

Abschlussprüfung Sommer 2018

Fachinformatiker für Anwendungsentwicklung

Dokumentation zur betrieblichen Projektarbeit

**Dynamische Einbindung von**

**Machine Learning Modellen in iOS App**

Abgabetermin: 02.05.2017

**Prüfungsbewerber:**

Ole Sperlich

Stedinger Straße 73

26135 Oldenburg



**Ausbildungsbetrieb:**

CEWE Stiftung & Co. KGaA

Meerweg 30-32

26133 Oldenburg

Inhaltsverzeichnis

[Abbildungsverzeichnis 4](#_Toc511226912)

[Tabellenverzeichnis 4](#_Toc511226913)

[1 Einleitung 5](#_Toc511226914)

[1.1 Projektumfeld 5](#_Toc511226915)

[1.2 Projektbeschreibung 5](#_Toc511226916)

[1.3 Projektziel 6](#_Toc511226917)

[1.4 Projektbegründung 6](#_Toc511226918)

[2 Projektplanung 6](#_Toc511226919)

[2.1 Projektphasen 6](#_Toc511226920)

[2.2 Ressourcenplanung 7](#_Toc511226921)

[2.3 Projektanforderungen 7](#_Toc511226922)

[3 Analysephase 8](#_Toc511226923)

[3.1 Ist-Analyse 8](#_Toc511226924)

[3.2 Fachliche Einarbeitung 8](#_Toc511226925)

[3.3 Technische Einarbeitung 8](#_Toc511226926)

[3.4 Wirtschaftlichkeitsanalyse 9](#_Toc511226927)

[3.4.1 „Make or Buy Decision“ 9](#_Toc511226928)

[3.4.2 Projektkosten 9](#_Toc511226929)

[3.4.3 Nutzwertanalyse 10](#_Toc511226930)

[4 Entwurfsphase 10](#_Toc511226931)

[4.1 Umsetzung planen 10](#_Toc511226932)

[4.1.1 Planung der serverseitigen Speicherung 10](#_Toc511226933)

[4.1.2 Planung des Feedbacks in UI 10](#_Toc511226934)

[4.3 Recherche 11](#_Toc511226935)

[5 Implementierungsphase 11](#_Toc511226936)

[5.1 Umsetzung des Features 11](#_Toc511226937)

[5.2 Einbindung in UI 11](#_Toc511226938)

[6 Qualitätssicherung 11](#_Toc511226939)

[6.1 Funktionstests 11](#_Toc511226940)

[6.2 Performancetests 11](#_Toc511226941)

[6.3 Usabilitytests 11](#_Toc511226942)

[7 Dokumenation 11](#_Toc511226943)

[8 Abnahme des Projekts 11](#_Toc511226944)

[Literaturverzeichnis 12](#_Toc511226945)

[A. Anhang 13](#_Toc511226946)

[A.1. Screenshot der Filialauswahl per Suche nach PLZ oder Ort 13](#_Toc511226947)

[A.2. Detaillierte Zeitplanung 13](#_Toc511226948)

[A.3. Verwendete Ressourcen 14](#_Toc511226949)

[A.4. Screenshot der Filialauswahl per Umkreissuche 15](#_Toc511226950)

[A.5. Quellcodeauszug der Erweiterung der XTCI API 16](#_Toc511226951)

[A.6. Quellcodeauszug der Klasse CWOrderWizardScreenStoreSelectionRadius.cpp 18](#_Toc511226952)

[A.7. Quellcodeauszug der Integration des XTCI-Requests 21](#_Toc511226953)

# Abbildungsverzeichnis

[Abbildung 1: Use-Case-Diagramm 7](#_Toc418682859)

[Abbildung 2: Kartenanzeige 11](#_Toc418682860)

[Abbildung 3: Mockup 12](#_Toc418682861)

[Abbildung 4: Aktivitätsdiagramm 14](#_Toc418682862)

[Abbildung 5: Klassendiagramm 15](#_Toc418682863)

[Abbildung 6: Screenshot der Filialauswahl per Umkreissuche 18](#_Toc418682864)

# Tabellenverzeichnis

[Tabelle 1: Grobe Zeitplanung 8](#_Toc418682865)

[Tabelle 2: Projektkosten 9](#_Toc418682866)

[Tabelle 3: Nutzwertanalyse 10](#_Toc418682867)

[Tabelle 4: Soll-Ist-Vergleich 20](#_Toc418682868)

# 1 Einleitung

Die folgende Projektdokumentation zeigt den Verlauf des IHK-Abschlussprojektes „Dynamische Einbindung von Machine Learning Modellen in iOS App“, welches im Rahmen der Ausbildung zum Fachinformatiker der Fachrichtung Anwendungsentwicklung im Unternehmen CEWE Stiftung & Co. KGaA durchgeführt wurde.

## 1.1 Projektumfeld

Kunde/Auftraggeber bei diesem Projekt ist die CEWE Stiftung & Co. KGaA. CEWE ist im Bereich Foto-Service Marktführer in Europa. Weitere wichtige Geschäftsfelder für CEWE sind Fotofinishing und kommerzieller Online-Druck. Momentan beschäftigt CEWE 3.600 Mitarbeiter und beliefert neben stationärem Handel und Internethandel 20.000 Handelspartner in 25 europäischen Ländern.

//TODO: Fußnote

Verweis: <https://company.cewe.de/de/ueber-uns/unternehmensgruppe.html> Abruf: 06.04.2018

Konkreter Auftraggeber innerhalb von CEWE ist der Leiter eines neu gegründeten Projektteams, das sich mit der Erforschung von Anwendungsmöglichkeiten für künstliche Intelligenz bei CEWE befassen soll. Das Team hat das Ziel, ökonomisch sinnvolle Anwendungsmöglichkeiten zu finden und zu implementieren. Die primäre Aufgabe ist zunächst die Entwicklung eines Assistenten, der automatisch ein CEWE FOTOBUCH mit den schönsten Fotos des Users generiert. Dieser soll auf Mobilgeräte ausgelegt sein und wird auf künstlichen neuronalen Netzen basieren.

## 1.2 Projektbeschreibung

Es gibt bereits Assistenten, die ein CEWE FOTOBUCH erstellen können. Diese basieren auf der Auflösung eines Bildes und auf der Sterne-Bewertung die ein User einem Bild im Dateiexplorer vergeben kann. Allerdings bewertet kaum ein User seine Bilder vorher, bzw. ist dies auch gar nicht möglich auf Mobilgeräten. Des Weiteren sind gerade mit Smartphones aufgenommene Bilder trotz hoher Pixelzahl/Auflösung nicht besonders scharf.

Der CEWE FOTOBUCH Assistent für Mobilgeräte soll deshalb Bilder danach bewerten können, wie ästhetisch ansprechend ein Bild ist. Dazu soll ein neuronales Netz trainiert werden, das die Ästhetik eines Bildes mit einem Score zwischen 0 und 10 bewerten kann.

Um verschiedene Netze/Modelle ausprobieren und vergleichen zu können, wurde eine iOS App entwickelt mit dem Namen Aesthetics (zum Zeitpunkt der Antragsstellung noch ML Tester). In dieser App kann der User Bilder aus seiner Galerie auswählen oder selbst ein Foto aufnehmen. Dieses wird dann bewertet und Score und Zeitpunkt werden angegeben. Dafür kann der User vorher auswählen, mit welchen Machine Learning Modell die Bilder bewertet werden sollen. Screenshots des Ist-Zustands der App befinden sich im Anhang A. 1 Screenshots zur Projektbeschreibung.

Inhalt des IHK-Abschlussprojekts war es, eine dynamische Einbindung von neuen Modellen zu ermöglichen ohne dass ein Entwickler das neue Modell manuell in die App integrieren und eine neue Version verteilen muss.

## 1.3 Projektziel

Ziel des Projekts war es, dass das Entwicklerteam, das sich mit der Suche nach einem geeigneten Netz beschäftigt, jederzeit ein neues Modell einpflegen kann ohne den Entwickler von Aesthetics beauftragen zu müssen, die App zu modifizieren und die neue Version zu verteilen.

Dazu sollte die Möglichkeit geschaffen werden, Modelle zentral auf einem Serverpfad zu speichern, welche dann im Hintergrund der App abgefragt und heruntergeladen würden.

## 1.4 Projektbegründung

Die Hauptfunktion der App ist der Vergleich von neu entwickelten/entdeckten Modellen mit den schon vorhandenen. Daher ist es essentiell, dass regelmäßig neue Modelle eingebunden werden. Nach jetzigem Stand der Entwicklung können die Mitarbeiter, die ein neues Modell testen wollen nicht daran weiterarbeiten, solange das Modell manuell integriert werden muss.

Hinzu kommt, dass der Entwickler der App oft über längere Zeit nicht verfügbar ist, weshalb sich ein weiterer Entwickler in die App einarbeiten müsste, um die neuen Modelle einbinden zu können.

Daher wurde entschieden, dass eine dynamische Lösung für dieses Problem entwickelt werden soll. Es wurde befunden, dass der daraus resultierende Nutzen den Entwicklungsaufwand rechtfertigt. Darauf soll aber in der Wirtschaftlichkeitsanalyse dieser Projektdokumentation noch weiter eingegangen werden.

# 2 Projektplanung

## 2.1 Projektphasen

Das Abschlussprojekt hatte einen vorgegebenen Zeitrahmen von 70 Stunden. Bevor das Projekt begann, wurde es in verschiedene Phasen unterteilt, um die Entwicklung zu planen. Die grobe Planung der Hauptphasen ist der Tabelle 1 zu entnehmen. Die Hauptphasen wurden weiterhin in detailliertere Phasen unterteilt. Die detaillierte Zeitplanung befindet sich im Anhang //TODO: Verweis

|  |  |
| --- | --- |
| Projektphase | Stunden |
| Analyse | 8 |
| Entwurf | 7 |
| Implementierung | 42 |
| Qualitätssicherung | 8 |
| Dokumentation | 3 |
| Abnahme des Projekts | 2 |
| Gesamt | 70 |

Tabelle 1: Zeitplanung der Hauptphasen

## 2.2 Ressourcenplanung

Im Anhang //TODO: Verweis befindet sich eine Aufstellung aller benötigten Ressourcen. Zu den Ressourcen zählen sowohl Hardware, Software und Personal. Für das Projekt musste keine zusätzliche Hardware angeschafft werden. Die eingesetzte Software unterlag Lizenzen, die die kommerzielle Nutzung kostenfrei erlaubten oder die CEWE bereits gekauft hatte. So entstanden sowohl für Hardware als auch für Software keine Kosten.

Auflistung (Für Anhang) der verwendeten Ressourcen

Hardware:

Büroarbeitsplatz mit Mac Mini

iPhone 7 – Testgerät

iPhone 8 – Testgerät

iPad Mini – Testgerät

Download-Server

Software:

Mac OS X - Betriebssystem

Xcode – Entwicklungsumgebung für Swift

Atom – Editor zum Erstellen von JSON Dateien

Git – Versionsverwaltung

CocoaPods – Dependency Manager

Word – Projektdokumentation

Personal:

Leiter des iOS Teams – Definition der Anforderungen und Abnahme des Projekts

Entwickler – Umsetzung des Projekts

## 2.3 Projektanforderungen

//TODO: Mega geiles Use Case Diagramm

# 3 Analysephase

## 3.1 Ist-Analyse

Die App Aesthetics ist in der Programmiersprache Swift geschrieben und der Quellcode befindet sich in einer Projektdatei der Entwicklungsumgebung Xcode. Deshalb wurde das Abschlussprojekt natürlich auch mit Swift in Xcode umgesetzt. Zur Einbindung von Machine Learning zur Bildanalyse wurden die von Apple entwickelten Frameworks CoreML und Vision benutzt, sowohl in der schon vorher vorhandenen Implementierung als auch im Abschlussprojekt.

Die Integration eines neuen Modells in die App geschah vor Beginn des Abschlussprojekts wie folgt aus: Der Entwickler, der das Modell einbinden sollte, brauchte das unkompilierte Modell im Dateiformat .mlmodel. Dafür musste ihm entweder das Modell geschickt werden oder er konnte es herunterladen, wenn es ein öffentlich zugängliches Modell war. In der Projektstruktur lag ein Verzeichnis mit allen Machine Learning Modellen. Dahin wurden dann alle Modelle kopiert. Da manche Modelle hunderte MB groß sind, wurde dadurch auch die App immer größer.

//TODO: EPK wäre voll nice

Um die Modelle nutzen zu können, musste aus der mlmodel-Datei ein Objekt der Klasse MLModel erzeugt werden. Dies konnte mit folgendem Befehl umgesetzt werden :

let model = try VNCoreMLModel(for: testModel().model)

In dieser Beispielzeile müsste die Datei testModel.mlmodel in dem Projektverzeichnis vorhanden sein. Beim Start der App wurde für jede mlmodel-Datei ein VNCoreMLModel erzeugt. Dieses wurde zusammen mit einem Namen in einem dictionary gespeichert. Ein dictionary bezeichnet in Swift eine key-value Datenstruktur. Der Name war hier der Key und das MLModel der Value. Sowohl der Name für das Modell als auch die zu kompilierenden Dateien mussten hart codiert im Code definiert werden.

## 3.2 Fachliche Einarbeitung

Bei der fachlichen Einarbeitung wurde betrachtet, welche Anwendungsfälle abgedeckt werden sollen. Beispielsweise gibt es verschiede Machine Learning Modelle zur Bildanalyse, z.B. solche, die die Ästhetik eines Bildes bewerten, oder solche, die Objekte und Gesichter in Bildern erkennen. Diese müssten unterschiedlich implementiert werden. Dies war aber nicht notwendig, da in der App nur eine Art von Modellen getestet wird.

Darüber hinaus war nicht viel fachliche Einarbeitung nötig, da der Autor schon vor dem Projekt mit dem Kontext vertraut war.

## 3.3 Technische Einarbeitung

Bei der technischen Einarbeitung wurde überlegt, wie das neue Feature in den bestehenden Code integriert werden kann. Dabei wurde beachtet, dass das vorhandene Model-View-Controller-Prinzip nicht verletzt werden sollte.

## 3.4 Wirtschaftlichkeitsanalyse

### 3.4.1 „Make or Buy Decision“

Bevor ein gegebenes Problem gelöst wird, sollte zuvor entschieden werden, ob eine (evtl. schon vorhandene) Lösung zugekauft wird oder ob es intern mit geringeren Kosten gelöst werden kann.

Da bei diesem Projekt das Problem in der Erweiterung einer schon vorhandenen selbst entwickelten App ist, gibt es keine bereits vorhandenen Lösungen, die CEWE hätte dazukaufen können.

Da das Projekt außerdem einen sehr kleinen Umfang hatte, wäre es unverhältnismäßig teuer, einen externen Entwickler zu beauftragen, der sich zunächst auch noch einarbeiten müsste.

Deshalb sollte das Projekt in Eigenentwicklung umgesetzt werden.

### 3.4.2 Projektkosten

Im Folgenden sollen die Kosten kalkuliert werden, die in diesem Projekt angefallen sind. Hierfür werden Stundenlöhne von 5€ für Auszubildende und 60€ für Projektleiter angenommen. Diese basieren auf gängigen Annahmen bei CEWE und beinhalten Lohn- und Lohnnebenkosten.

Zu den Stundenlöhnen müssen noch Gemeinkosten hinzugerechnet werden. Zu den Gemeinkosten gehören u.a. Kosten für Räumlichkeiten, Nutzung von Hard- und Software, Strom und Wasser. Bei CEWE werden diese Gemeinkosten nach folgendem Pauschalsatz berechnet:

Gemeinkosten = Stundenlohn \* 0,5

Es fielen keine weiteren Kosten an, denn alle benötigte Hardware und Software war schon vorhanden oder kostenfrei.

Die Projektkosten betrugen insgesamt 705,00 €. Die Austellung der Kosten ist der Tabelle 2 zu entnehmen.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vorgang | Personal | Zeit | Personal-kosten | Gemein-kosten | Gesamt |
| Projektumsetzung | 1x Auszubildender | 70 h | 350,00 € | 175,00 € | 525,00 € |
| Rücksprachen & Abnahme | 1x Teamleiter | 2 h | 120,00 € | 60,00 € | 180,00 € |
| Projektkosten gesamt: 705,00 € | | | | | |

Tabelle 2: Projektkosten

### 3.4.3 Nutzwertanalyse

Das Projekt hat keinen direkt bewertbaren monetären Nutzen. Ob es trotzdem lohnenswert ist, kann durch eine Nutzwertanalyse ermittelt werden, denn die Nutzwertanalyse ist ein qualitatives, nicht-monetäres Bewertungsmittel. Dabei werden zwei oder mehrere Alternativen nach selbstgewählten Kirterien bewertet und verglichen. Um zu ermitteln, ob das Projekt eine sinnvollere Lösung für das Problem der Modelleinbindung in die App bewirkt hat, wurde die manuelle Einbindung der Modelle mit der automatisierten Einbindung verglichen.

//TODO: Nutzwertanalyse erklären und verweisen

**Schnelligkeit der Einbindung**

Die Schnelligkeit der Einbindung ist das Hauptkriterium bei der Bewertung der Sinnhaftigkeit des Projekts. Daher ist die Gewichtung hier auch am größtem gewählt worden. Durch das dynamische Einbinden können Modelle deutlich schneller getestet werden, weshalb die Bewertung in der Nutzwertanalyse deutlich zugunsten der dynamischen Einbindung ausfällt.

**Entwicklungsaufwand**

Der Entwicklungsaufwand wurde als etwas weniger wichtig eingeschätzt als der Nutzen, der daraus resultiert. Der initiale Entwicklungsaufwand wäre natürlich entfallen, wenn die bisherige Lösung beibehalten worden wäre. Der Pflegeaufwand im Nachhinein wäre durch das ständige neue Einbinden aber irgendwann ähnlich groß. Ob der Pflegeaufwand nun größer oder kleiner ist als der initiale Entwicklungsaufwand bei diesem Projekt hängt davon ab, wie lange und intensiv die App benutzt wird. In der Nutzwertanalyse wurde näherungsweise eine gleiche Bewertung gegeben.

**Fehlerresistenz**

Dadurch, dass der User selbst Modelle einbinden soll, wird die App fehleranfälliger. Da die User aber alle Entwickler sind, hält sich das Risiko von Fehlern in Grenzen. Durch eine Anwenderdokumentation in Form einer README.txt wurde das Risiko außerdem weiter eingedämmt. Die Fehleranfälligkeit/Fehlerresistenz wurde als nicht so wichtig eingestuft, weil der Entwickler der App im Fehlerfall immer noch eingreifen kann, da er auch im Team mitarbeitet.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Kriterium | Gewichtung | Manuelles Einbinden | | Dynamisches Einbinden | |
| Bewertung | Gewichtet | Bewertung | Gewichtet |
| Schnelligkeit | 5 | 1 | 5 | 4 | 20 |
| Entwicklungsaufwand | 3 | 3 | 9 | 3 | 9 |
| Fehlerresistenz | 2 | 4 | 8 | 2 | 4 |
| Gesamt | **10** |  | **22** |  | **33** |
| **Nutzwert** |  |  | **2,2** |  | **3,3** |

Tabelle 3: Nutzwertanalyse

Ein weiterer nicht-monetärer Nutzen des Projekts ist der Wissensaufbau für das junge Team, denn denn Wissen Thema der Einbindung von Machine Learning Modellen in eine iOS App war vor Beginn des Projekts kaum vorhanden. Die Einbindung per Download und clientseitigem Kompilieren kann für den CEWE FOTOBUCH Assistenten beispielsweise genutzt werden, um die Dateigröße der App gering zu halten und um das Machine Learning Modell des Assistenten zu erneuern ohne eine neu Version der App zu veröffentlichen. Das ist sinnvoll, da das Modell auch durch reale Nutzer dazulernen kann und so immer der neuste Stand eingespielt werden kann.

# 4 Entwurfsphase

## 4.1 Umsetzung planen

### 4.1.1 Planung der serverseitigen Speicherung

In Rücksprache mit dem Auftraggeber, dem Teamleiter, wurde überlegt, wie die Speicherung der Modelle auf dem Server und der Zugriff darauf umgesetzt werden könnte. Die erste gedachte Lösung war, einen Pfad festzulegen, der für die Modelle genutzt werden soll. Dieser hätte dann hartcodiert im Quellcode genutzt werden können.

Manchmal stehen aber schon Modelle, die ein User testen möchte, auf öffentlichen Webseiten zur Verfügung. Beispielsweise gibt es einige Beispielmodelle auf der Webseite von Apple zum Download oder andere Modelle, die in öffentlichen Git-Repositories bereitliegen. Diese Modelle müssten erst von einem User der App heruntergeladen und auf den Server hochgeladen werden, damit die App sie dann wieder herunterlädt. Eine bessere Lösung ist es, dass die App die Modelle dann direkt über die öffentliche Webseite bezieht.

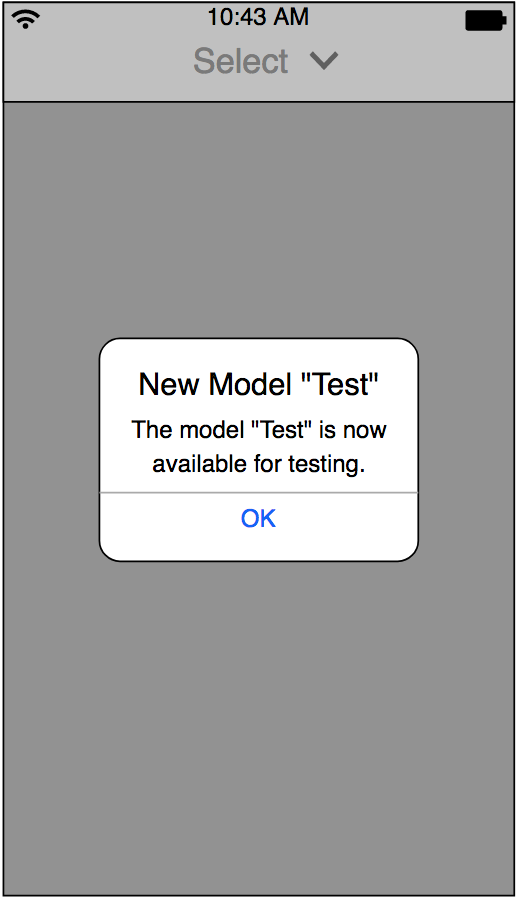
Um das umzusetzen, sollte nur eine JSON Datei auf dem Server liegen, die eine Konfiguration mit den gewünschten URLs beinhaltet. Wer nun ein Modell einbinden will, muss nicht unbedingt sein Modell in diesen Pfad hochladen, sondern trägt einfach die entsprechende URL in der Konfiguration ein. Es sollte dennoch möglich sein, Modelle auf den Server hochzuladen. Der für dieses Projekt genutzte Server ist ein bereits bei CEWE genutzter Server, der es erlaubt, über HTTP/HTTPS Ressourcen herunterzuladen. Daher kann die URL des Modells auf gleiche Weise eingetragen werden wie die öffentlichen Modelle.

### 4.1.2 Planung des Feedbacks in UI

Obwohl das neue Feature ein Hintergrundprozess der App ist, sollte es trotzdem Feedback für den User geben. Ein besonders nutzerfreundliches und schönes Design war dabei nicht zwingend notwendig, da die Anwender nur interne Entwickler sind, die den Prozess zur Einbindung eines Modells kennen. Diese wissen also, dass ein Modell im Hintergrund geladen wird, da sie es vorher selbst in die Konfigurationsdatei geschrieben haben.

Es soll trotzdem eine Meldung angezeigt werden, wenn ein neues Modell heruntergeladen und zum Testen verfügbar ist. Dies sollte mit dem Standard-Alert von iOS-Apps realisiert werden. Im folgenden Mockup ist der geplante Alert zu sehen.

Außerdem musste beachtet werden, dass es nun den Fall geben kann, dass gar keine Modelle in der App vorhanden sind. Vorher war immer bereits mindestens ein Modell eingebunden und eines war immer vorausgewählt. Die geplante Anpassung war nun, dass in der Kopfzeile „Loading models…“ steht, solange noch kein Modell verfügbar ist. Ist dann das erste heruntergeladen, soll „Select“ in der Kopfzeile stehen mit dem Indikator für ein Dropdown (Pfeil nach unten). Da auch keine Bilder bewertet werden können, solange kein Modell ausgewählt ist, sollte außerdem die Fußzeile ausgeblendet werden (die Fußzeile ist den Screenshots im Anhang A. 1 Screenshots zur Projektbeschreibung zu entnehmen). Die geplanten Änderungen sind ebenfalls der Abbildung 1 zu entnehmen.



## 4.3 Recherche

Für die Implementierung musste recherchiert werden, wie Dateien über HTTP/HTTPS heruntergeladen werden können. Ebenfalls musste recherchiert werden, wie diese Dateien permanent auf einem Endgerät gespeichert werden. Des Weiteren wurde auch in Erfahrung gebracht, wie mlmodel-Dateien mit dem CoreML-Framework kompiliert werden können, denn die in der Ist-Analyse vorgestellte Methode ist so nur anwendbar, wenn der Name der Datei schon bekannt ist. Zuletzt wurde zusätzlich eine Bibliothek zum Parsen von JSON-Dateien gesucht und eingebunden.

# 5 Implementierungsphase

## 5.1 Umsetzung des Features

Download per HTTP/HTTPS, Kompilieren der Modelle, Permanente Speicherung auf dem Gerät, JSON Konfiguration

Für die Implementierung des Features in der App wurde die Entwicklungsumgebung Xcode benutzt. Der Zugang zum Server lag zu Beginn des Projekts noch nicht vor und zu Testzwecken konnte ein GitHub-Repository genutzt werden, da dort auch Dateien über HTTPS heruntergeladen werden können. Daher begann die Entwicklung nicht mit dem Anlegen eines Serverpfades mit JSON Konfigurationsdatei, sondern mit der Implementierung des Features in der App.

Die Implementierung begann mit der Überlegung, wie der Code des neuen Features in die Projektstruktur der App eingebunden werden kann. Da dieser Hintergrundprozess eine weitgehend eigenständige Komponente darstellt, wurde eine neu Klasse ModelDownloader angelegt, die für das Herunterladen, Speichern und Kompilieren der mlmodel-Dateien zuständig ist.

Der Download von Dateien sollte zuerst implementiert werden. Dies wurde umgesetzt mit einem URLSessionDownloadTask. Er lädt eine Datei mit einer gegebenen URL herunter und führt eine weitere Funktion (sog. Closure) asynchron aus. Der Download wir im Hintergrund ausgeführt und läuft auch weiter wenn die App nicht aktiv ist. Das Closure muss selbst implementiert werden und bekommt als Parameter die URL der temporär gespeicherten URL und zusätzlich einen HTTPURLResponse oder einen Error vom Downloadtask übergeben. Die Implementierung dieses Tasks ist dem Anhang A. 3 Quellcodeauszüge aus der Klasse ModelDownloader zu entnehmen. Die Funktion, die den Download ausführt ist die Funktion loadFile(title: String, url: NSURL, next:@escaping (\_:String, \_:URL) -> Void) -> Void.

Der URLSessionDownloadTask speichert die heruntergeladene Datei standardmäßig in einem temporären Verzeichnis. So würden allerdings bei jedem Start der App alle Modelle neu heruntergeladen werden. Deshalb sollte eine heruntergeladene Datei in einem permanenten Verzeichnis gespeichert werden. Apple hat für Dateien, die von Apps ohne Einfluss des Users erzeugt werden, ein Verzeichnis mit dem Namen Application Support vorgesehen. Deshalb wird in der loadFile-Funktion zunächst eine destinationFileUrl definiert, die auf eine Datei im Application Support Verzeichnis verweist. In dem Closure des Downloadtasks wird die Datei dann vom temporären Verzeichnis zum Zielverzeichnis verschoben. Außerdem wird die Funktion next(\_:String, \_:URL) in dem Closure aufgerufen, welche der loadFile-Funktion als Parameter übergeben wurde und asynchron ausgeführt wird, sobald die Datei verschoben wurde. Je nachdem, wie die next-Funktion implementiert ist, kann in dieser die heruntergeladene Datei verarbeitet werden.

Der nächste Schritt, der implementiert wurde, ist das Kompilieren der mlmodel-Dateien, nachdem diese heruntergeladen und verschoben wurden. Dafür wurde die Funktion compileModel(title: String, destinationURL: URL) geschrieben. Mit MLModel.compileModel(at: destinationURL) wird eine mlmodel-Datei kompiliert und eine mlmodelc-Datei in einem temporären Verzeichnis gespeichert. In der compileModel-Funktion wird auch diese Datei ins Application Support Verzeichnis verschoben.

Als nächstes sollte ermöglicht werden, dass die konfigurierten Download-URLs aus der JSON Datei ausgelesen werden. Diese JSON Datei hat folgenden Aufbau:

[

{

"title": "Test Model",

"url": "https://example.com/testmodel.mlmodel”

}

]

Diese Datei liegt auf einem festgelegten Downloadserver-Pfad bei CEWE. Mit der Funktion loadURLsFromJSON(next: @escaping ([String: String]) -> Void) wird die JSON Datei gelesen, geparst und in ein dictionary überführt. Zum Auslesen der Datei wurde ein NSURLSessionDataTask benutzt, der im Gegensatz zum NSURLSessionDownloadTask die Datei nicht abspeichert und auch nicht weiterläuft, wenn die App geschlossen ist. Die dafür genutzte URL ist in der Konstante jsonConfigPath gepseichert. Dieser Task ruft einen Closure auf gibt als Parameter die gelesenen Daten als Bytebuffer mit. Das Parsen der Daten wurde mithilfe der Bibliothek SwiftyJSON umgesetzt, welche zur kommerziellen Nutzung kostenlos zur Verfügung steht. Aus dem geparsten JSON-Objekt wird ein dictionary mit title und url erstellt. Dieses dictionary wird dann der next-Funktion als Parameter übergeben.

Diese drei Funktionen sind private. Die einzige Schnittstelle nach außen ist die Funktion getModels(). In dieser wird die loadURLsFromJSON-Funktion aufgerufen und deren next-Funktion implementiert. Dort wird mit jeder URL des dictionarys die loadFile-Funktion aufgerufen, welche als next-Funktion compileModel(title: destinationURL:) übergeben bekommt.

//TODO: Was passiert in PictureTableViewController

//TODO: Löschen von Modellen

## 5.2 Einbindung in UI

# 6 Qualitätssicherung

## 6.1 Funktionstests

Für den Test der Funktionalitäten wurden verschiedene Testfälle und deren erwarteten Ergebnisse definiert. Diese sollten nacheinander ausgeführt und eventuelle Fehler behoben werden, bis alle Testfälle erfolgreich abgeschlossen sind. Jeder dieser Tests war direkt erfolgreich, weshalb keine weiteren Anpassungen vorgenommen werden mussten.

## 6.2 Performancetests

Performancetests waren dringend notwendig, da sowohl das Kompilieren der Modelle als auch deren Anwendung bei der Bildbewertung sehr rechenintensiv ist. In Vorgängerversionen der App führte dies sogar teilweise zu Abstürzen. Wegen Downloads und der Erzeugung von kompilierten Modelldateien wurde außerdem auf Speicherengpässe getestet. Die Performancetests ergaben, dass weder die CPU- und Arbeitsspeicherauslastung noch der Speicherbedarf zu groß ist, weshalb dadurch keine weiteren Anpassungen vorgenommen werden mussten.

## 6.3 Usabilitytests

Da der User der App das neue Feature nicht aktiv nutzen kann, sondern dieses im Hintergrund läuft, mussten keine umfassenden Usabilitytests gemacht werden. Es wurde lediglich getestet, ob alle ursprünglichen Funktionen parallel zum neuen Hintergrundprozess noch immer funktionieren wie vor dessen Einführung. Dies war der Fall, weshalb auch hier keine weiteren Anpassungen gemacht werden mussten.

# 7 Dokumentation

Eine Anwenderdokumentation in der App selbst war nicht nötig, da der User das neue Feature nicht aktiv nutzt, sondern dies im Hintergrund ausgeführt wird.

Für das Einpflegen eines neuen Modells wurde hingegen eine Dokumentation angelegt. Neben der JSON Konfigurationsdatei liegt im selben Verzeichnis eine README.txt, in der geschrieben steht, wie die URLs in der Konfigurationsdatei einzutragen sind. Der Pfad der Konfigurationsdatei ist im firmeninternen Wiki hinterlegt, in dem Bereich, in dem auch andere Dokumentationen des Teams hinterlegt werden.

Bei der Programmierung wurde auf aussagekräftige Kommentare und Funktionsnamen geachtet. Die Kommentare dienen als Entwicklerdokumentation und sollen eventuellen zukünftigen Entwicklern der App dabei helfen, den Quellcode leicht und schnell zu verstehen.

# 8 Abnahme des Projekts

# 9 Fazit

## 9.1 Soll-Ist-Vergleich

Das Projekt wurde erfolgreich im Rahmen von 70 Stunden umgesetzt. Dabei konnten alle zuvor definierten Anforderungen umgesetzt werden. Es gab geringe Abweichungen von der Zeitplanung, welche sich aus der Tabelle 4 entnehmen lassen. Dort sind die geplanten Zeiten (Soll) den tatsächlich benötigten Zeiten (Ist) gegenübergestellt. Wie aus der Tabelle zu entnehmen ist, glichen sich die entstandenen Differenzen wieder aus.

|  |  |  |  |
| --- | --- | --- | --- |
| Projektphase | Soll | Ist | Differenz |
| Analyse | 8 | 8 | 0 h |
| Planung | 7 | 7 | 0 h |
| Implementierung | 42 | 43 | + 1 h |
| Qualitätssicherung | 8 | 8 | 0 h |
| Dokumentation | 3 | 2,5 | - 0,5 h |
| Abnahme | 2 | 1,5 | - 0,5 h |
| Gesamt | 70 h | 70 h | 0 h |

Tabelle 4: Soll-Ist-Vergleich

## 9.2 Ausblick

Das User Interface der App ist noch nicht optimal und könnte noch verbessert werden. Dies ist aber nicht notwendig, da die Anwender der App keine Endnutzer sind, sondern Entwickler, die auch wissen, was im Hintergrund der App passiert. Daher würde es sich nicht lohnen, an der Stelle Zeit zu investieren.

Die Bildbewertung soll im fertigen CEWE FOTOBUCH Assistenten dazu dienen, dass aus einer Menge von Fotos die schönsten ausgewählt werden. Es ist denkbar, dass diese Funktionalität auch in der App Aesthetics zu Testzwecken implementiert werden soll.

# Literaturverzeichnis

[1] **CEWE:** *Unternehmen*, <http://company.cewe.de/de/unternehmen.html>, Abruf: 05.05.2015

[2] **Wikipedia:** *IntelliJ IDEA*, <http://de.wikipedia.org/wiki/IntelliJ_IDEA>, Abruf: 05.05.2015

[3] **Qt Documentation:** *Signals & Slots*, <http://doc.qt.io/qt-4.8/signalsandslots.html>, Abruf: 05.05.2015

[4] **IsiCore:** *Agile Entwicklung und iterative Vorgehensmodelle*, <http://www.isicore.de/isicore-blog/agile-entwicklung-iterative-vorgehensmodelle>, Abruf: 05.05.2015

[5] **ITWissen:** *DTD*, <http://www.itwissen.info/definition/lexikon/document-type-definition-DTD.html>, Abruf: 05.05.2015

[6] **Ubuntuusers:** *Qt*, <http://wiki.ubuntuusers.de/Qt>, Abruf: 05.05.2015

[7] **Ubuntuusers:** *Qt Creator*, <http://wiki.ubuntuusers.de/Qt_Creator>, Abruf: 05.05.2015

# A. Anhang

## A.1. Screenshots zur Projektbeschreibung

## 

## A.2. Detaillierte Zeitplanung

|  |  |
| --- | --- |
| **Projektphasen** | **Stunden** |
| **Analyse** | 8 |
| Ist-Analyse | 2 |
| Fachliche Einarbeitung | 2 |
| Technische Einarbeitung | 2 |
| Kostenanalyse | 2 |
| **Entwurf** | **7** |
| Recherche | 5 |
| Umsetzung planen | 2 |
| **Implementierung** | 42 |
| Umsetzung des Features | 34 |
| Einbindung in UI | 8 |
| **Qualitätssicherung** | 8 |
| Funktionstests | 3 |
| Performancetests | 4 |
| Usabilitytests | 1 |
| **Dokumentation** | 3 |
| **Abnahme des Projekts** | 2 |
| **Gesamt** | 70 |

## A.3. Quellcodeauszüge aus der Klasse ModelDownloader

/// The public method to download and compile the models

public func getModels() {

loadURLsFromJSON(next: { (configDict) in

for item in configDict {

let title = item.key

if title == "" { continue }

guard let url = NSURL(string: item.value) else { continue }

self.loadFile(title: title, url: url, next: self.compileModel(title: destinationURL:))

}

})

}

/// Downloads a file and saves it to Application Support Directory

///

/// - Parameters:

/// - title: the title of the model

/// - url: specifies which file shall be downloaded

/// - next: the function to process the file

private func loadFile(title: String, url: NSURL, next:@escaping (\_:String, \_:URL) -> Void) -> Void {

// Create destination URL

guard let applicationSupportUrl:URL = FileManager.default.urls(for: .applicationSupportDirectory, in: .userDomainMask).first else {

return

}

let destinationFileUrl = applicationSupportUrl.appendingPathComponent(title + ".mlmodel", isDirectory: false)

if FileManager.default.fileExists(atPath: destinationFileUrl.path) {

//next function processes downloaded file

next(title, destinationFileUrl)

return

}

let sessionConfig = URLSessionConfiguration.default

let session = URLSession(configuration: sessionConfig)

let request = URLRequest(url:url as URL)

let task = session.downloadTask(with: request) { (tempLocalUrl, response, error) in

if let tempLocalUrl = tempLocalUrl, error == nil {

if let statusCode = (response as? HTTPURLResponse)?.statusCode {

print("Successfully downloaded. Status code: \(statusCode)")

}

do {

try FileManager.default.moveItem(at: tempLocalUrl, to: destinationFileUrl)

} catch (let writeError) {

print("Error creating a file \(destinationFileUrl) : \(writeError)")

}

//next function processes downloaded file

next(title, destinationFileUrl)

}

}

//call to start task

task.resume()

}

/// Compiles a model and adds it to the models dictionary

///

/// - Parameters:

/// - title: the title of the model

/// - destinationURL: the URL the compiled model file shall be saved to

private func compileModel(title: String, destinationURL: URL) {

do {

let fileManager = FileManager.default

//deleting lastPathComponent (directory) and append it as file (.mlmodelc)

let appSupportDirectory = destinationURL.deletingLastPathComponent()

let permanentCompiledUrl = appSupportDirectory.appendingPathComponent(destinationURL.lastPathComponent, isDirectory: false).deletingPathExtension().appendingPathExtension("mlmodelc")

do {

if !fileManager.fileExists(atPath: appSupportDirectory.path) {

\_ = try fileManager.createDirectory(at: appSupportDirectory, withIntermediateDirectories: false, attributes: nil)

}

//only compile model if not already existing

if !fileManager.fileExists(atPath: permanentCompiledUrl.path) {

let tempCompiledUrl = try MLModel.compileModel(at: destinationURL)

//store compiled model permanently

try fileManager.moveItem(at: tempCompiledUrl, to: permanentCompiledUrl)

ptvc.showSimpleAlert(title: "New Model \"\(title)\"", message: "The model \"\(title)\" is now available for testing.")

}

} catch {

print("\(error.localizedDescription)")

}

let model = try MLModel(contentsOf: permanentCompiledUrl)

models[title] = model

} catch {

fatalError("No model for you today: \(error)")

}

}

/// Loads a dictionary from jsonConfigPath with a model's title as key and URL as value

///

/// - Parameter next: the function to process the dictionary

private func loadURLsFromJSON(next: @escaping ([String: String]) -> Void) {

guard let url = URL(string: self.jsonConfigPath) else { return }

let task = URLSession.shared.dataTask(with: url, completionHandler: {(data, response, error) -> Void in

guard let data = data else { return }

let aestheticsConfig = JSON(data: data)

var configDict = [String: String]()

for item in aestheticsConfig.arrayValue {

configDict[item["title"].stringValue] = item["url"].stringValue

}

next(configDict)

})

//call to start task

task.resume()

}

## A.5. Quellcodeauszug der Erweiterung der XTCI API

***locationlist\_request.dtd:***

[...]

<!ELEMENT locationlist\_request (

((ocsid) | (keyacc\_id, client\_id, client\_version, client\_name?, locale?, affiliate\_id?)),

((zip, city?) | (zip?, city) | (zip, city)

| (loc\_id) | (latitude, longitude, distance, force\_hit?) | (address\_id, distance, force\_hit?))?, truncate?, onsite\_printer\_only?)>

[...]

***locationlist\_requestProcessor.java:***

[...]

// determine geo coordinates out of address id and search by geo locations

else if ((addressID != LocationVO.NO\_VALUE\_AVAILABLE) && (distance != LocationVO.NO\_VALUE\_AVAILABLE)) {

GeoPositionVO geoPosition = service.getGeoPosition(logName, addressID);

locations = service.getLocations(logName, xs.getKeyAccID(), scID, geoPosition, distance, force\_hit, onsite\_only, onsite\_wlan\_only);

}

[...]

***LocationService.java:***

**/\*\***

**\***

**\*** **Gets** **geo** **position** **from** **user's** **address.**

**\***

**\*** **@param** *aManagerName* - the connection manager

**\*** **@return** GeoPosition - found geo position

**\*** **@throws** *IciException* - could not get geo position properly

**\***

**\*/**

public GeoPositionVO getGeoPosition(String aManagerName, int anAddressID) throws IciException;

***DefaultLocationService.java:***

**/\*\***

**\***

**\*** **Gets** **geo** **position** **from** **user's** **address.**

**\***

**\*** **@param** *aManagerName* - the connection manager

**\*** **@return** GeoPosition - found geo position

**\*** **@throws** *IciException* - could not get geo position properly

**\***

**\*/**

@Override

public GeoPositionVO getGeoPosition(String aManagerName, int anAddressID) throws IciException {

IConnectionManager manager = getConnectionManager(aManagerName);

manager.doLogText(this, "getting geo position from ICI ...", Location.INFO);

GeoPositionVO geoPosition = null;

try {

UserAddress userAddress = new UserAddress(manager, anAddressID);

Location location = Location.getGeoCoordinates(userAddress.getStreet(), userAddress.getZip(), userAddress.getCity(), userAddress.getIsoCountry(), "google");

if ((location != null) && (location.getLatitude() != Location.NO\_VALUE\_AVAILABLE) && (location.getLongitude() != Location.NO\_VALUE\_AVAILABLE)) {

geoPosition = new GeoPositionVO(location.getLatitude(), location.getLongitude());

}

}

catch (DBReadException anException) {

String message = "unable to read user's address (" + anAddressID + ").";

int errorCode = manager.isSystemOverloaded() ? ErrorCodes.SYSTEM\_OVERLOAD : ErrorCodes.DATABASE\_ERROR;

manager.doLogException(this, anException, message, LocationVO.FATAL);

throw new IciException(errorCode, message, anException, LocationVO.FATAL);

}

catch (IOException anException) {

String message = "unable to determine geo position out of address (" + anAddressID + ").";

manager.doLogException(this, anException, message, LocationVO.FATAL);

throw new IciException(ErrorCodes.INVALID\_USER\_ADDRESS\_DATA, message, anException, LocationVO.FATAL);

}

finally {

manager.doClose(this);

}

return geoPosition;

}

## A.6. Quellcodeauszug der Klasse CWOrderWizardScreenStoreSelectionRadius.cpp

//------------------------------------------------------------------------------

/\*\*

\* Konstruktor

\*/

//------------------------------------------------------------------------------

CWOrderWizardScreenStoreSelectionRadius::CWOrderWizardScreenStoreSelectionRadius(QWidget \* theParent, CWOrderWizard \* theWizard)

: QWidget(theParent)

, myMapLabel(NULL)

, myLocationListBox(NULL)

, myRadiusComboBox(NULL)

, myOrderWizard(theWizard)

{

createGUI();

}

//------------------------------------------------------------------------------

/\*\*

\* Holt eine neue Filialliste.

\* Dieser Slot wird angesprungen, wenn der "Suchen"-Button geklickt wurde.

\*/

//------------------------------------------------------------------------------

void

CWOrderWizardScreenStoreSelectionRadius::updateLocationListAndMap()

{

// Ermitteln der momentanen Filialauswahl bzw. der Stammfiliale

CWStore \* store = CWStoreList::instance() ->findStore(CWStore::currentStore());

QString lastLocationId = store ? store->locationId() : CWCustomer::instance()->get(Customer\_Field\_LastLocationId);

bool success = loadLocations();

if (success) {

fillLocationList(lastLocationId);

loadMap();

}

}

//------------------------------------------------------------------------------

/\*\*

\* Lädt alle Filialen, die sich in einem bestimmten Umkreis der Kundenadresse

\* befinden.

\*/

//------------------------------------------------------------------------------

bool

CWOrderWizardScreenStoreSelectionRadius::loadLocations() const

{

QString addressId = CWCustomer::instance()->billingAddressId();

QString radius = myRadiusComboBox->currentText();

radius.chop(3);

return CWNetworkingController(CWNetworkingFactory:: createLocationsByAddressIdAndRadiusAction(addressId, radius)).run();

}

//------------------------------------------------------------------------------

/\*\*

\* Füllt die Auswahlliste mit den Informationen der Filialen.

\*/

//------------------------------------------------------------------------------

void

CWOrderWizardScreenStoreSelectionRadius::fillLocationList(const QString & theLastLocationId)

{

CWStoreList \* locationList = CWStoreList::instance();

myLocationListBox->clear();

if (locationList == NULL || locationList->count() == 0) {

myLocationListBox->addItem(noStoreText());

} else {

myLocationListBox->addItems(locationList->contentWithWordWrap());

}

myLocationListBox->reset();

myLocationListBox->setCurrentRow(0);

myLocationListBox->repaint();

myLocationListBox->update();

// Markieren der vorherigen Filialauswahl bzw. der Stammfiliale

CWStore \* location = CWStoreList::instance() ->findStoreById(theLastLocationId);

if (location) {

QList<QListWidgetItem\*> items = myLocationListBox ->findItems(location->toStringWithWordWrap(), Qt::MatchExactly);

if (!items.isEmpty()) {

myLocationListBox->setCurrentItem(items.first());

}

}

}

//------------------------------------------------------------------------------

/\*\*

\* Lädt eine Karte, in der die Filialen markiert sind.

\*/

//------------------------------------------------------------------------------

void

CWOrderWizardScreenStoreSelectionRadius::loadMap()

{

QSize size = myMapLabel->size();

QString url = QString("http://www.mapquestapi.com/staticmap/v4/getmap?key=[KEY]&")

+QString("size=%1,%2").arg(size.width()).arg(size.height())

+QString("&declutter=true&scalebar=false&pois=");

for (int i=0; i<locationList->count(); i++) {

url.append("red\_1,"+locationList->value(i)->latitude() +","+locationList->value(i)->longitude()+"|");

}

url.chop(1);

CWHttpOperation operation = CWHttpOperation::create(QUrl(url));

operation.setHeader("Referer", "http://www.cewe.de/");

QFileInfo targetFile(QDir(CWConfigBasic::instance()->temporaryDirectory()), "StoreSelectionMap.jpeg");

bool ok = operation.downloadToFile(targetFile);

if (ok) {

QPixmap mapPixmap = QPixmap(targetFile.filePath());

myMapLabel->setPixmap(mapPixmap);

}

}

//------------------------------------------------------------------------------

/\*\*

\* Speichert die aktuelle Auswahl des Benutzers.

\*/

//------------------------------------------------------------------------------

bool

CWOrderWizardScreenStoreSelectionRadius::atNext()

{

CWStore::setCurrentStore(selectedItem());

return true;

}

## A.7. Quellcodeauszug der Integration des XTCI-Requests

***CWXTCIRequestBuilder.cpp:***

//------------------------------------------------------------------------------

/\*\*

\* Baut den Request für die Liste der Filialen in einem bestimmten Umkreis der

\* Kundenadresse.

\* @param theAddressId Die ID der Kundenadresse in dessen Umkreis die Filialen

\* benötigt werden

\* @param theRadius Die Größe des Umkreises

\*/

//------------------------------------------------------------------------------

QString

CWXTCIRequestBuilder::buildLocationListAddressIdAndRadiusRequest(const QString & theAddressId, const QString & theRadius)

{

QDomDocument doc = buildLocationListBaseRequest(LocationListRequestName);

QDomElement root = doc.documentElement();

Q\_ASSERT\_X(!root.isNull(), "CWXTCIRequestBuilder::buildLocationlistAddressIdAndRadiusRequest", "Kein Dokumentelement");

buildDomElement(root, "address\_id", theAddressId);

buildDomElement(root, "distance", theRadius);

buildDomElement(root, "force\_hit", 1);

QString retVal = prependXMLAndDTDHeader(LocationListRequestName, doc.toString());

return retVal;

}

***CWNetworkingFactory.cpp:***

//------------------------------------------------------------------------------

/\*\*

\* Wir wollen alle Filialen in einem bestimmten Umkreis der Kundenadresse haben.

\* @param theAddressId Die ID der Kundenadresse in dessen Umkreis wir die Filialen

\* haben wollen

\* @param theRadius Die Größe des Umkreises

\* @return Eine CWNetworkingAction die uns alle Filialen in dem Umkreis der

\* Kundenadresse holt.

\*/

//------------------------------------------------------------------------------

CWNetworkingAction \*

CWNetworkingFactory::createLocationsByAddressIdAndRadiusAction(const QString & theAddressId, const QString & theRadius)

{

return new CWNetworkingActionLocationList(CWXTCIRequestBuilder:: buildLocationListAddressIdAndRadiusRequest(theAddressId, theRadius));

}