassen `ClassTemplate`, `MethodTemplate`, `FieldTemplate`, `MixTemplate`, `DefaultMixTemplate`, `TaggedElement` und `TaggedCode` wurden schon in Abschnitt 2.5 beschrieben. Transformer-Entwickler sollten sich den Abschnitt 4.1.3 durchlesen, in dem die Benutzung dieser Klassen detailliert beschrieben wird.

## 3.3 Klassen in mobcon.message.*

### MessageHandler

Diese von `java.lang.Thread` abgeleitete Klasse ist nötig damit der Client nicht von einem Netzwerk-Aufruf blockiert wird und somit seine anderen Aktivitäten fortsetzen kann. Die Klasse besteht aus zwei Methoden, `sendNetMessage(NetMessage)` und `getNetMessage(NetMessage)`. Erstere sendet eine Nachricht, während die Zweite eine Nachricht sendet und anschließend auf eine Antwort des Servers in Form eines `NetMessage`-Objekts wartet.

### NetMessage

Modelliert eine Nachricht in MobCon. Diese Klasse implementiert das `mobcon.message.Storeable` Interface und hat damit auch die `object2record`- und `record2object`-Methoden, die die Klasse in ein Bytearray umwandeln bzw. aus einem Bytearray ein Objekt dieser Klasse erzeugen kann. Der Aufbau eines `NetMessage`-Objekts wird detailliert in Abschnitt 2.10 wiedergegeben.

### Storeable

Dieses Interface besteht aus nur zwei Methoden, der `object2record`- und `record2object`-Methode. Klassen die dieses Interface implementieren müssen gewährleisten, ihren Inhalt in ein Bytearray umwandeln und umgekehrt aus einem Bytearray ein Objekt dieser Klasse rekonstruieren zu können. Alle Klassen, die über das Netzwerk gesendet oder in dem nichtflüchtigen Speicher des Clients abgelegt werden sollen, müssen dieses Interface implementieren, da sowohl die Klasse `NetMessage` als auch Javas J2ME MIDP Bytearrays als Träger der Daten erwartet.

## 3.4 Klassen in mobcon.server.*

### ContainerApp

Diese Klasse modelliert die serverseitigen Transformer. Hier können serverseitige Transformer hinzugefügt werden und auf diese anhand der TID und IID zugegriffen werden.

### CR

Der Connector Router auf der Serverseite verarbeitet die empfangenen Nachrichten und leitet sie zu dem richtigen serverseitigen Transformer weiter. Zuerst wird die richtige Anwendung anhand der CID gesucht, die als `ContainerApp`-Objekt

zurückgegeben wird. Anschließend wird, abhängig von der TID und IID, der richtige serverseitige Transformer als `TransformerApp`-Objekt aus dem `ContainerApp`-Objekt ausgewählt. Diesem wird dann die Nachricht als Bytearray, die OID und die Socket-Verbindung übergeben. Danach ist das `TransformerApp`-Objekt für die weitere Verarbeitung der Nachricht verantwortlich.

#### TransformerApp

Alle serverseitigen Transformer müssen sich von dieser Klasse ableiten. Sie ist alleine für das Verarbeiten und Rücksenden von Nachrichten (`NetMessage`-Objekte) verantwortlich. Wichtigste Methode in dieser Klasse ist die `execute(int, byte[], Socket)`-Methode. Diese wird von der `ContainerApp`-Klasse aufgerufen, wenn eine Nachricht eintrifft die für diesen Transformer bestimmt ist.

## 3.5 Klassen in mobcon.storeables.*

Dieses Package beinhaltet einige von MobCon benötigte Implementierungen des `Storeable`-Interfaces. Klassen die das `Storeable`-Interface implementieren, gewährleisten, dass sie über das Netzwerk übertragen bzw. in dem nichtflüchtigen Speicher des Clients abgelegt werden können. Diese Klassen halten Objekte, die sie dann in Bytearrays umwandeln, oder aus Bytearrays ihre ursprünglich gehaltenen Objekte rekonstruieren.

#### StoreableStringArray

Diese Klasse hält ein Stringarray.

#### StoreableStringData

Diese Klasse hält ein Bytearray und einen String. Sie wird verwendet um den Daten in Bytearray-Form noch eine Identifikation oder Auszeichnung durch den String zuzuordnen.

#### StoreableStringHashtable

Diese Klasse hält ein `Hashtable`, dessen Schlüssel und Einträge ausschließlich Strings sind.

## 3.6 Klassen in mobcon.util.*

**ByteArray**

Wandelt einen String hexadezimaler Zeichen in ein Bytearray um und umgekehrt. Wird benutzt um die IDs CID, TID und IID Platz sparend als Bytearray über das Netzwerk zu senden und umgekehrt diese für den Benutzer besser lesbar angeben zu können. Diese Klasse wurde von dem Open-Source-Projekt GenJava [15] übernommen.

**JadCreator**

Diese Klasse generiert die von Javas J2ME MIDP geforderte JAD-Datei und fügt sie der Anwendung hinzu.

**RandomGUID**

Generiert einen 128-Bit langen, eindeutigen Schlüssel. Diese Klasse wird von MobCon benutzt um die CID einer mobilen Anwendung zu generieren oder um eine eindeutige Session-ID für den Session-Transformer zu erstellen. Diese Klasse wurde als Open-Source von Java Exchange [16] übernommen.

## 3.7 Von MobCon erzeugte Klassen

Diese Klassen werden im Laufe des Transformationsprozesses von MobCon erzeugt und der Client- oder Serverseite der Anwendung zugewiesen. Welcher Seite diese Klassen im Einzelnen zugewiesen werden steht hinter dem Namen in Klammern.

**CTX (Client)**

Der Containterkontext ist verantwortlich für das Senden und Empfangen von Nachrichten auf der Clientseite und leitet sie an die Communication Manager Unit (MU) weiter.

**MobConMain (Server)**

Der Einstiegspunkt des serverseitigen Containers. Dieses Objekt muss auf der Serverseite instanziert und die Socket-Verbindungen ihm übergeben werden, damit MobCon ordnungsgemäß arbeiten kann.

**MU (Client)**

Communication Manager Unit. Beinhaltet die Serveradresse aus dem Deployment Descriptor und ruft den `MessageHandler` auf, um Nachrichten vom Client zu senden.

## 3.8 Änderungen an QDox und vDoclet

Die beiden Tools QDox und vDoclet waren nicht auf die speziellen Bedürfnisse von MobCon ausgelegt, deshalb mussten leichte Veränderungen an dem Code beider Tools vorgenommen werden. Standardmäßig können mit QDox und vDoclet keine Informationen über den Inhalt einer Methode, Array-Dimensionen von Feldern und importierte Klassen aus dem MobCon-Code extrahiert werden. Dies musste geändert werden.

Also musste in QDox der Parser insofern Abgewandelt werden, als dass er die Inhalte von Methoden, von der Klasse importierte Java-Packages und Array-Dimensionen von Feldern einlesen kann. Die Änderungen in vDoclet betreffen vor allem die Integration dieser neu gewonnenen Klassen-Informationen in das `DocInfo`-Objekt. Dieses Objekt wird von den Transformern benutzt, um die Struktur des MobCon-Codes einzulesen und abzufragen.

## 3.9 Verzeichnisstruktur

`framework`
: Root-Verzeichnis des Frameworks

`framework/changed-src`
: Sourcen der geänderten Bibliotheken von QDox und vDoclet

`framework/classes`
: Kompilierte Sourcen des Frameworks

`framework/lib`
: Zum Kompilieren benötigte Bibliotheken

`framework/out`
: JAR-Datei der kompilierten Framework Sourcen

`framework/src`
: Java-Sourcen des MobCon-Frameworks

## 3.10 Ant-Aufrufe

Alle Aufrufe können vom MobCon Root-Verzeichnis aus ausgeführt werden. Bevor dies jedoch gemacht wird, sollte die Datei `default.properties` entsprechend angepasst werden.

### 3.10.1 Die Befehle

`ant transformer`

Packt die Transformer-Dateien jedes Transformers, also das Velocity-Script, das TagDic, eventuelle Server-Sourcen und das Manifest in jeweils eine JAR-Datei (Transformer-Modul). Außerdem werden die JAR-Dateien in die zurzeit verwendete Anwendung kopiert und die TagDic-Dokumentation generiert.

`ant framework.compile`
: Kompilieren des Frameworks

`ant framework.jar`
: Packen des kompilierten Codes

`ant framework`
: Kompiliert und packt den Code

# Kapitel 4

# Dokumentation für Transformer-Entwickler

Dieses Kapitel soll einen Transformer-Entwickler dazu befähigen, eigenständig Transformer-Module zu erstellen. Es soll allerdings keine Einführung in Velocity oder vDoclet sein, deshalb sollten eventuell benötigte Grundlagen schon vorhanden sein oder nachgearbeitet werden, da deren Einführung den Rahmen dieser Diplomarbeit sprengen würde.

Der Transformer-Entwickler sollte am meisten Überblick über die Bereiche der anderen MobCon-Entwickler haben, da er doch an der Schnittstelle zwischen Framework und Anwendung arbeitet. Deshalb sollte ein Transformer-Entwickler, bevor er anfängt Transformer-Module zu erstellen, einen Blick auf die von MobCon standardmäßig mitgelieferten Transformer werfen und überlegen wie er Anwendungs-Entwicklern bei der Erstellung mobiler Anwendungen helfen könnte.

Übersicht über dieses Kapitel:

- **Benutzung der Transformer-Skripte (*.vm)**
  Übersicht über die Transformer-Skripte, zu Velocity hinzugefügte Variabeln und der Benutzung der Template-Objekte.

- **Benutzen des TagDics (*.tag)**
  Erklären der Einträge und der Benutzung des TagDics eines Transformers, sowie der Konventionen zur besseren Lesbarkeit der TagDics für den Anwendungs-Entwickler.

- **Erstellen des Transformer-Manifests (Manifest.mf)**
  Auflistung und Beschreibung der Einträge der Manifest-Datei im Transformer-Modul.

- **Konventionen und Standard-Methoden von MobCon**
  Die von MobCon und MIDlet benutzten Standard-Methoden werden aufgelistet und Konventionen zur Entwicklung mit MobCon vorgestellt.

- **Verzeichnisstruktur**
  In diesem Abschnitt wird die Verzeichnisstruktur der vom Transformer-Entwickler verwendeten Umgebung vorgestellt.

- **Ant-Aufrufe**
  Diese Aufrufe können zur Erstellung der Transformer benutzt werden.

## 4.1 Benutzung der Transformer-Skripte (*.vm)

### 4.1.1 Durch MobCon in die Velocity-Engine eingefügte Variabeln

***$CT*** ist das Standard `ClassTemplate`, welches zum Transformer gehört und das später ausgegeben, bzw. mit dem `ClassTemplate` aus dem vorherigen Transformer vereinigt wird.

***$MAINCT*** ist das `ClassTemplate`, welches schon alle vorherig bearbeiteten `ClassTemplate`-Objekte vereinigt. Sozusagen der aktuelle Stand des Transformationsprozesses.

***$transMan*** ist der während des Transformationsprozesses verwendete Transformationsmanager (`TransMan`-Klasse).

***$transformer*** ist die `Transformer`-Klasse, die zurzeit im Transformationsprozess ausgeführt wird. Diese Variabel wird zum Beispiel dazu genutzt, um das `TagDic`-Objekt des `Transformer`-Objekts zu erhalten (`$transformer.getTagDic()`) oder zusätzlich generierte `ClassTemplate`-Objekte dort abzulegen (`$transformer.addOtherClassTemplate(…)`), die später auch ausgegeben werden.

***$docInfo*** ist das `doclet.docinfo.DocInfo` Objekt von vDoclet.

***$vdoclet*** ist das `vdoclet.Generator`-Objekt von vDoclet. Wird vor allem für das Erstellen neuer Objekte in Velocity benötigt. Dies geschieht mit dem Aufruf `$vdoclet.makeBean("classname")).`

***$class*** ist das `ClassInfo`-Objekt das von vDoclet generiert wird und den MobCon-Code modelliert. Zum Beispiel liefert `$class.fields` alle Felder des MobCon-Codes oder mit `$class.shortName` wird der unqualifizierte Name der Klasse zurückgegeben.

***$PREVERIFY*** gibt das `ArrayList`-Objekt zurück, welches die mit dem J2ME Wireless Toolkit Preverifier [12] zu überprüfenden Source-Dateinamen enthält. Der Preverifier stellt fest, ob die Java-Klassen mit J2ME MIDP kompatibel sind und liefert neue, an MIDP angepasste class-Dateien. Falls eine neue Client-Source generiert wird, ist es wichtig diese auch diesem Objekt hinzuzufügen. Normalerweise geschieht das automatisch und wird vom Framework übernommen.

***$MCM*** ist das, zu dieser mobilen Anwendung gehörende, `MobConMain`-Objekt.

***$CID*** ist die CID der Anwendung, die im Moment erstellt wird.

***$TID*** ist die TID des Transformers, der im Moment abgearbeitet wird.

***$IID*** ist die IID des Transformers, der im Moment abgearbeitet wird.

***$SERVERIP*** ist die Server-IP aus dem oder `null`, falls keine Server-IP unter dem Schlüssel `application/transformer/persistence/server/ip` im Deployment Descriptor angegeben wurde.

***$KEY*** ist der zur Verschlüsselung verwendete Schlüssel, falls dieser unter dem Schlüssel `application/transformer/encryption/generatekey` im Deployment Descriptor festgelegt wurde, `null` sonst.

### 4.1.2 Erzeugen neuer Objekte in Velocity

Mit `#set($object = $vdoclet.makeBean("…"))` kann ein neues Objekt erzeugt werden, auf das mit `$object` zugegriffen werden kann. Als Parameter wird dem Aufruf der voll qualifizierte Klassenname übergeben.

Die Klassennamen, die zur Erzeugung der Template-Objekte benutzt werden sind:

- ***mobcon.util.FieldTemplate***
- ***mobcon.util.MethodTemplate***
- ***mobcon.util.ClassTemplate***

### 4.1.3 Benutzung der Template-Objekte im Tranformer-Code

Da den Template-Objekten im Transformer-Code eine sehr große Bedeutung zukommt sollen deren Methoden hier einmal im Detail erläutert werden.

***Methoden von mobcon.util.FieldTemplate:***

```
public String getAccess()
public void setAccess(String tmpAccess)
```

Setzt den *Zugriffstyp* dieses Feldes, beziehungsweise gibt ihn zurück.

```
public String getType()
public void setType(String type)
```

Setzt den *Typ* dieses Feldes, beziehungsweise gibt ihn zurück.

```
public String getName()
public void setName(String name)
```

Setzt den *Name* dieses Feldes, beziehungsweise gibt ihn zurück.

```
public String getValue()
public void setValue(String value)
```

Setzt den *Wert* dieses Feldes, beziehungsweise gibt ihn zurück.

***Methoden von mobcon.util.MethodTemplate*:**

```
public String getAccess()
public void setAccess(String tmpAccess)
```

Setzt den *Zugriffstyp* dieser Methode, beziehungsweise gibt ihn zurück.

```
public String getType()
public void setType(String type)
```

Setzt den *Typ* dieser Methode, beziehungsweise gibt ihn zurück.

```
public String getName()
public void setName(String name)
```

Setzt den *Name* dieser Methode, beziehungsweise gibt ihn zurück.

```
public String getValue()
public void setValue(String value)
```

Setzt den *Wert* dieser Methode, beziehungsweise gibt ihn zurück.

```
public void setAbstract(boolean set)
public boolean getAbstract()
```

Setzt diese Methode *abstract*, beziehungsweise gibt `true` zurück, falls es eine *abstract* Methode ist.

```
public void addException(String exception)
```

Fügt dieser Methode eine *Exception* hinzu.

```
public MethodTemplate mixMethodTemplates(MethodTemplate mt2)
```

*Vereinigt* diese Methode mit der Methode `mt2`. Dazu wird das `MethodeTemplate`-Objekt aus der Klasse benutzt, welche diese Methode beinhaltet.

```
public void addParameter(FieldTemplate ft)
public String getArgs()
public ArrayList getParameters()
```

Fügt dieser Methode einen *Parameter* in Form eines `FieldTemplate`-Objekts hinzu, beziehungsweise gibt die Parameter als String oder als `ArrayList`-Objekt zurück.

```
public void addField(FieldTemplate ft)
```

Fügt dieser Methode ein *Feld* hinzu. Dieses Feld wird am Anfang der Methode deklariert und initialisiert.

```
public String getBegin()
public String getBody()
public String getEnd()
```

Gibt den eingefügten Code dieser Methode als String zurück. Der Code kann jeweils von den drei Teilen der Methode (Begin, Body, End) abgefragt werden.

```
public void addBegin(TaggedCode cs)
public void addBegin(String code)
public void addBegin(String code, String tag)
```

Fügt dieser Methode an der Stelle „Begin“ eine *Codezeile* hinzu. Der Code kann als `TaggedCode`-Objekt, als String oder als String mit einem Tag hinzugefügt werden. Intern wird aber jeder Code als `TaggedCode`-Objekt behandelt, das heißt der String wird in dieses Objekt umgewandelt, beziehungsweise noch ein Tag zu diesem Objekt hinzugefügt. Außerdem werden die eingefügten `TaggedCode`-Objekte automatisch mit dem Tag ***`begin`*** versehen.

```
public void addBody(TaggedCode cs)
public void addEnd(TaggedCode cs)
```

Fügt dieser Methode an der Stelle „Body“ oder „End“ eine *Codezeile* hinzu. Andere Methoden ähnlich zu denen von `addBegin()` existieren auch.

```
public Vector getAllSnippets()
```

Gibt einen Vector aller `TaggedCode`-Objekte aller Teile (Begin, Body, End) dieser Methode zurück.

```
public Vector getSnippets(String[] inTags)
public Vector getSnippets(String tag)
```

Gibt einen Vector aller `TaggedCode`-Objekte aller Teile (Begin, Body, End) dieser Methode zurück, die mit den entsprechenden Tag/s dekoriert sind.

```
public MethodTemplate cloneMethod()
```

Gibt einen exakten Clone dieser Methode zurück.

***Methoden von mobcon.util.ClassTemplate:***

```
public String getName()
public void setName(String tmpName)
```

Setzt den *Namen* dieser Klasse, beziehungsweise gibt ihn zurück.

```
public String getBaseClass()
public void setBaseClass(String tmpBaseClass)
```

Setzt den Namen der Klasse, die diese Klasse *erweitert*, beziehungsweise gibt diesen zurück.

```
public String getAccess()
public void setAccess(String tmpAccess)
```

Setzt den *Zugriffstyp* dieser Klasse, beziehungsweise gibt ihn zurück.

```
public void addInterface(String iName)
public Collection getInterfaces()
```

Fügt ein *Interface* welches diese Klasse implementiert hinzu, beziehungsweise gibt die implementierten Interfaces zurück.

```
public void addImport(String iName)
public Collection getImports()
```

Fügt den Namen einer *importierten Klasse* dieser Klasse hinzu, beziehungsweise gibt die Namen der importierten Klassen zurück.

```
public void addMethod(MethodTemplate mt)
public Collection getMethods()
```

Fügt dieser Klasse eine *Methode* hinzu, beziehungsweise gibt die Methoden der Klasse zurück.

```
public void addField(FieldTemplate ft)
public Collection getFields()
```

Fügt dieser Klasse ein *Feld* hinzu, beziehungsweise gibt die Felder der Klasse zurück.

```
public void setMixTemplate(MixTemplate mt)
public MixTemplate getMixTemplate()
```

Setzt das *Vereinigungsverhalten* dieser Klasse, beziehungsweise gibt es zurück.

```
public ClassTemplate mixClassTemplates(ClassTemplate ct2)
public ClassTemplate mixClassTemplates(ClassTemplate ct1,
ClassTemplate ct2)
```

Vereinigt diese Klasse mit der Klasse `ct2`, beziehungsweise vereinigt zwei Klassen miteinander. Dabei wird das im `MixTemplate`-Objekt der Klasse definierte Vereinigungsverhalten benutzt.

***Methoden von mobcon.util.TaggedCode:***

```
public String getName()
public void setName(String tmpName)
```

Setzt den *Namen* dieser Klasse, beziehungsweise gibt ihn zurück.

```
public String getCode()
public void setCode(String in)
```

Setzt den Inhalt der *Codezeile* dieses Objekts, beziehungsweise gibt sie zurück.

```
public void addCode(String in)
```

Hängt an die Codezeile dieses Objekts, einen weiteren Teil an.

***Methoden von mobcon.util.TaggedElement:***

Die Klassen `MethodTemplate`, `FieldTemplate` und `TaggedCode` erweitern alle diese Klasse. Damit stehen ihnen natürlich auch folgende Methoden zur Verfügung.

```
public Vector getTags()
public void setTag(String in)
public boolean getTag(String in)
public void removeTag(String in)
```

Setzt oder entfernt einen *Tag* dieses Objekts, gibt true zurück falls dieser gesetzt ist oder gibt alle Tags dieses Objekts zurück.

```
public void setVisible()
public void setInvisible()
public boolean isVisible()
```

Falls dieses Objekt `visible` gesetzt ist, dann wird es in der Funktion `toString()` ausgegeben.

```
public String toString()
```

Gibt den String zurück, der dieses Objekt im Anwendungs-Code repräsentiert.

## 4.2 Benutzen des TagDics (*.tag)

Die TagDics werden jedem Transformer-Modul in Form einer Datei mit der Endung `*.tag` beigefügt. Sie hält durch „=“ getrennte Schlüssel-/Wertpaare, wobei die Schlüssel die Eigenschaften des vom Transformer implementierten technischen Concerns sind und die Werte die tatsächlich benutzten Tags im MobCon-Code. Auf diese Weise können die im MobCon-Code benutzten Tags frei definiert werden. Besonderer Bedeutung wird im TagDic dem Schlüssel „PREFIX“ zugewiesen. Dieser muss im MobCon-Code allen im TagDic benutzten Werten vorangestellt werden. Um zum Beispiel den Tag *`access`* des Data-Persistence-Transformers im MobCon-Code zu verwenden, muss der PREFIX des Transformers bekannt sein, hier *`@dp.`* Daraus resultiert der Tag *`@dp.access`*, der den zu dekorierenden Objekten im MobCon-Code vorangestellt werden muss.

Die Veränderungen, welcher der Transformer am Anwendungs-Code vornimmt, sollten vom Transformer-Entwickler im TagDic dokumentiert werden. Dadurch kann der Anwendungs-Entwickler leichter die erstellten Transformer nutzen und weiß auch welche Konsequenzen die Benutzung hat.

Die Dokumentation geschieht in einer standardisierten Art und Weise. Allen Einträgen im TagDic wird eine ***globale Beschreibung*** der Veränderungen oder Wirkungen als Kommentar (vorausgehendes Zeichen „#“) vorangestellt. Folgende Tags werden zur Dokumentation benutzt:

- *`@description`*: Übersicht über den Zweck und/oder die Arbeitsweise dieses Transformers.
- *`@addedFields`*: Alle Felder die, außer den schon getagten Feldern, in den Anwendungs-Code eingefügt werden.
- *`@addedMethods`*: Alle Methoden die dem Anwendungs-Code hinzugefügt werden.

- *@changedMethods*: Alle Standard-Methoden des Anwendungs-Codes, die verändert werden.

- *@serverOps*: Alle Operationen (OID) auf der Serverseite des Transformers und deren Wirkung. Die Operationen können über Nachrichten benutzt werden.

In das TagDic wird ***jedem Schlüssel-/Wertpaar*** folgende, als Kommentar (vorausgehendes Zeichen „#“) eingefügte, Tags vorangestellt:

- *@effects*: Übersicht über die Veränderungen oder Wirkungen, die durch die Benutzung dieses Tags im Anwendungs-Code hervorgerufen werden.

- *@parameter*: Typ des an den Tag übergebenen Parameters.

- *@classTag*: Falls dieser Tag im MobCon-Code der Klasse und nicht einem Objekt vorangestellt werden soll.

Auf jeden Fall sollten vor der Erstellung eines neuen Transformers die schon implementierten Transformer angesehen werden, um einen Überblick über deren Struktur zu bekommen.

***Beispiel: Auszug aus dem TagDic des Session-Transformers***

```
# @description =
# Stores to a session related data on the server.
# The mobileID is used as sessionID.

# @addedMethods =
# - Void storeContext()
# Stores the Hashtable context on server-side
# - Void retrieveContext()
# Gets the Hashtable context from server-side

# @changedMethods =
# - Void startApp()
# To begin: Gets the session-context from server,
# . . .

# @serverOps =
# 1
# Data from client: StoreableStringHashtable ssh
# Store Hastbale context on server.
# 2
# . . .
```

```
22
23 PREFIX = @ses
24
25 # @effects =
26 # Storeable-Objects can be added to session.
27 # They will be initialized in the startApp() method with the
28 # value of the last saved session.
29 # @parameter =
30 # Storeable
31 addToSession = addToSession
```

*Listing 4.1: TagDic des Session-Transformers*

## 4.3 Erstellen des Transformer-Manifests (Manifest.mf)

In dem Manifest des Transformers werden folgende Einträge erwartet:

- **Transformer-Name**: Der Umgangssprachliche Name des Transformers. Wird nicht zur Identifikation benutzt, sondern nur zur besseren Lesbarkeit für den Benutzer.

- **TID(Transformer-Type-ID)**: Der Entwickler muss vorher feststellen, ob schon ein Transformer dieses Typs entwickelt worden ist, falls ja sollte er (muss aber nicht) die gleiche Transformer-Type-ID verwenden. Falls nein, muss er diesem Transformer-Typ eine neue ID zuweisen. Erwartet wird eine zweistellige hexadezimale Zahl (wie z.B. 2A).

- **IID(Transformer-Type-Instance-ID)**: Falls mehrere Transformer des gleichen Typs in einer mobilen Anwendung benutzt werden, so müssen diese fortlaufend mit einem hexadezimalen Zahlenwert nummeriert werden. Begonnen wird bei 00, der letzte Wert ist AA.

- **File-Name**: Name der Datei im JAR, die den Transformer-Code (*.vm) enthält.

- **Tag-File**: Dateiname der TagDic-Datei (*.tag) im JAR.

- **Use-Before**: Eine durch “/” getrennte Liste von Transformer-IDs (TID.IID) die Transformer bezeichnen, welche *vor* diesem Transformer benutzt werden müssen.

- **Use-After**: Eine durch “/” getrennte Liste von Transformer-IDs (TID.IID) die Transformer bezeichnen, welche erst *nach* diesem Transformer benutzt werden dürfen.

- **Server-File**: Der Dateiname des Java-Source-Codes im JAR, welche die serverseitige Transformer-Logik hält.

## 4.4 Konventionen und Standard-Methoden von MobCon

All diese Methoden werden standardmäßig von MobCon oder von einem MIDlet eingefügt beziehungsweise benutzt:

- Der Konstruktor „`Abstract`“+„`Klassenname der Anwendung`“`()`.
- `void destroyApp(boolean unconditional)` wird bei Beendigung des MIDlets aufgerufen.
- `void pauseApp()` wird bei Pausierung des MIDlets aufgerufen.
- `void startApp()` wird beim Starten der Anwendung aufgerufen.

Konventionen, die bei der Erstellung des Transformers beachtet werden sollten:

- Code, der aufgrund eines dekorierten Felds generiert wird, sollte nicht direkt in den Anwendungs-Code eingefügt werden, sondern sollte in einer Methode „`call`“+„`Feldname`“ gekapselt werden.
- Code der Operationen auf dem Speicher des Clients oder des Servers durchführt, sollte durch „`retrieve…`“ `bzw.` „`store…`“ gekennzeichnet werden.

Die Nichtbeachtung dieser Konventionen, sollten keine schwerwiegenden Folgen nach sich ziehen. Zur besseren Lesbarkeit des Codes sollten sie aber respektiert werden.

## 4.5 Verzeichnisstruktur

`transformer`
Root-Verzeichnis für den Transformer-Entwickler.

`transformer/build`
Batch Befehle, die zur Erstellung der Transformer-Module benötigt werden.

`transformer/out`
JAR-Dateien (Transformer-Module) der Transformer.

`transformer/src`

In jeweils eigenen Verzeichnissen werden hier die Source-Dateien (*.vm, *.tag, *mf, *.java) jedes Transformers abgelegt.

`transformer/txt`

Die automatisch generierte Dokumentation der TagDic-Dateien der Transformer.

## 4.6 Ant-Aufrufe

Der Aufruf kann vom MobCon Root-Verzeichnis aus ausgeführt werden. Bevor dies jedoch gemacht wird, sollte die Datei `default.properties` entsprechend angepasst werden.

`ant transformer`

Packt die Transformer-Dateien jedes Transformers, also das Velocity-Script, die TagDic-Datei, eventuelle Server-Sourcen und das Manifest in jeweils eine JAR-Datei. Außerdem werden die JAR-Dateien in die zurzeit verwendete Anwendung (in `default.propertiers` angegeben) kopiert und die TagDic-Dokumentation generiert.

**Hinweis**: Zur Zeit muss für einen neuen Transformer per Hand ein `antcall` in das ant-Target `transformer` der Datei `build.xml` im MobCon Root-Verzeichnis eingefügt werden, falls dieser Transformer auch bei der Erstellung der JAR-Dateien berücksichtigt werden soll.

Der in `build.xml` eingefügte Code sieht für einen neuen Transformer mit dem Namen „NEU“ folgendermaßen aus:

```
1       <antcall target="transformer.call">
2       <param name="transformer.current"
3              value="NEU"/>
4       </antcall>
```

*Listing 4.2: Für neuen Transformer eingefügter Code in build.xml*

# Kapitel 5

# Dokumentation für Anwendungs-Entwickler

Der Anwendungs-Entwickler wird in MobCon vorwiegend die Aufgabe übernehmen den MobCon-Code zu erstellen und diesen mit Tags zu dekorieren. Dazu sollte er allerdings wissen, welche Funktionalität die Transformer haben. Nur durch diese Kenntnis und, damit verbunden, das Wissen welche Tags im TagDic der Transformer welche Funktion haben, kann effizient eine mobile Anwendung erstellt werden. Jeder Anwendungs-Entwickler sollte deswegen die im Anhang A dargestellte Transformer-Dokumentation durchgelesen haben, oder zumindest die Dokumentation der Transformer, welche seinen zu implementierenden technischen Concerns entspricht.

Übersicht über dieses Kapitel:

- **MobCon-Code**
  Wie der MobCon-Code dekoriert werden kann und undekorierter Code behandelt wird, ist in diesem Abschnitt nachzulesen.

- **Transformationsprozess aus Sicht des Anwendungs-Entwicklers**
  Wurde der Transformationsprozess schon aus Sicht des Frameworks beschrieben, so werden hier die wichtigsten Schritte des Transformationsprozesses aus der Sicht des Anwendungs-Entwicklers vorgestellt.

- **Starten einer Anwendung**
  In diesem Abschnitt wird beschrieben wie Anwendungen aus MobCon heraus gestartet werden können und welche Auswirkungen dies hat.

- **Die Datei “depend.xml”**
  Der Aufbau und die Nutzung dieser Datei zur Steuerung des Transformationsprozesses wird hier dokumentiert.

- **Verzeichnisstruktur**
  In diesem Abschnitt wird die Verzeichnisstruktur der Entwicklungsumgebung einer Anwendung vorgestellt.

- **Ant-Aufrufe**
  Diese Aufrufe können zur Erstellung, Reinigung oder zum Starten einer mobilen Anwendung benutzt werden.

## 5.1 MobCon-Code

Der MobCon-Code ist die mit Tags dekorierte Java-Source-Datei, die den Input des Transformationsprozesses der Transformer darstellt. Die im MobCon-Code benutzten Tags, werden im TagDic der verwendeten Transformer definiert.

Die Klassen-Tags, das sind die Tags die vor der Klassendefinition stehen, bestimmen welcher Transformer in den Transformationsprozess einbezogen wird. Welcher Tag dies für die einzelnen Transformer ist, ist Bestandteil der Transformerlogik und sollte der Transformer- oder TagDic-Dokumentation entnommen werden. Normalerweise wird aber der *`PREFIX`*-Tag des Transformers dazu verwendet.

Undekorierter Code (Methoden und Felder) wird wie er ist in den Anwendungs-Code der Clientseite eingefügt. Bereits vom Transformationsprozess generierte Methoden können durch einfaches Überschreiben, das heißt gleicher Name, gleicher Rückgabewert und gleiche Parameter, erweitert werden. Die neuen Methoden-Körper werden an das Ende der bereits generierten Methoden-Körper angehängt. Sollte die neue Methode aber mit dem Tag *`@begin`* dekoriert sein, so wird ihr Inhalt *vor* den bereits generierten Code eingefügt.

## 5.2 Transformationsprozess aus Sicht des Anwendungs-Entwicklers

Um ein besseres Verständnis für den Transformationsprozess zu bekommen, soll hier einmal auf die wichtigsten Schritte dieses Prozesses aus der Sicht des Anwendungs-Entwicklers eingegangen werden.

Folgende Schritte werden beim Starten des Transformationsprozesses durch den Ant-Aufruf `ant application` durchlaufen:

- Das Framework sucht nach Anwendungs-Sourcen im `src/mobile`-Verzeichnis.
- Danach werden die Transformer-Module (JAR-Dateien) im `lib/plugin`-Verzeichnis registriert, so dass sie später verwendet werden können.
- Die Transformer-JAR-Dateien werden ausgelesen und der Abhängigkeitsbaum der Transformer anhand ihrer Transformer-Manifest-Dateien erstellt.
- Falls ein anderer Abhängigkeitsbaum als der aus den Transformer-JAR-Dateien erstellte, verwendet werden soll, so kann dieser in der Datei `lib/plugin/depend.xml` angegeben werden.
- Der Deployment Descriptor im Verzeichnis `descriptor` wird eingelesen und die enthaltenen Informationen an das Framework weitergegeben.
- Die durch die Klassen-Tags des MobCon-Codes ausgewählten Transformer parsen den MobCon-Code.
- Der generierte Anwendungs-Code wird in die Verzeichnisse `classes/mobile` beziehungsweise `classes/server` geschrieben.
- Dateien in den Verzeichnissen `src/addsources/mobile` beziehungsweise `src/addsources/server` werden in die Verzeichnisse `classes/mobile` bzw. `classes/server` kopiert.
- Die Client-Sourcen aus dem Verzeichnis `classes/mobile` werden in das Verzeichnis `classes/mobile/bin` kompiliert, die Server-Sourcen aus `classes/server` in das Verzeichnis `classes/server/output`.
- Zusätzliche Client/Server-Ressourcen, das sind zum Beispiel Bilder, aus dem Verzeichnis `ressources/mobile` beziehungsweise `ressources/server` werden zu den kompilierten Dateien in die jeweiligen Verzeichnisse (`classes/mobile/bin` oder `classes/server/output`) kopiert.
- Die kompilierten Client-Sourcen aus `classes/mobile/bin` werden vom Preverifier des J2ME Wireless Toolkits, welcher überprüft ob der Code MIDP kompatibel ist, bearbeitet und zusammen mit den anderen Ressourcen in eine JAR-Datei gepackt. Schließlich wird die, von MIDP benötigte, JAD-Datei

generiert, welche zusammen mit der JAR-Datei im Verzeichnis `classes/mobile/app` abgelegt wird.

- Die kompilierten Server Sourcen werden zusammen mit den anderen Ressourcen in eine JAR-Datei gepackt, welche sich im Verzeichnis `classes/server/app` befindet.

## 5.3 Starten einer Anwendung

Wenn der Transformationsprozess erfolgreich beendet wurde, wird das nächste Ziel des Entwicklers das Starten der Anwendung sein. Die fertigen Dateien liegen für den Client im Verzeichnis `classes/mobile/app` und für den Server in `classes/server/app` bereit.

Durch `ant application.server.start` wird der Inhalt des Verzeichnisses `classes/server/app` nach `demo/server` kopiert und dort der Server `TestServer` gestartet, falls dieser von der mobilen Anwendung benötigt wird.

Danach sollte der Client mit `ant application.mobile.start` gestartet werden. Dies hat zur Folge, dass alle Dateien aus `classes/mobile/app` nach `demo/mobile` kopiert werden. Dann wird das WTK (Wireless Toolkit) gestartet, welches die generierte JAD-Datei einliest und schließlich die Anwendung in Form einer Handy-Emulation startet.

## 5.4 Die Datei “depend.xml”

Falls eine andere Abarbeitungsreihenfolge der Transformer, als die, die sich durch das Einlesen der Transformer-Manifest-Dateien ergeben hat, angewandt werden soll, so kann diese Reihenfolge in der Datei `depend.xml` angegeben werden.

Diese Datei muss in Verzeichnis `applications/xxx/lib/plugins` (Anwendungsname `xxx`) erstellt werden. Falls MobCon diese dort findet, wird die dort definierte Abarbeitungsreihenfolge angewandt.

Diese Datei setzt sich aus mehreren XML-Tags mit folgender Bedeutung zusammen:

- `flow`: Haupt-Tag, mit diesem Tag beginnt und endet die Datei.

- `group`: Nach diesem Tag wird eine Gruppe von Transformern, ohne Abhängigkeiten untereinander, angegeben.

- `transformer`: Kennzeichnet den Anfang einer Transformerbeschreibung. Folgen müssen die Tags `tid` und `iid`.

- `name`: Der Name des Transformers. Der Wert dieses Tags wird nur zur besseren Lesbarkeit angegeben. Die Benutzung ist freiwillig.

- `tid`: Transformer-TID.

- `iid`: Transformer-IID.

- `merger`: Das `ClassTemplate` des nachfolgenden Transformers wird auf eine spezielle Weise mit dem `ClassTemplate` des Transformers vereinigt, in dem dieses Tag definiert wurde. Das spezielle Vereinigungsverhalten wird in der Klasse spezifiziert, welche im Tag `class` angegeben wird.

- `class`: Der Name der Klasse, die für das Vereinigungsverhalten der Template-Objekte verantwortlich ist.

### 5.4.1 Aufbau der Datei

Die Transformer zwischen den `group`-Tags werden Einer nach dem Anderen abgearbeitet. Diese Transformer haben keine Abhängigkeiten untereinander, das heißt sie könnten theoretisch parallel abgearbeitet werden. Um einen Transformer zu definieren muss dessen TID und IID angegeben werden. Die Transformer werden nacheinander, gemäß der Reihenfolge in der Gruppe, ausgeführt.

Falls ein anderes Vereinigungsverhalten der `ClassTemplate`-Objekte der Transformer erwünscht wird, dann muss dies in einem `merger`-Tag definiert werden. Der Name der Klasse, welche das Vereinigungsverhalten spezifiziert, wird in dem `class`-Tag angegeben. Diese Klasse muss über den Klassenpfad erreichbar sein. Das neu definierte Vereinigungsverhalten gilt dann für den Transformer, in dem es definiert wurde, und den nachfolgenden Transformer. Falls ein anderes Verhalten für das Vereinigen von Gruppen erwünscht wird, so kann man dies in dem `group`-Tag auf gleiche Art und Weise definieren.

### 5.4.2 Beispiel

```
<flow>
 <group>
     <transformer>
           <name>New Transformer</name>
          <tid>01</tid>
          <iid>01</iid>
     </transformer>
 </group>

 <group>
     <transformer>
          <tid>02</tid>
```

```
13            <iid>01</iid>
14            <merger>
15                  <class>DepMerger</class>
16            </merger>
17      </transformer>
18      <transformer>
19            <tid>03</tid>
20            <iid>01</iid>
21      </transformer>
22  </group>
23 </flow>
```

*Listing 5.1: Abhängigkeits-Datei depend.xml*

Ergebnis des Transformationsprozesses, der anhand dieser Datei angegeben wurde:

- Als erstes wird der Transformer 01.01 abgearbeitet und das resultierende `ClassTemplate`-Objekt in der Gruppe gespeichert.
- Dann werden die Transformer 02.01 und 03.01 ausgeführt. Deren `ClassTemplate`-Objekte werden mit Hilfe der Klasse `DepMerger`, welche das `MixTemplate`-Interface implementiert, vereinigt. Das resultierende `ClassTemplate`-Objekt wird in der Gruppe gespeichert.
- Danach werden die beiden `ClassTemplate`-Objekte der Gruppen miteinander vereinigt, wobei das resultierende `ClassTemplate`-Objekt auch gleichzeitig das Endergebnis des Transformationsprozesses ist.

## 5.5 Verzeichnisstruktur

Mit `xxx` wird folgend der Name einer MobCon-Anwendung bezeichnet.

`applications/xxx/`
: Root-Verzeichnis der Anwendung `xxx`.

`applications/xxx/classes`
: Root-Verzeichnis des generierten Anwendungs-Codes.

`applications/xxx/classes/mobile`
: Generierter Anwendungs-Code des Clients.

`applications/xxx/classes/mobile/app`
: JAR- und JAD-Datei des Clients.

`applications/xxx/classes/mobile/bin`

Kompilierte Client-Sourcen.

`applications/xxx/classes/server`
Generierte Java-Sourcen des Servers.

`applications/xxx/classes/server/app`
JAR-Datei des Server.

`applications/xxx/classes/server/output`
Kompilierte Server-Sourcen.

`applications/xxx/demo`
Verzeichnis der Demo-Anwendung.

`applications/xxx/descriptor`
Deployment Descriptor der Anwendung.

`applications/xxx/lib`
Für die mobile Anwendung benötigte Bibliotheken.

`applications/xxx/lib/plugins`
Von der mobilen Anwendung benutzte Transformer-Module.

`applications/xxx/lib/plugins/MixTemplate`
MixTemplate-Objekte.

`applications/xxx/resources`
Zusätzliche Ressourcen für Client und Server (Bilder, Daten…).

`applications/xxx/src`
Ort der Anwendungs Sourcen.

`applications/xxx/src/addsources`
Zu kompilierende Anwendungs Sourcen.

`applications/xxx/src/mobile`
Ort des MobCon-Codes.

`applications/xxx/src/server`
Zu kompilierende Anwendungs Sourcen für den Server.

## 5.6 Ant-Aufrufe

Alle Aufrufe können vom MobCon Root-Verzeichnis aus ausgeführt werden. Bevor sollte jedoch die Datei `default.properties` entsprechend angepasst werden.

### 5.6.1 Die Datei “default.properties”

Diese Datei befindet sich im Root-Verzeichnis von MobCon, in welchem sich auch die von Ant benötigte Datei `build.xml` befindet. In der Datei `default.properties` kann die zur Zeit aktuelle mobile Anwendung ausgewählt werden und hier muss auch der Pfad zur Preverify- und Emulate-Anwendung des Java MIDP Wireless Toolkit angegeben werden.

Die Einträge haben folgende Bedeutung:

- `APPLICATIONDIR`
  Das Root-Verzeichnis der Anwendung.

- `APPLICATIONNAME`
  Der Klassenname der Anwendung. Sollte der Gleiche sein, wie der der im MobCon-Code benutzt wurde.

- `WTKPREVERIFY`
  Der Pfad zur `preverify.exe` des Java MIDP Wireless Toolkits.

- `WTKEMULATE`
  Der Pfad zur `emulate.exe` des Java MIDP Wireless Toolkits.

### 5.6.2 Die Befehle

`ant application`

Generiert und kompiliert den Anwendungs-Code der Client- und Serverseite. Dieser wird anschließend in eine JAR-Datei für jede Seite gepackt.

`ant cleanapp`

Löscht alle generierten und kompilierten Klassen.

`ant newapp`

Legt eine Verzeichnisstruktur für eine neue MobCon-Anwendung an. Das Verzeichnis muss nur noch umbenannt und der Deployment Descriptor eingefügt werden.

`ant application.mobile.start`

Startet die Clientseite einer mobilen Anwendung mit dem J2ME Wireless Toolkit.

`ant application.server.start`

Startet die Serverseite einer mobilen Anwendung.

# Anhang A

# Transformer im MobCon-Package

Im Folgenden werden die im Zuge dieser Diplomarbeit entwickelten Transformer beschrieben.

Die Beschreibung der Transformer beginnt mit einer kurzen Übersicht über dessen Arbeitsweise und Wirkung auf den Anwendungs-Code. Anschließend wird näher auf die zum Transformer gehörenden Tags eingegangen, wobei die Verwendung der Standardtags aus dem TagDic vorausgesetzt wird. Deren Wirkung wird beschrieben und mögliche Parameter des Tags angegeben. Diese Übersicht wird noch einmal nach den Objekten im MobCon-Code unterteilt, die mit den spezifischen Tags dekoriert werden können.

Daraufhin werden zum Anwendungs-Code hinzugefügte Felder und Methoden aufgezählt und deren Aufgaben beschrieben. Anschließend wird auf die Veränderungen an schon vorhandenen Methoden näher eingegangen.

Wird eine Methode durch den Transformer verändert, so werden nähere Angaben zu der Stelle, in die der generierte Code in die Methode eingefügt wird, gemacht. Dies geschieht mit „Zum Anfang“ (äquivalent zum Aufruf `addBegin()`) oder „Zum Ende“ (`addEnd()`). Falls keine Angaben gemacht werden, wird der Code in der Mitte (`addBody()`) der Methode eingefügt.

Schließlich werden die vom Transformer auf der Serverseite bereitgestellten Operationen (Server Op, OID) vorgestellt. Die vom, beziehungsweise zum, Client in Bytearray-Form gesendeten Daten werden für jede Operation angegeben und die Auswirkungen der Operationen näher beschrieben

Dabei gelten folgende Konventionen:

- `{Field.name}` bezeichnet den Namen des dekorierten Feldes.
- `^{Field.name}` bezeichnet den Namen des dekorierten Feldes, jedoch ist der erste Buchstabe in diesem Fall groß geschrieben.
- `{Field.type}` bezeichnet den Typ des dekorierten Feldes
- `{class.shortName}` bezeichnet den unqualifizierten Klassennamen des gerade zu bearbeitenden MobCon-Codes.

## A.1 Data Persistence (dp.jar)

**Überblick:**

Dieser Transformer hilft Daten auf Client- und Serverseite persistent zu sichern. Auf der Clientseite wird dies mit Hilfe des `RecordStore`-Objekts erreicht, durch welches Daten im nichtflüchtigen Speicher des Clients gehalten werden können. Auf der Serverseite werden die Daten mit Hilfe von `Treemap`-Objekten gespeichert.

Die zu speichernden Daten werden in einem `StoreObject` mit Namen `{class.shortName}StoreObject` zusammengefasst, welches automatisch von diesem Transformer generiert wird und die als persistent gekennzeichneten Felder/Objekte hält. Dieses Objekt ist auch für die Umwandlung der Felder/Objekte in das zur Speicherung benötigte Bytearray verantwortlich. Normalerweise sollte auf dieses Objekt aber nie direkt, sondern nur über die `get/set^{Field.name}` Methoden zugegriffen werden. Natürlich implementiert dieses Objekt das `Storeable`-Interface und die `object2record`- und `record2object`-Methoden.

**WICHTIG:**
Immer die `get/set^{Field.name}`-Methoden benutzen, falls Werte der persistenten Felder geändert werden sollen. Falls diese Methoden nicht benutzt werden, kann für eine persistente Speicherung nicht garantiert werden!

| Tag | Wirkung (Falls benutzt) | Parameter |
|---|---|---|
| *@dp* | Standard-Präfix | - |
| **Standard- und Storeable-Objekte** | | |
| *access* | Dieses Feld wird dem `{class.shortName}StoreObject` hinzugefügt. Ein Feld mit Namen `All^{Field.name}` wird zu dem `{class.shortName}StoreObject` hinzugefügt, welches den Standardwert dieses Felds hält (`String = "*", int = -1, Object = null`). Get/Set-Methoden werden für dieses Feld im `{class.shortName}StoreObject` generiert. | - |
| *store* | Nur primitive Typen oder Objekte, die sich von `Storeable` ableiten, können mit diesem Tag dekoriert werden. Dieses Feld wird anschließend auf dem Client oder auf dem Server persistent gespeichert. Präziser: Es wird in die `object2record`- und `record2object`-Methoden des `{class.shortName}StoreObject` eingeschlossen. | - |

| Hinzugefügte Felder | Aufgabe |
|---|---|
| `{class.shortName}Store store` | Verwaltet den `RecordStore` dieser Anwendung. Der Name des `RecordStore`-Objekts ist `{class.shortName}`. |

| Hinzugefügte Methoden | Aufgabe |
|---|---|
| `void set^{Field.name}({Field.type} in)` | Diese Methode wird für jedes Feld, das mit *store* dekoriert ist, generiert. Speichert das `{class.shortName}StoreObject` im nichtflüchtigen Speicher des Clients. |
| `{Field.type} get^{Field.name}()` | Lädt das `{class.shortName}StoreObject` aus dem nichtflüchtigen Speicher des Clients und gibt das Feld zurück. |

| | |
|---|---|
| `String storeObject(Storeable sa)` | Speichert ein Objekt, welches das `Storeable`-Interface implementiert, auf dem Server und gibt einen `String` zurück, welches dieses Objekt identifiziert (Für einen späteren Zugriff). |
| `void retrieveObject(String id, Storeable sa)` | Empfängt das Objekt mit der Kennung id (vorher mit der Methode `storeObject()` schon gespeichert) vom Server und speichert es in `sa`. `sa` sollte vom gleichen Typ wie das schon gespeicherte Objekt sein. |
| `void store()` | Speichert das `{class.shortName}StoreObject` auf dem Server. Somit werden die Felder auf dem Server persistent gemacht. |
| `void retrieve()` | Empfängt das `{class.shortName}StoreObject` vom Server. |
| `String getMobileID()` | Empfängt eine eindeutige ID vom Server und speichert diese im `{class.shortName}StoreObject`. Damit wird dem Paar Client/Anwendung eine eindeutige ID zugewiesen, welche persistent gespeichert und wieder hergestellt werden kann. |

| **Veränderte Methoden** | **Veränderung** |
|---|---|
| `void startApp()` | Zum Anfang:<br>Laden des `{class.shortName}StoreObject`. |
| `void destroyApp(boolean)` | Speichert das `{class.shortName}StoreObject` auf dem Server und schliesst den `{class.shortName}Store store`. |

| Server Op | Daten | Aufgabe |
|---|---|---|
| 1 | Vom Client: `{class.shortName}StoreObject` | Speichert das `{class.shortName}StoreObject` an der mit `getMobileID()` ermittelten Position im Treemap des Servers. |
| 2 | Vom Server: `{class.shortName}StoreObject` | Sendet vom Server das `{class.shortName}StoreObject` des Clients als Bytearray zurück. |
| 3 | Vom Client: `Storeable`<br>Vom Server: String ID | Speichert das `Storeable` auf dem Server und gibt die ID zurück, mit der man später wieder darauf zugreifen kann. |
| 4 | Vom Client: String ID<br>Vom Server: `Storeable` | Sendet das `Storeable`, an der Stelle ID in der `Treemap`, zurück zum Client. Die ID wurde mit dem Speichern des Objekts auf dem Server schon empfangen. |

## A.2 Screen Manager (screen.jar)

**Überblick:**

Dieser Transformer vereinfacht das Erstellen der Bildschirme (`Displayable`-Objekte) einer mobilen Anwendung. Es können auf einfache Art und Weise Objekte eines Anzeigeschirms, zum Beispiel Textfelder oder Listboxen, erzeugt und benutzt werden.

Die Tags eines `Form`-Objekts im MobCon-Code, die auf andere Objekte im MobCon-Code verweisen, werden in der Reihenfolge dem `Form`-Objekt im Anwendungs-Code hinzugefügt, in der sie als Tag im MobCon-Code stehen.

| Tag | Wirkung (Falls benutzt) | Parameter |
|---|---|---|
| **Alle Objekte:** | | |
| *@scr* | Standard-Präfix | - |
| *height* | Höhe des Objekts | int |
| *width* | Breite des Objekts | int |
| **Form-Objekte:** | | |
| *label* | (Obligatorisch) Label dieses Objekts.<br>Fügt die Methode `void call^{Field.name}` dem Anwendungs-Code hinzu. | String |
| *firstDisplay* | Dieses Form-Objekt wird als erstes in der Anwendung angezeigt. | - |
| *nextButton* | Fügt einen next-Button diesem Form-Objekt hinzu. Bei Betätigung des Knopfes wird dasjenige Form-Objekt aufgerufen, welches als Parameter übergeben wurde. | Form |
| *backButton* | Fügt einen back-Button diesem Form-Objekt hinzu. Bei Betätigung des Knopfes wird dasjenige Form-Objekt aufgerufen, welches als Parameter übergeben wurde. | Form |
| *exitButton* | Fügt einen exit-Button diesem Form -Objekt hinzu. | |
| *image* | Fügt ein Image-Objekt diesem Form-Objekt hinzu. Wird immer als letztes Element des Form -Objekts im Anwendungs-Code eingefügt. | Image |
| *textField* | Ein oder mehrere TextField-Objekte werden diesem Form-Objekt hinzugefügt. | Ein oder mehrer TextField-Objekte |
| *command* | Fügt einen selbst erstellten Button diesem Form -Objekt hinzu. Das im Parameter übergebene Command-Objekt sollte schon vorhanden sein. | Command |
| *stringItem* | Fügt ein StringItem-Objekt dem Form-Objekt hinzu. | |

| | | |
|---|---|---|
| *imageItem* | Fügt ein `ImageItem`-Objekt dem `Form` -Objekt hinzu. | |
| *choiceGroup* | Fügt ein `ChoiceGroup`-Objekt dem `Form` -Objekt hinzu. | |
| **TextField-Objekte:** | | |
| *label* | (Obligatorisch) Label dieses Objekts.<br>Fügt die Methode `void call^{Field.name}` hinzu. | `String` |
| *string* | Text, der in diesem `TextField` angezeigt wird. | `String` |
| *constraints* | Constraints dieses `TextField`-Objekts. | `Constraints` |
| *maxSize* | Maximale Größe (in Zeichen) dieses `TextField`-Objekts. | `int` |
| **List-Objekte:** | | |
| *label* | (Obligatorisch) Label dieses Objekts.<br>Fügt die Methode `void call^{Field.name}` hinzu. | `String` |
| *listType* | `ListType` dieses Objekts. | `ListType` |
| *listElements* | Die Elemente die in dieser Liste angezeigt werden als String(s). | Ein oder mehrere `Strings` |
| *listElementArray* | Die Elemente die in dieser Liste angezeigt werden als Array. | `String[]` |
| *commandAction* | Führt die angegebene Methode aus, wenn ein Element der Liste ausgewählt wird. | `Methoden-Name` |
| **Command-Objekte:** | | |
| *label* | (Obligatorisch) Label dieses Objekts.<br>Fügt die Methode `void call^{Field.name}` hinzu. | `String` |
| *addComand* | (Obligatorisch)<br>Bewirkt, dass dieses `Commando` der Anwendung hinzugefügt wird. Kann später auf einen Button gelegt werden. Dies geschieht mit dem Tag `command` im `Form`-Objekt. | - |

| | | |
|---|---|---|
| *execute* | Als Parameter bekommt dieses Tag einen Code-Block, der ausgeführt wird wenn das `Command`-Objekt aufgerufen wird. Für kompliziertere Commando-Logik ist es besser, die Logik selbst in eine Methode `void call^{Field.name}` zu kapseln. | `Code-Block` |
| **Alert-Objects:** | | |
| *alertTimeout* | Die Zeit, die der Alert-Screen gezeigt wird. In Millisekunden. | `int` |
| *alertNextScreen* | Das auf den Alert-Screen folgende `Displayable`-Objekt. | `Displayable` |
| *alertType* | Typ des Alarms. | `Allert.TYPE` |
| *alertImage* | Dieses Bild wird auf dem Alert-Screen angezeigt. | `Image` |
| *alertText* | Dieser Text wird auf dem Alert-Screen angezeigt. | `String` |

| Hinzugefügte Felder | Aufgabe |
|---|---|
| `String firstForm` | Der Name des `Form`-Objekts, welches als erstes in der Anwendung angezeigt wird. |
| `Hashtable context` | Hier werden Informationen über die aktuelle Session gespeichert (Siehe Transformer session). |
| `String[] listElements` | Ein Stringarray mit den angezeigten List-Elementen eines `List`-Objekts. |
| `Command nextCommand,`<br>`Command backCommand,`<br>`Command exitCommand` | Werden hinzugefügt, falls ein `Form`-Objekt der Anwendung diese benutzen will. |
| `Command selectCommand^{Field.name}` | Der Select Befehl eines dekorierten `List`-Objekts. |
| `TextBox messageBox` | Standard `TextBox`, die mit `callMessageBox(String text)` aufgerufen werden kann. |

| Hinzugefügte Methoden | Aufgabe |
|---|---|
| `void call^{Field.name}()`, für `Form`-Objekte | Instanziert das `Form`-Objekt und fügt die in den Tags dieses `Form`-Objekts spezifizierten Objekte, wie Textfelder und Buttons, dem `Form` -Objekt hinzu. Speichert im `Context`, dass dieses `Form` -Objekt aufgerufen wurde. Wenn diese Methode ausgeführt wird, dann wird dieses `Form`-Objekt auf dem Screen angezeigt. |
| `void call^{Field.name}()`, für `List`-Objekte | Instanziert das `List`-Objekt, fügt die in den Tags dieses Objekts spezifizierten Objekte dem `List`-Objekt hinzu. Wenn diese Methode ausgeführt wird, dann wird dieses List-Objekt auf dem Screen angezeigt. |
| `void call^{Field.name}()`, für `TextField`-Objekte | Instanziert das `TextField`-Objekt, fügt die in den Tags dieses Objekts spezifizierten Objekte dem `TextField`-Objekt hinzu. Um dieses Objekt auf dem Screen anzuzeigen, muss es nur einem `Displayable`-Objekt in der form `displayable.append(…)` hinzugefügt werden. |
| `void call^{Field.name}(String text)`, für `TextField`-Objekte | Wie zuvor, nur mit einem anderen Textinhalt. |
| `void call^{Field.name}()`, für `Command`-Objekte | Leere Methode, oder falls das ***`execute`*** Tag des `Command`-Objekts gesetzt ist, mit dem Code aus dem Parameter dieses Tags. Wird aufgerufen, wenn das damit verbundene `Commando`-Objekt ausgeführt wird. |
| `void viewDisplay(String displayName)` | Diese Methode weist den Namen von `Displayable`-Objekten ihre zugehörigen `void call^{Field.name}()`-Methoden zu. Wenn diese Methode ausgeführt wird, wird auch, die zum `Displayable`-Objekt mit Namen `displayName` gehörende `call^{Field.name}`-Methode aufgerufen |

| Veränderte Methoden | Veränderung |
|---|---|
| `void startApp()` | Zum Ende:<br>Das `Displayable`-Objekt, dessen Name in `firstForm` gespeichert ist, wird auf dem Bildschirm angezeigt. |
| `void destroyApp(boolean unconditional)` | Zum Anfang:<br>Fügt alle Felder, die mit `addToSession` dekoriert sind zum Session-Context hinzu und speichert sie auf dem Server. |
| `void commandAction( Command command, Displayable screen)` | Die Standard-Methode um Commandos auszuführen wird so verändert, dass alle Commandos ihre, mit Hilfe von Tags definierte, Wirkung haben. |

## A.3 Image Adapter (image.jar)

**Überblick:**

Dieser Transformer lädt Bilder in den gängigsten Formaten vom Server, passt diese dem mobilen Endgerät an oder transformiert die Bilder gemäß übergebenen Parametern. Bilder können auch lokal vom Client geladen werden, diese müssen allerdings im PNG-Format vorliegen und gemäß `Class.getResourceAsStream()` zugänglich sein. Alle Bilder auf dem Server werden von ihrem ursprünglichen Bildformat in das einzige von MIDP unterstützte Format, das „PNG“-Format, konvertiert.

Für die Bildtransformationen wird Suns JIMI Software Development Kit [13] benutzt. Dies ist eine Bibliothek zur Transformation und Anpassung von Bilddaten.

| Tag | Wirkung (Falls Benutzt) | Parameter |
|---|---|---|
| *@img* | Standard-Präfix | - |
| **Image-Objekte** | | |
| *name* | (Obligatorisch)<br>Dateiname des Bildes entweder auf dem Server oder lokal auf dem mobilen Endgerät. | `String` |
| *local* | Lädt das Bild lokal aus den Java-Ressourcen, muss gemäß `Class.getResourceAsStream(name)` zugänglich sein. | - |
| *height* | Maximale Höhe des angezeigten Bildes. | `int` |
| *width* | Maximale Breite des angezeigten Bildes. | `int` |
| *maxcolors* | Maximale Anzahl der Farben des angezeigten Bildes. | `int` |
| *maxsize* | Maximale Größe in Byte des angezeigten Bildes. | `int` |

| Hinzugefügte Methoden | Wirkung |
|---|---|
| `Image retrieveImage(String imageName)` | Lädt das Bild mit Dateinamen `imageName` vom Server und gibt ein, dem mobilen Endgerät angepasstes, `Image`-Objekt zurück. |
| `Image retrieveImage(String imageName, int width, int heigth)` | Wie die vorherige Methode, nur das hier ein `Image`-Objekt mit der Höhe `height` und der Breite `width` zurückgegeben wird. Falls das resultierende Bild für das mobile Endgerät zu groß wäre, werden die Höhe und Breite automatisch angepasst. |
| `Image retrieveImage(String imageName, int width, int heigth, int numColors, int maxSize, boolean dither)` | Wie die vorherige Methode, wobei hier noch zusätzlich die maximale Größe, maximale Anzahl der Farben angegeben oder der dither-Modus benutzt werden kann. |

| Veränderte Methoden | Veränderung |
|---|---|
| `void startApp()` | Falls `Image`-Objekte im MobCon-Code dekoriert worden sind, werden diese in den Speicher des mobilen Endgeräts geladen. Das heißt sie stehen schon beim starten der Anwendung zur Verfügung. |

| Server Op | Daten | Aufgabe |
|---|---|---|
| 1 | Vom Client:<br>String name, int colors, int width, int height, boolean dither, int memory. | Lädt das Bild vom Server und passt es gemäß den angegebenen Parametern an den Client an. |

## A.4 Session (session.jar)

**Überblick:**

Dieser Transformer speichert Kontext- und Session-Informationen der Anwendung auf dem Server. Diese Informationen werden beim Verlassen der Anwendung gespeichert und bei einem Neustart der Anwendung wieder in das mobile Endgerät geladen. Der Kontext ist in Form eines Objekts realisiert, welches aus einer `Hashtable`, in der Wert-/Schlüsselpaare in Form von String-Objekten abgelegt werden können, und zusätzlichen Objekten besteht, die mit dem Tag *`addToSession`* dekoriert sind. Diese Objekte müssen alle das `Storeable`-Interface implementieren.

Die Schlüssel der `Hashtable` sind folgende:

- **sessionId**, der eindeutige Bezeichner dieser Session.
- **lastDisplay**, der letzte in dieser Session angezeigte Bildschirm.
- **lastCommand**, das letzte in dieser Session ausgeführte Kommando.
- **userName**, der Name des Benutzers.
- **userPwd**, das Passwort des Benutzers.

| Tag | Wirkung (Falls benutzt) | Parameter |
|---|---|---|
| *@ses* | Standard-Präfix | - |
| **Klassen** | | |
| *rememberLastDisplay* | (Als Class-Tag)<br>Der letzte angezeigte Bildschirm der Anwendung wird im Kontext gespeichert und beim nächsten Aufruf der Anwendung als erstes angezeigt. Mit *noBackEntry* dekorierte Bildschirme werden nicht gespeichert. | - |
| **Storeable-Objekte** | | |
| *addToSession* | Objekte, die sich vom `Storeable`-Interface ableiten, können dem Kontext hinzugefügt werden. Sie werden beim Aufruf der `startApp()`-Methode mit den alten Werten geladen. | `Storeable` |
| **Displayable-Objekte** | | |
| *noBackEntry* | Ein `Displayable`-Objekt kann damit dekoriert werden. Dieses Tag bewirkt, dass dieser Bildschirm nicht „gemerkt" wird im Sinne von *rememberLastDisplay*. | - |

| Hinzugefügte Methoden | Aufgabe |
|---|---|
| `void storeContext()` | Speichert den Kontext auf dem Server. |
| `void retrieveContext()` | Lädt den Kontext vom Server. |

| Veränderte Methoden | Veränderung |
|---|---|
| `void startApp()` | Zum Anfang:<br>Lädt den Kontext vom Server, initialisiert alle Felder die mit *addToSession* dekoriert sind und setzt den Wert des Feldes `firstForm` mit dem Wert von `lastForm` |

| | |
|---|---|
| | aus dem Kontext. |
| `void destroyApp(boolean unconditional)` | Zum Anfang:<br>Speichert die mit *`addToSession`* dekorierten Felder im Kontext und speichert ihn auf dem Server. |

| Server Op | Data | Aufgabe |
|---|---|---|
| 1 | Vom Client:<br>`StoreableStringHashtable ssh` | Speichert den Kontext auf dem Server. |
| 2 | Vom Client:<br>`String sessionId`<br>Zum Client:<br>`StoreableStringHashtable ssh` | Sendet den Kontext zum Client. |

## A.5 Logging (log.jar)

**Überblick:**

Dieser Transformer fügt zu allen Methoden eine Ausgabe des Methodennamens zuzüglich der PREFIX-Tags aller Transformer, die diese Methode verändert haben, auf der Java-Konsole hinzu. Dadurch kann ein eventuell notwendiges Debugen der Anwendung erleichtert werden.

| Tag | Wirkung (Falls Benutzt) | Parameter |
|---|---|---|
| @log | Standard-Präfix | - |
| **Klassen** | | |
| *`action`* | Der Code-Block, der als Parameter dieses Tags übergeben wird, wird vor der Standard-Logging-Ausgabe ausgeführt. | Code-Block |
| *`logCommand`* | Dieser String wird der Logging-Ausgabe eines `Commando`- | `String` |

| | | |
|---|---|---|
| | Aufrufs vorangestellt. | |
| *logMethod* | Dieser String wird der Logging-Ausgabe eines Methoden-Aufrufs vorangestellt. | `String` |

| Veränderte Methoden | Veränderung |
|---|---|
| Alle Methoden | Zum Anfang:<br>Allen Methoden wird eine Logging-Ausgabe hinzugefügt. |

## A.6 Encryption (enc.jar)

**Überblick:**

Dieser Transformer verschlüsselt alle mit dem *encrypt*-Tag dekorierten `TaggedCode`-Objekte der Anwendung, beziehungsweise entschlüsselt die mit *decrypt* dekorierten `TaggedCode`-Objekte. Die `TaggedCode`-Objekte müssen eine Zuweisung im Sinne von `LHS = RHS;` sein wobei der Code/die Variable der RHS (Right Hand Side) ver-/entschlüsselt und der Variablen LHS (Left Hand Side) zugewiesen wird. Außerdem werden die Methoden `encrypt()` und `decrypt()` der Anwendung hinzugefügt, mit denen beliebige Bytearrays verschlüsselt und entschlüsselt werden können.

Zur Verschlüsselung wurden die Algorithmen von BouncyCastle [14] verwendet, die auch von den Java-Entwicklern zur Benutzung in mobilen Anwendungen vorgeschlagen wurden.

| Tag | Wirkung (Falls Benutzt) | Parameter |
|---|---|---|
| *@enc* | Standard-Präfix | - |
| **Klassen** | | |
| *encrypt* | Durchsucht alle TaggedCode-Objekte nach den *encrypt/decrypt* Tag und ver-/entschlüsselt den Code wie im Überblick vorgestellt. Falls dieses Tag nicht gesetzt wird, werden nur die Methoden `encrypt()` und | - |

| | | |
|---|---|---|
| | `decrypt()` der Anwendung hinzugefügt. | |

| Veränderte Methoden | Veränderung |
|---|---|
| Alle Methoden, die mit *`encrypt`* oder *`decrypt`* dekorierte `TaggedCode`-Objekte besitzen. | Versendete Daten werden verschlüsselt (bei *`encrypt`*) und empfangene entschlüsselt (bei *`decrypt`*). Die `TaggedCode`-Objekte müssen eine Zuweisung im Sinne von `LHS = RHS;` sein wobei der Code/die Variable der RHS (Right Hand Side) ver-/entschlüsselt und der Variablen LHS (Left Hand Side) zugewiesen wird. |

| Hinzugefügte Methoden | Aufgabe |
|---|---|
| `byte[] encrypt(byte[] in)` | Verschlüsselt das Bytearray `in` und gibt das verschlüsselte Bytearray zurück. |
| `byte[] decrypt(byte[] in)` | Entschlüsselt das Bytearray `in` und gibt das entschlüsselte Bytearray zurück. |

## A.7 Class Template Mixing Example (ctmixex.jar)

**Überblick:**

Dieser Transformer wurde erstellt, um eine mögliche Anwendung der `ClassTemplate`-Vereinigung mit einem anderen Vereinigungsverhalten zu demonstrieren.

Der Transformer soll einen Third-Party Transformer darstellen, der zusammen mit den von MobCon ausgelieferten Transformern benutzt werden soll. Dieser Transformer geht von einem anderen Aufbau des Anwendungs-Codes aus, welches ein oft vorkommendes Problem sein sollte, da nicht alle Transformer-Entwickler einen gleichen Satz von Variabeln im Anwendungs-Code benutzen werden.

Genau das ist auch hier der Fall. Der Transformer geht davon aus, dass eine Variable vom Typ `Display`, welche die Bildschirminformationen speichert, mit dem Namen `disp` benutzt wird. Dieser Name wird aber nicht von den MobCon-Transformern benutzt. Der von ihnen vergebene Name lautet `display`. Der

Transformer generiert eine Methode `void doSomething()`, die den gerade angezeigten Bildschirm, welcher in dem Objekt mit Namen `disp` gespeichert ist, auf der Java-Konsole ausgibt. Also, muss der Aufruf der Methode zum Absturz führen.

Um dies zu verhindern und den Transformer in dieser Anwendung benutzen zu können, muss ein von `MixingTemplate` abgeleitetes Objekt erstellt werden um die beiden Transformer kompatibel zu machen. Dieses Objekt, nennt die Variabeln um, so dass beide Transformer in derselben Anwendung benutzt werden können.

Dieses Beispiel wird in der Anwendung ctmixing (Anhang C.1) der Diplomarbeit vorgestellt.

# Anhang B

# Anwendungs-Code des „Hello World“-Beispiels

Hier soll einmal nachfolgend der komplette Anwendungs-Code des Beispiels aus Abschnitt 2.7 aufgeführt werden.

**AbstractMobApp.java:**

```
1  import java.util.Hashtable;
2  import javax.microedition.lcdui.*;
3  import javax.microedition.midlet.*;
4  import javax.microedition.midlet.MIDletStateChangeException;
5  import mobcon.message.*;
6  import mobcon.storeables.*;
7
8  public abstract class AbstractMobApp extends MIDlet
9  implements CommandListener
10 {
11     protected Command exitCommand;
12     protected Display display;
13     private Form form;
14     public static String CID =
15                      "af7070f0d205db3d5896163a0102f907";
16     protected String firstForm = "form";
17     protected String[] listElements;
18     protected TextBox messageBox;
19     private TextField textField;
20
21     public  AbstractMobApp()
22     {
```

```
        exitCommand = new Command("Exit", Command.EXIT, 1);
        display = Display.getDisplay(this);
    }

    public void callForm()
    {
        form = new Form("Test");
        form.addCommand(exitCommand);
        form.setCommandListener(this);
        callTextField();
        form.append(textField);
        display.setCurrent(form);
    }

    public void callMessageBox( String label, String text)
    {
        messageBox = new TextBox( label, text, 256,
                                  TextField.ANY );
        display.setCurrent(messageBox);
    }

    public void callTextField()
    {
        String text = "";
        text = "Hello World";
        textField = new TextField("First Application", text,
                                   256, TextField.ANY);
    }

    public void callTextField( String text)
    {
        textField = new TextField("First Application", text,
                                   256, TextField.ANY);
    }

    public void commandAction( Command command,
                               Displayable screen)
    {
        if (command == exitCommand)
        {
           destroyApp(false);
           notifyDestroyed();
        }
    }

```

```
68      public void destroyApp( boolean unconditional)
69      {
70      }
71
72      public void pauseApp()
73      {
74      }
75
76      public void startApp()
77      {
78          viewDisplay(firstForm);
79      }
80
81      public void viewDisplay( String displayName)
82      {
83          if(displayName.equals("form")) callForm();
84      }
85
86
87  } //EOC
```

*Listing B.1: Von MobCon generierter Anwendungs-Code AbstractMobApp.java*

**MobApp.java:**

```
1   import java.util.Hashtable;
2   import javax.microedition.lcdui.*;
3   import javax.microedition.midlet.*;
4   import javax.microedition.midlet.MIDletStateChangeException;
5   import mobcon.message.*;
6   import mobcon.storeables.*;
7
8   public class MobApp extends AbstractMobApp
9   {
10      public  MobApp()
11      {
12          super();
13      }
14  } //EOC
```

*Listing B.2: Von MobCon generierter Anwendungs-Code MobApp.java*

# Anhang C

# Anwendungen der MobCon-Auslieferung

Mit MobCon werden nicht nur das Framework und die bereits implementierten Transformer ausgeliefert, sondern zusätzlich noch 5 Beispielanwendungen. Diese können zur Orientierung oder als Einstieg in MobCon benutzt werden. Auch können diese Beispielanwendungen als Vorlage zur Erstellung neuer Anwendungen dienen, da sie mehrere Gebiete mobiler Anwendungen abdecken.

Vorgehensweise zum Starten der Anwendungen:

- Datei `default.properties` im MobCon-Root-Verzeichnis dieser Anwendung anpassen.
- Mit `ant application` den Transformationsprozess starten.
- (Falls Server vorhanden) Mit `ant application.server.start` die Anwendung auf dem Server starten.
- In einem neuen Fenster, mit `ant application.mobile.start` die Anwendung auf dem Client starten.

## C.1 Anwendung ctmixing

Diese Anwendung benutzt den Transformer ctmixex, der mit MobCon ausgeliefert wurde. Dieser Transformer soll einen Third-Party Transformer darstellen, der von einem anderen Aufbau des Anwendungs-Codes ausgeht, welches ein oft

vorkommendes Problem sein sollte, da nicht alle Transformer-Entwickler einen gleichen Satz von Variabeln im Anwendungs-Code benutzen werden. Was dieser Transformer genau bewirkt ist in der Transformer-Dokumentation in Abschnitt C.7 nachzulesen.

Um ein vorzeitiges Abbrechen der Anwendung zu verhindern muss ein von `MixingTemplate` abgeleitetes Objekt erstellt werden um somit die in dieser Anwendung benutzten Transformer kompatibel zu machen. Dieses `MixingTemplate` muss auch in die Datei `depend.xml` eingefügt werden, um das Vereinigungsverhalten der Transformer zu ändern.

***MobCon-Code der Anwendung ctmixing:***

```
1   import javax.microedition.midlet.*;
2   import javax.microedition.lcdui.*;
3
4   /**
5    * @scr
6    * @log
7    * @log.logCommand
8    * @log.logMethod
9    * @ctmixex
10   */
11  public class MobApp
12  {
13      /**
14       * @scr.label "FirstScreen"
15       * @scr.firstDisplay
16       * @scr.nextButton form2
17       * @scr.stringItem stringItem1
18       */
19      Form form1;
20      /**
21       * @scr.label ""
22       * @scr.string "When no special MixTemplate is used, the
23                      application will abort when pressing next."
24       */
25      StringItem stringItem1;
26      /**
27       * @scr.label "Second Screen"
28       * @scr.exitButton
29       * @scr.stringItem stringItem2
30       */
31      Form form2;
```

```
32      /**
33       * @scr.label "Success! "
34       * @scr.string "Mixing Transformer successfully applied"
35       */
36      StringItem stringItem2;
37
38      public void callForm2()
39      {
40          doSomething();
41      }
42  }
```

*Listing C.1: MobCon-Code der Anwendung ctmixing*

Erläuterungen zu Listing x:

- *Zeile 9*: Das Tag `@ctmixex` veranlasst MobCon den Transformer ctmixex zu benutzen.

- *Zeile 40:* Die vom Transformer ctmixex in den Anwendungs-Code eingefügte Methode `doSomething()` wird beim Anzeigen des `Form`-Objekts `form2` aufgerufen. Wird kein spezielles `MixTemplate` in `depend.xml` definiert, so führt dies zum Absturz.

***Abhängigkeitsdatei „depend.xml“ der Anwendung ctmixing:***

```
1   <!-- It is supposed that the merger will be
2   applied to the transformer that follows in the group
3   or, when it is a group merger, to the group that follows -->
4   <flow>
5       <group>
6           <transformer>
7               <name>Screen</name>
8               <tid>02</tid>
9               <iid>01</iid>
10              <merger>
11                  <class>MixTemplateExample</class>
12              </merger>
13          </transformer>
14          <transformer>
15              <name>Mix Example</name>
16              <tid>07</tid>
17              <iid>01</iid>
18          </transformer>
19      </group>
20      <group>
```

```
21          <transformer>
22              <name>Logging</name>
23              <tid>05</tid>
24              <iid>01</iid>
25          </transformer>
26      </group>
27  </flow>
```

*Listing C.2: Abhängigkteidatei „depend.xml“ der Anwendung ctmixing*

Erläuterungen zu Listing x:

- *Zeile 14-16*: Da die beiden Transformer screen und ctmixex ein besonderes Vereinigungsverhalten benötigen, wird das benötigte `MixTemplate` `MixTemplateExample` in diesen Zeilen definiert. Es wird benutzt um die beiden Transformer zu vereinigen und somit kompatibel zu machen. Diese Zeilen sollten gelöscht werden, um zu sehen wie die Anwendung abstürzt, falls diese fehlen.

## C.2 Anwendung dp

Diese Anwendung ist ein Beispiel für die Verwendung persistenter Daten. Diese Daten können auf zweierlei Art und Weise persistent gehalten werden. Zum einen können sie lokal auf dem mobilen Endgerät gespeichert werden und zum anderen auf dem Server.

Dies kann in diesem Beispiel getestet werden, indem Daten in zwei `TextField`-Objekte eingegeben werden und diese dann entweder „local“ oder „remote“ gespeichert werden. Das heißt bei jedem Neustarten der Anwendung werden diese Daten entweder aus dem lokalen Speicher oder vom Server geholt.

***MobCon-Code der Anwendung dp:***

```
1   import javax.microedition.midlet.*;
2   import javax.microedition.lcdui.*;
3
4   /**
5    * @scr
6    * @dp
7    * @log
8    * @log.logCommand
9    * @log.logMethod
10   * @enc
```

```
 * @enc.encrypt
 */
public class MobApp
{
    /**
     * @scr.label "Form0"
     * @scr.firstDisplay
     * @scr.exitButton
     * @scr.nextButton form1
     * @scr.textField welcome
     */
    Form form0;
    /**
     * @scr.label "WELCOME!"
     * @scr.string "First form is made of persistent fields,
         that can be saved locally or remotely on server(Saving
         remotely, means always saving locally too!!!)."
     */
    TextField welcome;

    /**
     * @scr.label "Form1"
     * @scr.exitButton
     * @scr.textField textField1 textField2
     * @scr.command saveCommand saveRemoteCommand loadCommand
                    loadRemoteCommand
     */
    Form form1;

    /**
     * @dp.access
     * @dp.store
     */
    String text;
    /**
     * @dp.access
     * @dp.store
     */
    String text2;

    /**
     * @scr.addCommand
     * @scr.label "Save Local"
     * @scr.execute
        setText(textField1.getString());
```

```
56          setText2(textField2.getString());
57       */
58      Command saveCommand;
59
60      /**
61       * @scr.addCommand
62       * @scr.label "Save Remote"
63       * @scr.execute
64          setText(textField1.getString());
65          setText2(textField2.getString());
66          store();
67       */
68      Command saveRemoteCommand;
69
70      /**
71       * @scr.addCommand
72       * @scr.label "Load Local"
73       * @scr.execute
74          callMessageBox("FromStore", store.storeToString());
75       */
76      Command loadCommand;
77
78      /**
79       * @scr.addCommand
80       * @scr.label "Load Remote"
81       * @scr.execute
82          retrieve();
83          callForm1();
84       */
85      Command loadRemoteCommand;
86
87      TextField textField1;
88      TextField textField2;
89      /** TEXTFIELD ACTIONS **/
90      public void callTextField1()
91      {
92          String text = "";
93          text = getText();
94          textField1 = new TextField("TextField1", text, 256,
95                                     TextField.ANY);
96      }
97      public void callTextField2()
98      {
99          String text = "";
100         text = getText2();
```

```
101         textField2 = new TextField("TextField2", text, 256,
102                                    TextField.ANY);
103     }
104
105 }
```

*Listing C.3: MobCon-Code der Anwendung dp*

Erläuterungen zu Listing x:

- *Zeile 40-49*: Hier werden die zwei Strings `text` und `text2` mit den Tags ***`access`*** und ***`store`*** dekoriert. Das bedeutet, dass sie persistent gespeichert werden können. Lokal können sie mit den Methoden `get/setText()`, beziehungsweise `get/setText2()` geladen/gespeichert werden. Auf dem Server werden beide nach lokaler Speicherung mit den Methoden `retrieve()`, beziehungsweise `store()` gespeichert oder geladen.

- *Zeile 51-85:* Hier werden die `Command`-Objekte, die für das Speichern und Laden verantwortlich sind, definiert. Je nachdem, ob lokal oder auf dem Server geladen/gespeichert wird, werden die `get/set`-Methoden beziehungsweise zusätzlich die `store/retrieve`-Methoden der persistenten Felder `text` und `text2` aufgerufen. Mit dem ***`execute`*** Tag können Code-Zeilen eingegeben werden, welche ausgeführt werden wenn der entsprechende Button gedrückt wurde. Diese Objekte können einfach dem `Form`-Objekt in Zeile 35-36 hinzugefügt werden.

- *Zeile 90-103*: Die `callTextField()`-Methoden müssen hier definiert werden, da die `TextField`-Objekte in Zeile 34 adressiert, aber nicht dekoriert wurden. Das heißt es wurden auch keine `call…`-Methoden für diese generiert.

## C.3 Anwendung helloworld

Dieses Anwendungsbeispiel wurde schon ausführlich behandelt. Der MobCon-Code ist in Abschnitt 2.7 zu finden und der generierte Anwendungs-Code steht in Anhang B.

## C.4 Anwendung mobray

Um zu zeigen, dass MobCon auch effizient benutzt werden kann, wurde die mobile Anwendung MobRay entwickelt. MobRay ist kein neues Szenario, sondern ein Projekt der EU welches in Großbritannien schon zur Anwendung kommt.

Beschreibung des Szenarios:

Ein kleines Krankenhaus macht Röntgenbilder (X-Ray in Englisch, deswegen der Name MobRay) ihrer Patienten, hat aber keinen Arzt der alleine für die Auswertung dieser Bilder abgestellt ist. Anstelle dieses einen Arztes hat das Krankenhaus ein Online-System, in welches die Bilder und Daten von den Krankenschwestern eingegeben werden können. Die Ärzte können sich dann die Bilder auf ihre PDAs herunterladen, sie auswerten und ihre Auswertung zurück an den Server im Krankenhaus schicken. Somit können die Ärzte diese Arbeit zum Beispiel während der Anreise zum Krankenhaus, oder ganz allgemein, außerhalb des Krankenhauses durchführen.

Dieses Szenario wurde vollständig in einen ca. 300-zeiligen MobCon-Code implementiert und kann als mobile Anwendung im Anwendungs-Verzeichnis gefunden werden. Der generierte Anwendungs-Code umfasst ca. 1400 Zeilen, dabei ist aber der von MobCon bereitgestellte Code der Transformer auf der Serverseite noch nicht einbezogen.

Der MobCon-Code wird hier nicht vollständig wiedergegeben, sondern nur exemplarisch an typischen Beispielen erläutert. Der MobCon-Code kann grob in drei Teile zerlegt werden:

- Erstens, die mit Transformer-Tags dekorierten Objekte. Das sind meistens Objekte die auf dem Bildschirm angezeigt werden.

```
1   /**
2    * @scr.label "Login"
3    * @scr.firstDisplay
4    * @scr.exitbutton
5    * @scr.textField TF_userName TF_passWord
6    * @scr.command C_login
7    */
8   Form F_login;
```

*Listing C.4: Dekorierte Objekte, Auszug aus mobray*

- Zweitens, dekorierte Objekte mit Anwendungslogik, dass sind die `Command`-Objekte, die bei Benutzung eine im Tag festgelegte Code-Folge ausführen. Diese werden dazu benutzt, den Ablauf der Anwendung zu beschreiben

```
1   /**
2    * @scr.addCommand
3    * @scr.label "Login"
4    * @scr.execute
5       context.put("userName", TF_userName.getString());
6       context.put("userPwd", TF_passWord.getString());
7       if(TF_passWord.getString().equals("pwd"))
```

```
8                                       callF_welcome();
9           else callA_badPassword();
10       */
11      Command C_login;
```

*Listing C.5: Dekorierte Objekte mit Anwendungslogik, Auszug aus mobray*

- Und Drittens, der in Methoden gekapselten Anwendungslogik, der für MobRay spezifische Aktionen beschreibt oder mit dem serverseitigen Teil der Anwendungslogik (Hier das Krankenhaus) kommuniziert.

```
1       public void storeEntry(DataBaseEntry dbe)
2       {
3           try{
4               NetMessage message = new NetMessage();
5               message.setCID(CID);
6               message.setPID("0");
7               message.setIID("0");
8               message.setOID("1");
9               message.setData(dbe.object2record());
10              CTX.sendNetMessage(message);
11          } catch(Exception e){
12              System.out.println("Error in storeEntry: "+e);
13          }
14      }
```

*Listing C.6: Anwendungslogik, Auszug aus mobray*

Erläuterungen zu den Listings C.4, C.5 und C.6:

- *Listing C.4, Zeile 6*: Dem Form-Objekt wird ein `Command`-Objekt mit dem Tag *`command`* zugewiesen. Dieses kann in Form eines Buttons auf dem Bildschirm benutzt werden.

- *Listing C.5, Zeile 4-9*: In dem *`execute`*-Tag werden der Name und das Passwort des Anwenders im Kontext gespeichert. Bei der Eingabe des richtigen Passworts, in diesem Falle „pwd“, wird der Bildschirm `F_welcome` aufgerufen. Bei einem falschen Passwort wird der `Alert`-Bildschirm `A_badPassword` angezeigt.

- *Listing C.6*: In dieser Methode wird ein `DataBaseEntry`-Objekt, welches alle Informationen über einen Patienten hält, auf dem Server gespeichert. Der serverseitige Teil der Anwendungslogik wird mit der Kennung PID=IID=0 angesprochen.

## C.5 Anwendung screen

In dieser Anwendung werden die am häufigsten verwendeten Anzeige-Objekte in MIDP und deren Benutzung vorgestellt. Aus dieser Anwendung können leicht Code-Zeilen herausgenommen werden, um diese in einer anderen Anwendung zu benutzen. Deshalb wird dieses Beispiel anschließend ausführlichst erläutert.

***MobCon-Code der Anwendung screen:***

```
1   import javax.microedition.midlet.*;
2   import javax.microedition.lcdui.*;
3
4   /**
5    * @scr
6    * @img
7    * @log
8    * @log.logCommand
9    * @log.logMethod
10   */
11  public class MobApp
12  {
13      /**
14       * @scr.label "First"
15       * @scr.firstDisplay
16       * @scr.nextButton alertScreen
17       * @scr.exitButton
18       * @scr.stringItem stringItem
19       * @scr.textField textField1 textField2
20       */
21      Form form1;
22
23      /**
24       * @img.local
25       * @img.name "/xrayLocal.png"
26       */
27      Image ray;
28
29     /**
30       * @scr.label "Image"
31       * @scr.image ray
32       * @scr.altText "This is an image Item"
33       * @scr.layout ImageItem.LAYOUT_CENTER
34       */
35      ImageItem imageItem;
36
```

```
37      /**
38        * @scr.label "Second"
39        * @scr.nextButton list1
40        * @scr.backButton form1
41        * @scr.exitButton
42        * @scr.imageItem imageItem
43        * @scr.textField textField2
44        */
45       Form form2;
46
47       /**
48        * @scr.label "Text1"
49        * @scr.string "Hello"
50        */
51       TextField textField1;
52
53       /**
54        * @scr.label "Text2"
55        * @scr.string "How are you"
56        */
57       TextField textField2;
58
59       /**
60        * @scr.label "List"
61        * @scr.listType List.IMPLICIT
62        * @scr.listElements "NextWindow" "Exit"
63        * @scr.commandAction sampleListAction1
64        */
65       List list1;
66
67       /**
68        * @scr.label "List"
69        * @scr.listType List.EXCLUSIVE
70        * @scr.listElementArray elArray
71        * @scr.commandAction sampleListAction2
72        */
73       List list2;
74
75      /**
76        * @scr.label "Alert"
77        * @scr.alertText "This is an alert Screen"
78        * @scr.alertType ERROR
79        * @scr.alertTimeout Alert.FOREVER
80        * @scr.alertNextScreen form2
81        */
```

```
82      Alert alertScreen;
83
84     /**
85       * @scr.label "String"
86       * @scr.string "This is an string Item"
87       */
88      StringItem stringItem;
89
90      String elArray[];
91      /**
92       * @begin
93       */
94      public void callList2()
95      {
96          elArray = new String[2];
97          elArray[0] = "aaa";
98          elArray[1] = "bbb";
99      }
100
101     /**** Methods for the List Commands ****/
102     public void sampleListAction1(List list)
103     {
104         String selected =
105                         listElements[list.getSelectedIndex()];
106         if(selected == "NextWindow")
107         {
108             callList2();
109         }
110         if(selected == "Exit")
111         {
112             commandAction(exitCommand, list);
113         }
114     }
115     public void sampleListAction2(List list)
116     {
117         String selected =
118                         listElements[list.getSelectedIndex()];
119         callMessageBox("Message" ,selected);
120     }
121 }
```

*Listing C.7: MobCon-Code der Anwendung screen*

Erläuterungen zu Listing C.7:

- *Zeile 5-9*: Die Transformer `screen`, `image` und `logging` sollen diesen MobCon-Code transformieren. Außerdem wird dem Logging-Transformer mitgeteilt, dass sowohl Command-Aufrufe wie auch Methoden-Aufrufe geloggt werden sollen.

- *Zeile 13-21*: Dieses `Form`-Objekt wird der erste angezeigte Bildschirm der Anwendung sein (dekoriert mit ***`firstDisplay`***). Ein Exit- und ein Next-Button werden der Anwendung hinzugefügt. Außerdem werden auf dem Bildschirm noch ein `StringItem`-Objekt und zwei `TextField`-Objekte angezeigt.

- *Zeile 23-27*: Ein `Image`-Objekt wird definiert, dessen Daten lokal in der Datei `xrayLocal.png` zu finden sind. Dieses Objekt kann noch nicht direkt einem Bildschirm zugefügt werden. Es muss erst in einem `ImageItem`-Objekt gekapselt werden.

- *Zeile 29-36*: Das `ImageItem`-Objekt setzt sich aus dem vorher definierten `Image`-Objekt, einem alternativen Text (Falls das Bild nicht angezeigt werden kann) und dem Layout der Grafik zusammen. Dieses Objekt kann Bildschirmen in MIDP zugefügt werden.

- *Zeile 42*: Auf diese Weise werden in MobCon den Bildschirmen Bilder zugefügt.

- *Zeile 59-67*: Diesem `List`-Objekt wird mit dem Tag ***`commandAction`*** als Parameter der Name der Methode übergeben, die bei Auswahl aus der Liste aufgerufen wird. Diese muss später selbst implementiert werden. Die Elemente dieser Liste sind die zwei Strings „NextWindow“ und „Exit“.

- *Zeile 70*: Diesmal werden die Elemente der Liste in einem Array `elArray` übergeben.

- *Zeile 94-99*: Die Methode `callList2()`, welche bei Aufruf des Objekts `list2` ausgeführt wird, wird Code an den Anfang (Durch den Tag ***`begin`*** vor der Methode gekennzeichnet) angehängt. Dieser Code initialisiert das Array `elArray`.

- *Zeile 102-120*: Die beiden Methoden, welche aufgerufen werden wenn eine Auswahl in dem jeweiligen `List`-Objekt getroffen wurde, werden hier definiert. In der Methode `sampleListAction1(List)` wird das weitere Verhalten anhand des ausgewählten `String`-Objekts getroffen. Während bei der Methode `sampleListAction2(List)` das ausgewählte Objekt einfach nur auf dem Bildschirm ausgegeben wird.

# Literaturverzeichnis


[1] Sun's Java Home Page. *http://java.sun.com/*.

[2] Richard Monson-Haefel. *Enterprise JavaBeans, Third Edition*. O'Reilly, 2001.

[3] Sun's EJB Home Page. *http://java.sun.com/products/ejb/*.

[4] Microsoft's COM+. *http://www.microsoft.com/com/tech/COMPlus.asp* .

[5] Java's 2 Platform, Micro Edition (J2ME). *http://java.sun.com/j2me/*.

[6] Java's J2ME Mobile Information Device Profile (MIDP). *http://java.sun.com/products/midp/*.

[7] Thoughtworks' QDox. *http://QDox.codehaus.org/*.

[8] vDoclet. *http://vdoclet.sourceforge.net/*.

[9] Apache's Velocity. *http://jakarta.apache.org/velocity/*.

[10] Java's JavaDoc. *http://java.sun.com/j2se/javadoc/*.

[11] Xparse-J. *http://www.webreference.com/xml/tools/xparse-j.html*.

[12] J2ME Wireless Toolkit 2.0. *http://java.sun.com/products/j2mewtoolkit/*

[13] JIMI Software Development Kit. *http://java.sun.com/products/jimi/*

[14] Bouncy Castle Crypto APIs. *http://www.bouncycastle.org/*.

[15] GenJava Projekt. *http://www.generationjava.com*.

[16] Java Exchange. *http://www.javaexchange.com/*.

[17] Java 2 Platform, Standard Edition v1.4.1. *http://java.sun.com/j2se/1.4.1/index.html*.