Entwicklung Mobiler Applikationen

Roland Hediger

31. März 2014

Inhaltsverzeichnis

Teil I.

Theorie

1. Applikationsmodell

- Applikation hat verschiedene Zustände
- Spielt zentrale Rolle
- OS hat kontrolle über Zustandsübergänge
- **Zustandswechsel ruft Callback Funktion auf innerhalb der App. App kann fur eine kurze Zeit darauf reagieren**

  Arbeit innerhalb von Callback sollte möglichst kurz gehaölten werden, App kann möglicherweise abgeschossen werden falls es zu lange dauert.

1.1. Applikation Lebenszyklus Begriffe

tombstoning Die Applikation wird durch eine Userinteraktion verlassen oder durch ein Ereignis unterbrochen (z.B. eingehender Telefonanruf). Das OS verwaltet Zustandsinformationen der App. Wenn die Kontrolle zurückkehrt übergibt das OS der App diese Zustandsinfos wieder damit sie ihren Zustand wiederherstellen kann.

page state Der visuelle Zustand einer Seite. Z.B. Scrollposition eines Textes oder der Inhalt eines TextfeldesDie App kontrolliert selbst die page states ihrer eigenen Seiten.

application state Daten die von mehreren pages einer App benötigt werden, z.B. wie viele Kärtchen schon erfolgreich gelernt wurden und wie viele man nicht wusste. HINWEIS: diese Daten nicht erst beim Verlassen der Applikation speichern, sondern bei jedem Seitenwechsel. Damit wird die Menge der zu speichernden Daten klein gehalten und bei einem allfälligen tombstoning kann die App schnell reagieren.

persistent data: Daten die von allen Instanzen einer Applikation geteilt werden und die z.B. auch ein Reboot des Phones überleben sollen. Beispiel: die heruntergeladenen Karteikasten mit all ihren Kärtchen.

transient data: Daten die beim Tombstoning vom System gerettet werden und bei einer Auferstehung der App zur Wiederherstellung übergeben werden. Diese Daten überleben aber ein Reboot nicht. Beispiel: Daten aus Formularfelder die noch nicht gespeichert wurden. Damit kann nach einem Telefonanruf das Formular wieder gefüllt werden und die Benutzerin merkt nicht, dass die Applikation zwischenzeitlich tot war.

1.1.1. Zustandsübergänge

launching: Die App wird neu gestartet. Sie sollte so bald wie möglich angezeigt werden um die Benutzerin nicht lange warten zu lassen. Also keine persistenten Daten laden in dieser Zeit. Wie und wann denn sonst?

closing : die App wird beendet. Sämtliche noch nicht gespeicherte Daten sollten jetzt persistiert werden. Keine transienten Daten speichern, da die App nur über ein launching wieder gestartet werden kann.

deactivating: tombstoning der App. Transiente und persistente Daten speichern. Letzteres deshalb, weil es keine Garantie für eine Auferstehung gibt.

activating: es findet eine Auferstehung statt. So schnell wie möglich visuellen Zustand wiederherstellen. Ein Neustart und eine Auferstehung einer App sollen/können sich voneinander unterscheiden!

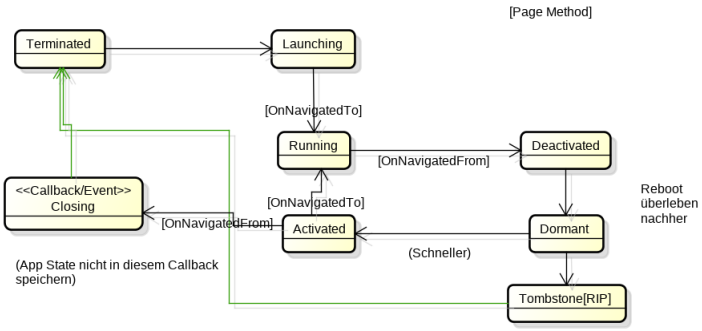


Abbildung 1.1.: figure

1.1.2. Aufbau einer App

Verzeichnisstruktur des Projektes

Properties: WMAppManifest.xml legt Capabilities, Startseite, App Icon und Hintergrundbild fest

References: DLLs welche verwendete Klassen enthalten

Images: Enthält Icons und statisches Bildmaterial

Views, ViewModels, Models: Enthalten Klassen der entsprechenden Namespaces. Siehe MVVM

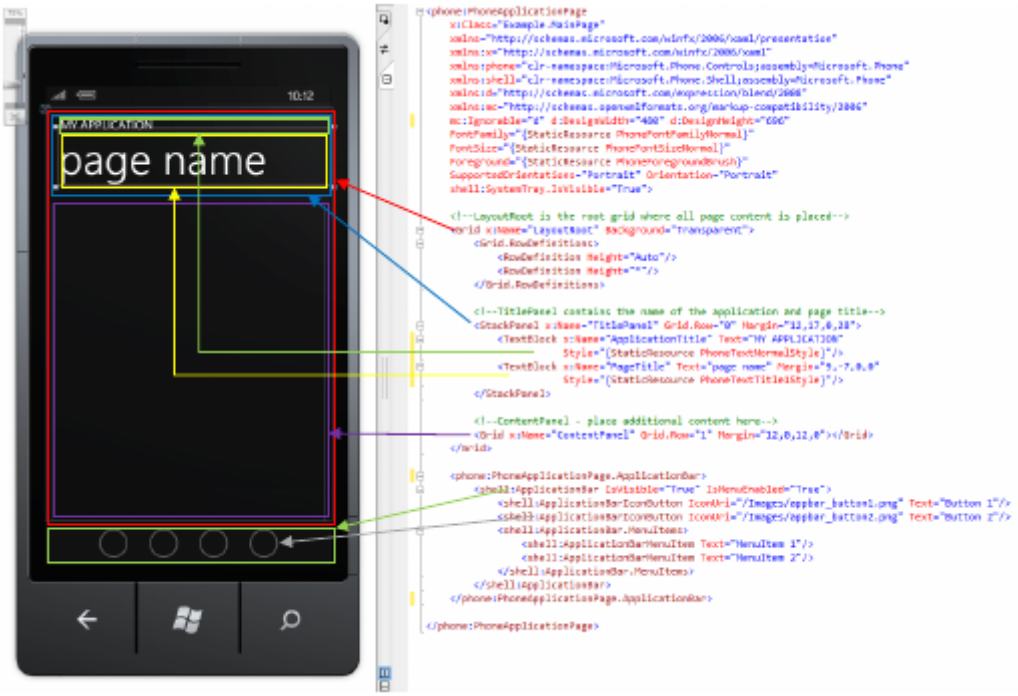
Data: Statische Daten

App.xamls(.cs)

Reprsentiert die Appliktaion als Ganzes, henthält Handler-Implementation für die Application-Lifeclycle Events.

MainPage.xaml(.cs)

Die Startseite muss nicht MainPage heissen, sie muss aber im File WMAppManifest.xml festgelegt werden



1.1.3. XAML Guide

x:Class voll qualifizierter Klassenname (für C# und CodeBehind)

xmlns:x xaml Namespace

xmlns:phone, xmlns:shell C# Namemspaces (ähnlich einem import) d (designer) und mc (markup compatibility) sind Namespacedeklarationen die von Designwerkzeugen wie Blend oder VisualStudio verwendet werden. FontFamily, FontSize und Foreground: Einstellungen für die ganze Seite. Diese Einstellungen werden an die inneren Elemente „vererbt“, müssen dort also nicht erneut gesetzt werden. Sie können allerdings überschrieben werden. erneut gesetzt werden. Sie können allerdings überschrieben werden. SupportedOrientations und Orientation: Geben die möglichen und die Default-Orientierung an.

SystemTray: gibt an ob die Icons des System Trays (Batteriestand, Empfangsqualität, ...) angezeigt werden.

Grid: das äussere Grid enthält die eigentlich interessanten Elemente der Seite. Es ist in zwei Zeilen aufgeteilt, wovon die erste automatisch die benötigte Höhe annehmen soll (Auto) und die zweite den Rest belegen darf (*).

StackPanel: ein weiterer Container für GUI-Elemente Details zur Verwendung von StackPanel und Grid finden Sie im Petzold-Buch Seiten 195, bzw. 228. ApplicationBar: wird unten eingeblendet und kann maximal 4 Buttons enthalten. Zudem werden an ihrem rechten Rand Auslassungspunkte (...) angezeigt. Tippt man auf diese erscheint ein allfälliges Menu mit den beschriebenen MenuItems. Details zur Verwendung der ApplicationBar finden Sie im Petzold-Buch im Kapitel 10, ab Seite 232.

1.1.4. Navigation

Seiten stellen die Einheiten der Navigation dar. Es ist nicht möglich in ein Menu oder in eine App hinein zu navigieren. Bei letzterem muss nämlich auch eine Seite angegeben werden (evtl. auch implizit eine Startseite). Prinzipiell sollte in eine mobile App so wenig Information wie möglich und so viel wie gerade nötig anzeigen. Zudem sollten Informationen sinnvoll auf Seiten verteilt werden. Es ist daher wichtig, die Navigation auch so einfach wie möglich zu halten und wildes herumspringen zwischen Seiten sollte vermieden werden. In allen mobilen OS hat sich eine baumartige Navigation mit dem Back-Konzept ähnlich zu Webbrowsern als die beste Variante durchgesetzt. Entweder muss die Back-Navigation manuell in die App eingebaut werden (iPhone, J2ME) oder sie ist fester Bestandteil des GUI-Konzeptes (Android und WP7 mit Back-Button). In WP7 erhält man den Wechsel von Seiten durch zwei Events angezeigt. Jede Seite sollte daher die OnNavigatedTo - und OnNavigatedFrom -Events implementieren. Sie beim Verlassen auf der alten bzw. beim Eintreten auf der neuen Page ausgelöst. Der übergebene Eventparameter enthält ein Property Content. Dies ist die Zielpage! Nun ergeben die beiden Event-Handler Sinn:

Listing 1.1: Navigation

```
protected override void OnNavigatedFrom(NavigationEventArgs e) {
  if (e.Content is QuestionPage)
    (e.Content as QuestionPage).Cardbox = this.cardbox;
  base.OnNavigatedFrom(e);
}

//Beim Eintreffen auf der Target-Page kann die OnNavigatedTo-Methode
   notwendige Initialisierungen
//vornehmen.
protected override void OnNavigatedTo(NavigationEventArgs e) {
  DataContext = cardbox.CurrentCard;
}
```

1.2. Data Binding

XAML ist keine Programmiersprache, es ist nicht möglich Programmlogik in XAML zu schreiben. Dazu sind die *.xaml.cs-Files da. Da steckt der sogenannte „Code Behind“ drin. C# kennt das Konzept von partiellen Klassen, d.h. eine Klasse muss nicht vollständig in einem File programmiert werden. Dieses Feature wurde mit XAML eingeführt, denn es ermöglicht es, die XAML-Seiten in C#-Code umzuwandeln. Dies ist die eine Hälfte der partiellen Klassen.

Die andere Hälfte sind die eben erwähnten *.xaml.cs-Files. Um die View also nicht durch den Code Behind abfüllen zu müssen wurden sogenannte Bindings entwickelt. Damit ist es möglich, Daten aus XAML-Files zu referenzieren, die unterschiedlichen Ursprung haben. Z.B. in einem anderen (Teil des) XAML-File (StaticBinding) oder aus einer anderen Klasse (Binding). Dabei kann XAML nur auf Properties zugreifen. Will man, dass der gebundene Wert bei einer Änderung auch im GUI angezeigt wird, so benötigt man sogar Dependency Properties (siehe Petzold-Buch Kapitel 11, S. 296). Beispiel:

Listing 1.2: Data Binding Example

```
<TextBlockx:Name="LastAccess" Text="{Binding LastAccessed}"
    Style="{StaticResource contentStyle}"/>
```

In diesem Textblock wird das Datum der letzten Änderung einer Kartei angezeigt. Das Attribut x:Name deklariert einen Namen, der im Code Behind File dann als Property der Page-Klasse zur Verfügung steht. Der Text wird von einem Property namens LastAccessed geholt. Doch auf welchem Objekt ist dieses Property verfügbar? Dies wird im Property DataContext angegeben, die jede XAML-„Klasse“ besitzt.

1.2.1. Bemerkungen für Entwicklung

Global App.cs variablen benötigt weil im WP7 keine Objekten zwischen Views übertragen werden können. Code behind sollte schlank gehalten werden.

2. MVVM

2.1. Einführung

Das Model-View-Controller (MVC) Architekturmuster spielt bei der Programmierung aller mobiler Apps eine wichtige Rolle. Zudem werden GUIs meist deklarativ definiert. Mit Hilfe eines GUI-Designer-Tools werden die einzelnen Seiten einer App entworfen und in einem speziellen XML-Format oder sonst einem proprietären Format gespeichert. Eine andere Form von Apps sind Browser-basierte Apps, d.h. es sind eigentlich Webapplikationen die mit HTML5, CSS3 und Javascript implementiert werden. Viele Apps – auch wenn sie native implementiert sind – gleichen ohnehin Rich-Internet-Applikationen (RIAs). Denn meistens ist eine App nur ein hübsches, mobiles Front-End für eine Server-basierte Anwendung (Beispiele: SBB-Fahrplan, Facebook, Kalender, Mail etc.). Browser-basierte Apps werden übrigens immer beliebter. Warum?

Browser - Kein Native Look and feel

- Muss jedes mal geladen werden
- Noch unvollständiger Zugruff auf hw rescourcen
- Webserver
- Einfacher Zahlungsmodelle im App Store
- \+ Befreit von Zertfizierungsprozess.
- \+ Javascript
- \+ Einmal schreiben, leicht updaten.
- \+ Keine Lizenzierungskosten

2.2. MVVM

Der Einsatz des MVC-Prinzips in mobilen Applikationen sollen hier am Beispiel von Phone7 GUI-Technologien erläutert werden. Auf anderen Plattformen mag die GUI-Beschreibungssprache anders aussehen (Proprietär, HTML5, etc.) oder Technologien wie das Data Binding müssen „von Hand“ implementiert werden. Das Prinzip ist aber oft sehr ähnlich. Wenn das GUI hauptsächlich deklarativ entwickelt wird, dann stellt sich die Frage, wo die GUI-Logik implementiert werden soll. Also all die Logik, die auf Benutzereingaben reagiert oder Daten zur Anzeige aufbereitet oder „veredelt“. Microsoft gibt die Antwort indem eine eigene Variante des MVC-Patterns eingesetzt wird. Es ist eine Art doppeltes Observer-Pattern und besteht aus:

View Wird in XAML und dem Code Behind definiert. Das .xaml.cs-File soll dabei möglichst schlank sein und lediglich dazu dienen die Events vom GUI an ein ViewModel weiterzuleiten. Die View ist einzig für die Darstellung zuständig. Diese kann allerdings sehr aufwändig sein, z.B. weil sie Animationen enthält.

ViewModel Die View „denkt“ sie kommuniziere direkt mit dem Modell. Diese Illusion wird durch sogenannte View-Models erzeugt. Sie stehen zwischen der View und dem Model. Sie haben folgende Verantwortlichkeiten:

1. Properties für das Binding in eine View zur Verfügung stellen.
2. Daten aus dem Model in die richtigen Datentypen für die View konvertieren.

Ein ViewModel dient also als Vermittler zwischen der View und dem Modell. Es ermöglicht so die Abkopplung des Models vom Rest der App. Es wird nun möglich, das Modell auf einem Server zu platzieren und das Front-End auf einem Smartphone als App laufen zu lassen.

Model Das Model schliesslich hat dieselben Verantwortlichkeiten wie im ursprünglichen MVC-Pattern mit einer Ausnahme: das Modell kann nicht von sich aus eine Zustandsänderung an die Views notifizieren.

2.2.1. MVC vs MVVM

Komponent	MVC	MVVM
Model	Verschikt updates direkt and der View - Observer zwischen View und Model	Kann View Model bei Callbakcs informieren jedoch nur bei einem Request im Webapps. Falls Model auf Phone ist, sind direkter Updates möglich
View	Stellt nur dat, kann bei Model direkt Zustand abfrage. Kennt Controller nicht.	Stellt nur dar. Verwendet Binding im Daten von Biew Model zu hohlen. View denkt dass das ViewModel sein Model ist.
Controller	verwaltet Input und koordiniert updates der View, leitet Input bei bedarf weiter an Modell.	(View Model) Bereitet die Daten des Modells für die View auf kennt Views nicht . Many to One Beziehung ist möglich zwischen Views und VM sollte aber vermeidet werden.

2.2.2. Alternativen zu MVVM

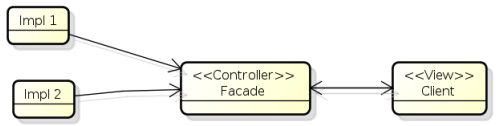


Abbildung 2.1.: Facade

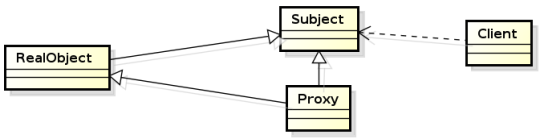


Abbildung 2.2.: Proxy

2.3. MVC in Cardbox

Ein Teil der Lernkartei-App dient im Folgenden zur Illustration, wie das MVVM-Pattern eingesetzt werden kann. Dabei wird vom folgenden Setup ausgegangen:

1. Die Overview-Page enthält eine Liste der lokal (also auf dem Smartphone) gespeicherten Karteikästen (cardbox). Pro Karteikasten wird dessen Titel und Beschreibung angezeigt. Durch Antippen eines Karteikastens wird auf die Cardbox-Page gewechselt. Durch Antippen des Download-Buttons wird auf die Download-Page gewechselt.
2. Die Download-Page lädt beim Betreten die Liste der Karteikästen herunter, die auf dem Server zur Verfügung stehen. Dabei wird pro Karteikasten angezeigt: Titel, Beschreibung, Grösse des Downloads und ob die Kärtchen Bilder enthalten oder nicht (dies in Form eines Icons).
3. Die Cardbox-Page zeigt Details zu einem Karteikasten an. Es wird dargestellt:

   Titel des Karteikasten (als Titel der Page)

   Wann das letzte Mal mit diesem Karteikasten gelernt wurde.

   Wie viele Kärtchen korrekt gelernt wurden und wie viele Kärtchen es insgesamt gibt. Diese Information wird als Text und als Progress-Bar angezeigt.

   Die Beschreibung des Karteikastens.

2.4. MVVM muss das sein? (Denzler)

Es gibt 2 Hauptkritikpunkte am MVVM-Pattern.

1. Die Implementierung ist ziemlich aufwendig. So aufwendig, dass für kleinere Applikationen statt einem Dreizeiler schnell mal das 10-20 fache an Code zu erstellen ist. Dies lohnt sich nicht immer und daher muss der Einsatz von MVVM gut abgewogen werden. Gerade bei mobilen Applikationen ist die zusätzliche Komplexität oft nicht zu rechtfertigen, da die Applikationen eher klein sind. Abhilfe schaffen da wiederum Frameworks (alle nicht von Microsoft). Die helfen einem zwar die Komplexität zu reduzieren, allerdings muss man dann zuerst das Framework beherrschen.

2. Ein bewährter Grundsatz im Software Design ist, das eine Klasse nur einen Grund haben sollte sich zu ändern. Ein View-Model hat wegen der zweifachen Verantwortlichkeit aber zwei Gründe. Damit wird sowohl das Single Responsibility Principle als auch das Prinzip des Separation of Concerns verletzt. Denn ein ViewModel muss angepasst werden, wenn die View oder das Model ändert. Es kann somit zur meistgeänderten Einheit einer Applikation werden.

Weitere Details zur Implementation des MVVM-Pattern auf WP7 können Sie dem Paper: The MVVM Design Pattern von Colin Melia entnehmen (auf dem AD). Das MVVM-Pattern wird von Microsoft stark favorisiert und ist als Reaktion auf die überladenen GUI-Klassen entstanden: Sie erinnern sich: .xaml.cs (also Code-Behind) Files die voll mit GUI-Logik waren und somit GUI und deren Steuerung vermischten. Eine Phone7/8-App muss kein MVVM-Design enthalten. Sich nicht daran zu halten bedeutet aber oft, dass die gesamte Applikationslogik im GUI steckt!

3. Persistenz

3.1. Vorteile/Nachteile Isolated Storage

+ Deinstallation
+ Speicher übersicht (Nutzung)
+ App upgrade
+ App kann nur in eigene Container schaden anrichten.
+ einfache Programmierung
+ Keine Spionierung
- Redundante Daten - mehr Speicher
- Meherere Containers für App Suite
- Spezileele mechanismen für Datenaustauch zwischen apps.

In der Regel sind diese Einschränkungen gar nicht so schlimm wie zunächst angenommen. Auch bei Desktop- oder gar Serverapplikationen wird nur selten in fremde Verzeichnisse geschrieben oder aus ihnen gelesen. Und wenn dies geschieht, dann ist es oft mit einem erhöhten Programmieraufwand verbunden, da dieser gemeinsame Zugriff mit speziellen Schutzmechanismen versehen werden muss.

3.2. Isolated Storage API

3.2.1. Settings

Wie der Name es schon andeutet, diese Variante ist vor allem geeignet um Applikationssettings persistent zu speichern. Oder andere „kleine“ Objekte, bzw. „kleine“ Datenstrukturen. Beim IsolatedStorageSettings-Objekt handelt es sich um ein gewöhnliches Dictionary (zu gut Java: HashMap), welches mit der Methode save() in den persistenten Speicher hinein serialisiert wird. Es ist daher nötig, dass alle Objekte die so gespeichert werden sollen, auch serialisierbar sind. D.h. sie müssen entweder das ISerializable implementieren oder durch das Attribut [Serializable] markiert werden.

Listing 3.1: Isolated Storage Settings

```
// appSettings is a Dictionary that can be persisted
private IsolatedStorageSettings appSettings =
    IsolatedStorageSettings.ApplicationSettings;
// remember email address
appSettings.Add("email", "mad@fhnw.ch");
// retrieve email
tbEmail.Text = (string)appSettings["email"];
// change email
appSettings["email"] = "emoba@fhnw.ch";
// delete email
appSettings.Remove("email");
//Wird die Applikation deaktiviert oder beendet, so ruft WP7 automatisch
    die save()-Methode auf den
//ApplicationSettings auf. Dies kann man selbstverstaendlich jederzeit auch
    selbst tun:
// persist application settings
appSettings.Save();
```

3.2.2. Files

Sollen umfangreichere Daten gespeichert werden, so steht das Filesystem zur Verfügung. (Für .Net-Kenner: Es handelt sich dabei um denselben IsolatedStorage, der auch sonst bei Silverlight auf dem Browser zur Verfügung steht). Auf WP7 sind keine Quotas aktiviert, d.h. jede App kann theoretisch den ganzen verfügbaren Speicher belegen. Es lassen sich Files und Directories erstellen und letztere können beliebig verschachtelt werden. Es stehen Schreib-, Lese-, Update- und Löschoperationen zur Verfügung. Files und Directories können auch mit Wildcards gesucht werden. Siehe dazu auch How to: Perform Isolated Storage Tasks unter: http://msdn.microsoft.com/en-us/library/cc265154(v=VS.95).aspx

Listing 3.2: File Storage

```
public void StoreX(X someObject, string dirname, string filename) {
  IsolatedStorageFile storage =
     IsolatedStorageFile.GetUserStoreForApplication();
  storage.CreateDirectory(dirname);
  using (IsolatedStorageFileStream file =
     storage.CreateFile(Path.Combine(dirname, filename))) {
    XmlSerializer xml = new XmlSerializer(typeof(X));
    xml.Serialize(file, someObject);
  }
}
public X LoadX(string dirname, string filename) {
  X result = null;
  IsolatedStorageFile storage =
     IsolatedStorageFile.GetUserStoreForApplication();
  IsolatedStorageFileStream file = null;
  try {
    file = storage.OpenFile(Path.Combine(dirname, filename), FileMode.Open);
    XmlSerializer xml = new XmlSerializer(typeof(X));
    result = (X)xml.Deserialize(file);
  } finally {
    if (file != null) {
      file.Close();
      file.Dispose();
    }
  }
  return result;
}
```

Zu beachten ist, dass die XML-Serialisierung nur public Felder oder Properties speichern kann. Private oder protected Variablen werden also nicht gespeichert. Eine XML-Serialisierung ist natürlich speicherhungriger als eine binäre Serialisierung. Trotzdem bietet sie einige Vorteile, die ihren Einsatz rechtfertigen:

- Weniger Versionierungskonflikte
- geeignet für die Entwickler (Menschlich Lesbar)
- Gute XML Tools verfügbar.

3.3. Datenbank

Datenbank-Unterstützung gibt es in WP7 erst seit Mango (also 7.1 bzw. 7.5). Es handelt sich um eine SQL Server Compact Edition DB. Natürlich befinden sich auch Datenbanken in den Isolated Storages der Apps und unterliegen somit denselben Restriktionen wie auch alle anderen Files. Eine Datenbank steht nur einer App zur Verfügung und läuft auch nur in deren Prozess. D.h. wenn die App beendet wird, läuft auch die DB nicht weiter. Zudem ist der Zugriff auf die DB nur mittels LINQ to SQL möglich (siehe Modul dnfc). In Android ist der Zugriff auf eine SQLite DB sogar aus mehreren Apps möglich. Allerdings handelt es sich um einfache dafür auch effiziente Datenbanksysteme. Die DB befindet sich üblicherweise in einem File und wird über Libraries angesprochen. Diese Libraries werden direkt mit dem Code der App verlinkt. Es existiert also kein Datenbankserver, es fehlt meist die Unterstützung für Transaktionen.

3.4. Deployment Hinweise bezüglich Speichern

- Dafür sorgen dass Kompatabilität zwischen Files von den alten Version und den neuen Version existiert.

4. Networking

4.1. Anbindung ins Internet

```
1 <Image Source="http://www.charlespetzold.com/Media/HelloWP7.jpg" />
```

Probleme:

- Statische Resource in Projekt kopieren lieber, Lange Ladezeiten für externe Resourcen.
- Fehlerbehandelung bei nicht vorhandene URI.
- Flugzeugmodus.
- Datenkosten
- Code Updates

Während die technischen Hürden für einen Internetzugriff nahezu abgebaut sind, spricht trotzdem vieles für eine saubere Architektur des Systems. Es macht Sinn den Netzwerkzugriff auf ein Modul/eine Komponente des Systems zu beschränken. Schliesslich handelt es sich bei jedem Netzwerkzugriff um eine Aussenschnittstelle, die im Interesse der Endbenutzer wie auch der Entwickler dokumentiert sein sollte. Best Practices bei Verwendung von Netzwerkverbindungen:

-
- Auf wenige Stellen im Code beschränken, in einem Modul bündeln.
- Wenn immer möglich asynchron, damit das Gerät nicht blockiert ist für die Dauer der Verbindung.
- Keine Annahmen über Bandbreiten oder Verfügbarkeit treffen.
- Fehlertoleranz einbauen

  App kann auch ohne Internetverbindung sinnvoll eingesetzt werden

  Default-Daten stehen lokal zur Verfügung

  Die zuletzt gefundenen Daten werden lokal gespeichert. Schlägt Verbindung fehl werden diese angezeigt (mit entsprechender Bemerkung natürlich)

PS: in Petzold's Buch wird ab Seite 70 beschrieben, wie man Bitmaps aus dem Code herunterlädt, damit man nicht in XAML eine URL angeben muss

4.2. WebClient Klasse

Listing 4.1: Webclient

```
WebClient web = new WebClient();
web.AllowReadStreamBuffering = false;
web.OpenReadCompleted += new
    OpenReadCompletedEventHandler(DownloadCompleted);
4 web.OpenReadAsync(new
    Uri("http://web.fhnw.ch/plattformen/mad/flashcards/overview.xml"));
```

1. Zunächst wird ein WebClient -Objekt erzeugt
2. Daten werden nicht zuerst lokal gepuffert, siehe unten.

3. Danach wird dessen OpenReadCompleted -Delegate mit einem Handler initialisiert, der aktiviert wird, wenn der Download beendet wurde.

4. Zuletzt muss der Download gestartet werden. Damit dies asynchron abläuft, wird die Methode OpenReadAsync verwendet

Wenn der Download mit der Methode CancelAsync() auf dem WebClient abgebrochen wurde. Auch in diesem Fall wird das Ereignis ausgelöst! In Abhängigkeit von der AllowReadStreamBuffering -Eigenschaft des WebClients; wurde dieses Flag auf

- true gesetzt (Default), dann wird der gesamte Download in einen Puffer zwischengespeichert und erst dann das OpenReadCompleted-Ereignis ausgelöst.
- false gesetzt, dann wird das Ereignis ausgelöst sobald Daten anstehen , auch wenn noch nicht alles heruntergeladen wurde. Dies ist auf dem Phone von Interesse, denn es verhindert ein allzu grosses Speicherprofil der App.

4.2.1. OnCompleted Delegate

Listing 4.2: OnCompleted Webclient (Stream Copy)

```
public void DownloadCompleted(object sender, OpenReadCompletedEventArgs
    args) {
    if (!args.Cancelled && args.Error == null)
    byte[] buffer = new byte[4096];
    using (IsolatedStorageFileStream file = storage.CreateFile(targetPath)) {
     StreamUtils.Copy(args.Result, file, buffer);
     }
}
```

5. Tasks (Datenaustausch zwischen Apps)

Ein typisches Beispiel für einen Datentransfer zwischen Apps ist, eine Aufnahme von der Kamera in die eigene App zu transferieren. Aus Sicherheitsgründen kann unter WP7 die Kamera nicht direkt in eine App eingebunden werden (in Android ist dies viel besser möglich). Wie geht man vor?

- Die eigene App muss die Kontrolle an die Kamera abgeben. D.h. die eigene App wird deaktiviert (Tombstone-Zustand)
- Die Kamera-App startet. Damit ist garantiert, dass Bilder immer mit derselben App geschossen werden. Eine Fehlbedienung wird dadurch eher unwahrscheinlich.
- Die Benutzerin löst aus. Die Kamera-App beendet sich und
- gibt die Kontrolle an die frisch reaktivierte eigene App zurück.

Listing 5.1: Kamera Beispiel

```
public partial class MainPage : PhoneApplicationPage {
  CameraCaptureTask camera = new CameraCaptureTask();
  // Constructor
  public MainPage() {
    InitializeComponent();
    camera.Completed += AquirePicture;
  }
  void AquirePicture(object sender, PhotoResult args) {
    if (args.TaskResult == TaskResult.OK) {
      BitmapImage bmp = new BitmapImage();
      bmp.SetSource(args.ChosenPhoto);
      img.Source = bmp;
    }
  }
  private void ManipulationStarted(object sender,
     ManipulationStartedEventArgs e) {
    camera.Show();
    e.Complete();
    e.Handled = true;
    base.OnManipulationStarted(e);
  }
}
```

5.1. Tombstoning und Tasks

die App wurde ja beendet und neu gestartet. Es kann also nicht sein, dass einfach die Callback-Methode aufgerufen wird, denn bei einem Neustart wird die App ja zuerst initialisiert.“ Stimmt! Deshalb ist die offizielle „Best Practice“ folgende:

- Den CaptureTask als Attribut in der Klasse definieren. Dann steht sie nämlich sowohl dem Konstruktor als auch einem weiteren ManipulationStarted-Ereignis zur Verfügung.
- Die Callback-Methode im Konstruktor registrieren und zwar so spät wie möglich. Warum? Weil nämlich beim Neustart der App, unmittelbar nach dem registrieren der Callback-Methode diese auch aufgerufen wird. Wird nun also die Registrierung nicht im Konstruktor gemacht, sondern z.B. lokal im ManipulationStarted-Event,

so würde nach einer Rückkehr von der Kamera, die Callback-Methode nicht aufgerufen! Wird die Callback-Methode zu früh im Konstruktor aufgerufen, dann läuft man in Gefahr, dass die App noch nicht komplett initialisiert ist wenn die Callback-Methode läuft

6. Push Notifications

Push Notifications sind kurze Meldungen die von einem zentralen PN-Server (Apple, Microsoft, Google) an ein Handy geschickt werden. Diese Meldungen können jederzeit empfangen und verarbeitet werden. Je nach OS haben sie unterschiedliche Ausprägungen und Eigenschaften. Push Notifications werden vom mobilen OS empfangen und an die adressierte Empfänger-App weitergeleitet – auch wenn diese App momentan nicht läuft! Typische Anwendungsgebiete für Push Notifications sind:

- E-Mail (Push-Mail Funktionalität)
- Chat, IM, Internettelefonie
- Ticker (Wetter, News, Börsenkurse)
- Tasks, Todos mit Alarmfunktion
- Social-Media (Status Updates)
- E-Shopping (Order state / tracking)

6.1. Notification Typen

Üblicherweise sind den Meldungen die verschickt werden können enge Grenzen gesteckt: maximale Downloadzeit und maximale Grösse werden von den Betreibern der PN-Server festgelegt. In WP7 gibt es z.B. drei verschiedene Typen von Push Notifications:

Raw Notifications:

- Können nur empfangen werden, wenn die App läuft.
- Können für alles Mögliche eingesetzt werden, aber ihre Grösse ist auf 1kB beschränkt.
- Triggern keine audiovisuellen Elemente, ausser die empfangende App ist entsprechend programmiert.

Toast Notifications:

- Enthalten nur Text der in einen Betreff und einen Inhalt eingeteilt sind.
- Falls die empfangende App nicht läuft wird der Inhalt in einem kleinen Fenster am oberen Rand des Bildschirmes dargestellt. Dieses Notifikationsfenster überlagert sämtliche anderen Inhalte.
- Falls die empfangende App nicht läuft erfolgt keine audiovisuelle Reaktion, ausser die App ist entsprechend programmiert

Tile Notifications

- Führt ein Update des Live Tile der empfangenden App durch falls sich dieses auf dem Startscreen befindet. Dabei kann die Grafik, der Text und der Zähler des Tiles verändert werden.
- Werden auch empfangen wenn die App läuft. Vorsicht: Tiles können dann nicht mehr mit dem Zustand der App übereinstimmen.
- Müssen kleiner als 80kB sein und in weniger als 15 Sekunden herunter geladen werden.

6.2. Funktionsweise

Um Push Notifications verschicken zu können sind mehrere Komponenten notwendig:

1. Eine WP7 App
2. Ein Webservice der die zu versendenden Daten liefert
3. Der Push Notification Service (PNS)
4. Ein Push Client (Komponente von WP7)

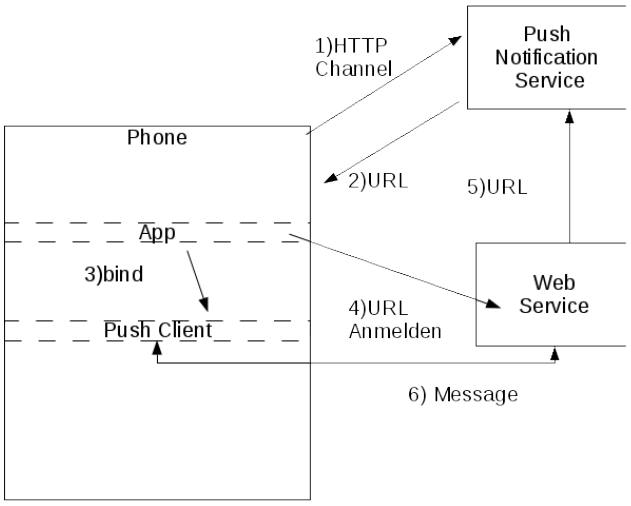


Abbildung 6.1.: figure

6.2.1. Schritte

Welche Schritte sind nun notwendig um auf eine Push Notification reagieren zu können?

1. Wenn die App (I) gestartet wird, muss sich einen HttpChannel öffnen und dem PNS (III) mitteilen, dass sie bereit ist Push Notifications zu empfangen.
2. Der PNS(III) liefert eine URL zurück, welche die App (I) zwischenspeichert.
3. Die App (I) teilt dem Push Client (IV: der in WP7 auf dem Phone zur Verfügung steht) mit, welche Art von Push Notification sie erwartet.
4. Die unter 2. empfangene URL wird an den Webservice (II) verschickt.
5. Der Webservice (II) verwendet diese URL um Notifikationen via den PNS(III) zu verschicken.
6. Der PNS(II) identifiziert das Phone anhand der URL und verschickt die Notification an den Push Client (IV).

7. Der Push Client (IV) leitet die Notifikation gemäss ihres Typs weiter.

8. Wenn es eine Tile- oder Toast-Notification war, dann öffnet ein antippen des Tiles oder des Toastes die App (I)

Einfach, nicht?!

Der PNS liefert die Notifikationen die aus verschiedenen Quellen stammen können an den Push Client in Paketen so bald wie möglich. Was „so bald wie möglich“ genau bedeutet ist nicht festgelegt. Ja, es gibt KEINE GARANTIE, dass Notifications überhaupt ausgeliefert werden. D.h. eine App sollte immer so gestaltet sein, dass sie auch ohne Push Notifications auskommt, bzw. dass der Verlust von Push Notifications keine zentrale Funktionalität beeinträchtigt.

6.3. Multitaking

Apple führte Push Notifications mit iOS 3 ein. Oft wurden Push Notifications mit Multitasking in Verbindung gebracht. Natürlich können Push Notifications kein echtes Multitasking ersetzen und dennoch können sie in gewissen Fällen wie ein Hintergrundtask agieren. Wie geht das? Ein Problem mit den ersten iPhones war, dass nur wenige Apps in der Lage waren asynchron benachrichtigt zu werden. Es waren dies hauptsächlich die E-Mail-App. Es war also nur den Apple iPhone-Entwicklern vorbehalten eine App zu einem beliebigen Zeitpunkt über irgendein Ereignis (z.B. das Eintreffen neuer Mails) zu benachrichtigen. Bald schon wuchs der Druck auf Apple dieses Feature auch anderen Entwicklern zugängig zu machen. Entwickler auf anderen mobilen Betriebssystemen hatten diese Probleme nicht. Da Android, Symbian, und Windows Mobile Multitasking kannten, war es ihnen möglich im Hintergrund ständig einen Service laufen zu lassen, der in regelmässigen Abständen ein Polling bei einem Server durchführte. So hatte der Benutzer eines solchen Smartphones den Eindruck, dass die Nachricht auf das Gerät „gepusht“ wurde. Apple iPhone und auch Microsofts Phone 7 haben aber absichtlich kein Multitasking vorgesehen – zumindest nicht für normale Entwickler. Im Kernel existiert auf beiden Systemen Multitasking. Interessanterweise hat nun auch Google seit Android 2.2 Push Notifications im Angebot. Warum denn eigentlich? Android bietet sehr wohl Hintergrund Services an!

6.4. Push vs Polling

+ Batterie

+ Netzwerk

+ Provider PNS kontrolliert Datenverkehr

+ Kein Polling , mehr leisting

- evtl Webservice nötig

- Polling hat sofortige Reaktion be server Down (zuverlässiger)

7. Multitasking

Ein Argument gegen echtes Multitasking ist, dass der beschränkte Bildschirmplatz sowieso nur eine App aufs Mal verträgt. Wenn also das Umschalten zwischen den Apps komfortabel und schnell genug ist, müssen die Apps im Hintergrund nicht unbedingt weiterlaufen. Sie würden meistens nichts tun als auf User-Input warten oder unnötig Rechenleistung verbrauchen um Dinge darzustellen die nicht sichtbar sind. Ist eine App gut auf das Tombstoning vorbereitet, kann sie bei einer Wiederauferstehung genau dort fortfahren, wo sie unterbrochen wurde, bzw. wo sie stünde wenn sie weitergelaufen wäre. Die Benutzerin würde also nicht unterscheiden können ob die App im Hintergrund weiter lief oder stehen blieb. Doch es gibt einige Ausnahmen. Die wichtigste Ausnahme vom Multitaskingverbot betrifft den Kernel des mobilen Betriebssystems (OS) selber. Das OS könnte ohne echtes Multitasking (d.h. mehrere gleichzeitige Prozesse mit getrenntem Adressraum) kaum seine Funktionalität anbieten. Komplette Trennung von Apps oder auch nur die ständige Empfangsbereitschaft für eingehende Telefonanrufe oder SMS wären nicht realisierbar. Für „normale“ Apps stehen ein paar Tricks zur Verfügung um ein echtes Multitasking vorzutäuschen oder teilweise gar zuzulassen.

Fast Application Switching Statt eine App bei einer Unterbrechung wirklich zu beenden und wieder neu zu starten, wird sie einfach angehalten bleibt aber im Hauptspeicher erhalten. Beim Fortfahren entfällt somit eine teure Initialisierung

Scheduled Tasks Tasks können in zwei Varianten im Hintergrund laufen: entweder in regelmässigen Abständen ganz kurz, oder dann wenn das Phone durch den Task nicht belastet wird.

Scheduled Notifications Erlaubt das Anzeigen von Erinnerungen zu bestimmten Zeitpunkten

Background Audio Ermöglicht einer App Musik abzuspielen auch nachdem sie geschlossen wurde.

Background Filetransfer Ein längerer Filetransfer (z.B. Download) wird fortgesetzt, auch wenn die startende App beendet wird.

7.1. Fast Application Switiching

Tombstoning ist ein schwerfälliger Mechanismus, da die Applikation nicht nur angehalten wird. Vielmehr wird der Speicher den die Applikation belegte freigegeben. Dies bedeutet, dass bei einer Wiederauferstehung die Applikation wieder geladen werden muss. Fast Application Switching (FAS) beschleunigt diesen Vorgang, indem die Applikation nur angehalten, deren Speicher aber behalten wird. Bei einer Wiederaufnahme müssen lediglichdie Event-Handler (Activated und OnNavigatedTo) durchlaufen werden. Im besten Falle haben diese überhaupt nichts zu tun. Mit der Einführung von FAS wurde auch das Application Lifecycle Modell angepasst: Trotz FAS ist es nötig ein Tombstoning vorzubereiten. Dies deshalb, weil bei fehlendem Speicher die „schlafenden“ Applikationen (im Zustand Dormant) aus dem Speicher geworfen werden. Falls kein Tombstoning vorgesehen war, fehlt eine sinnvolle Implementation der Activated-Methode und damit kehrt die Applikation unter Umständen nicht in den erwarteten Zustand zurück.

7.2. Scheduled Tasks

Es gibt zwei Arten von Tasks die im Hintergrund ausgeführt werden können:

1. PeriodicTask Aufgaben die periodisch wiederkehrend erledigt werden sollen und wenig Rechenleistung benötigen.
2. ResourceIntensiveTask Aufgaben die keine fixe Periodizität benötigen dafür aber viel Rechenleistung beanspruchen. Tasks werden durch sogenannte Agents ausgeführt (siehe auch die Klasse ScheduledTaskAgent)

Tasks werden durch sogenannte Agents ausgeführt (siehe auch die Klasse ScheduledTaskAgent)

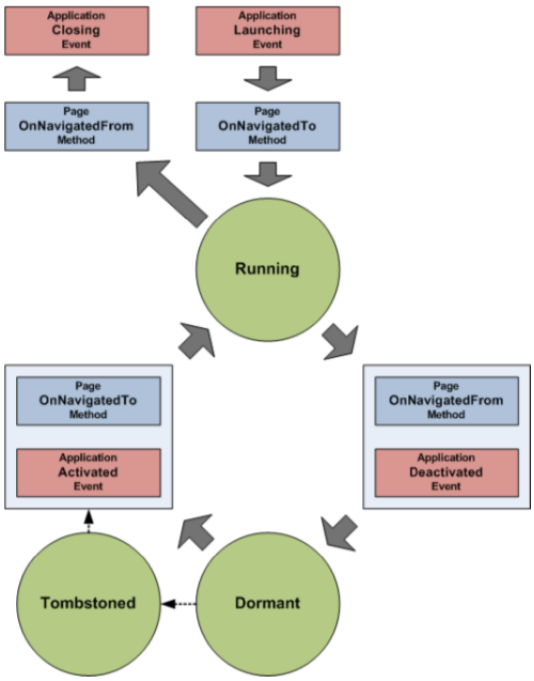


Abbildung 7.1.: figure

7.2.1. Background Agents

Eine Applikation kann nur einen Background Agent haben. Dieser wird entweder als PeriodicTask, Als ResourceIntensiveTask oder als beides gleichzeitig registriert. Wann der Agent ausgeführt wird, hängt von der Art des Agents ab. Der Agent-Code muss in einer Unterklasse von BackgroundAgent implementiert werden. Wenn der Agent gestartet wird, wird seine OnInvoke(ScheduledTask)-Methode aufgerufen. Der Agent kann aufgrund des Parameters ScheduledTask entscheiden, als welcher Task-Typ er gestartet wurde. Wenn der Agent seine Arbeit erledigt hat, muss er dies dem Betriebssystem melden. Er kann dies mittels einer von zwei Methoden tun:

NotifyComplete meldet dem Betriebssystem, dass der Task erfolgreich seine Arbeit erledigen konnte.

Abort meldet dem Betriebssystem, dass der Task seine Arbeit nicht erledigen konnte (z.B. weil eine Netzwerkverbindung nicht aufgebaut werden konnte). Zugleich wird das IsScheduled Property des Tasks auf false gesetzt. Dies bedeutet, dass dieser Task nicht mehr für weitere Ausführungen vorgesehen wird. Zudem kann die Applikation, wenn sie das nächste Mal läuft, dieses Property abfragen um herauszufinden ob ein Problem aufgetaucht ist.

7.2.2. Task Einshränkungen

Typ Unabhängig

- Nicht unterstützte APIs
- Hauptspeicherlimit : 6MB
- Erneuerung alle 2 Wochen : Das Scheduling eines Tasks läuft nach spätestens zwei Wochen ab. Danach muss er neu gescheduled werden. Dies kann natürlich bei jedem Start der Applikation erfolgen.
- Exception Limit : Herrausgeschmissen nach 2 aufeinanderfolgenden Exceptions

Periodische Tasks

- Intervall für Ausführung : 30 min
- Arbeitszeit der Agent : 25 Sekunden
- Keine Tasks in Batteriesparmodus (Optional)

- Limitierte Anzahl Agents : Herrstellerspezifisch : Min # 6

Resource Intensive Einshränkungen

- Dauer der Aerbeit : 10 min.
- Ladegerät notwendig
- WAN / USB Internet notwendig
- Min Batteriezustand : 90% voll
- Task lauft nur bei Screen Lock
- Kein Telefongespräch am laufen.

Kann sein dass solche Resource Tasks nie ausgeführt werden - Nie darauf verlassen.

7.3. Scheduled Notifications

Benachrichtigungen können von einer Applikation im Betriebssystem registriert werden. Sie erscheinen dann zum vorgegebenen Zeitpunkt (auch periodisch falls gewünscht), ohne dass die Applikation am Laufen ist. Es gibt zwei Arten von Benachrichtigungen: Alarms und Reminders. Die Anzeigen sind unten abgebildet.

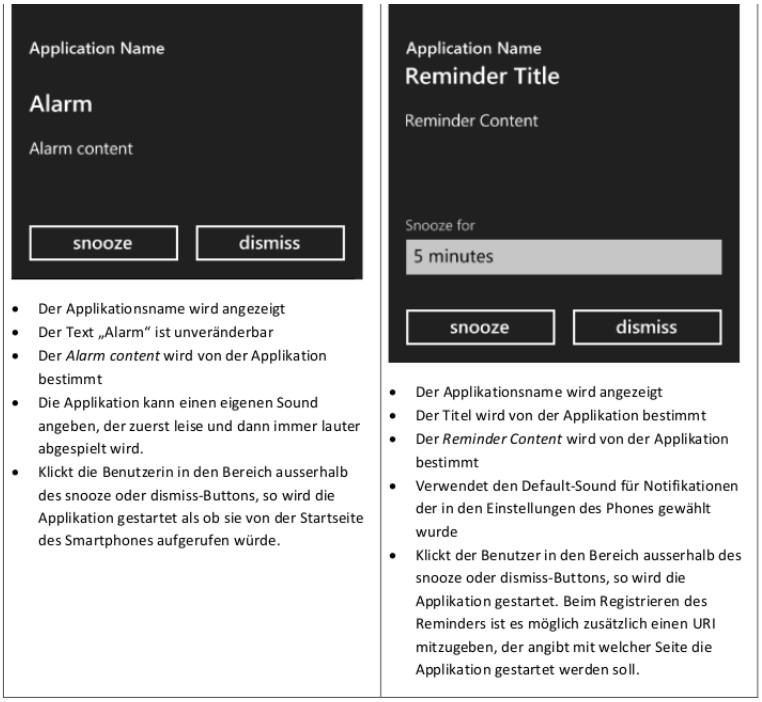


Abbildung 7.2.: figure

7.4. Background Audio

Wie im Hintergrund Audiofiles abgespielt werden können, auch nachdem die startende Applikation geschlossen wurde wird in einem How to detailliert erklärt. Deshalb erfolgt hier keine Vertiefung dieses Themas.

7.5. Background File Transfer

Grössere Filetransfers (Up- oder Downloads) sollten möglichst asynchron erfolgen. Um den Transfer auch fortzusetzen wenn die auslösende Applikation verlassen wurde, sind Background File Transfers geeignet. Um einen Filetransfer im Hintergrund durchzuführen muss ein Request-Objekt vom Typemoba Multitasking BackgroundTransferRequest erstellt und mit den nötigen Informationen abgefüllt werden. Es sind dies Quelle und Ziel des Transfers, die Methode (HTTP oder HTTPS) sowie weitere Details der Verbindung. Quelle und Ziel werden als URI und Verzeichnis/File im Isolated Storage angegeben, je nachdem ob es sich beim Transfer um einen Upload oder Download handelt. Ist der BackgroundTransferRequest fertig „ausgefüllt“ wird er dem BackgroundTransfer- Service übergeben. Dies ist im Prinzip eine Queue, die die Requests entgegennimmt. Beide Klassen sind im Namespace Microsoft.Phone.BackgroundTransfer zu finden.

7.5.1. Limitierungen

- Eine einzige Queue für alle Transfers, daher kann es vorkommen, dass ein Transfer nicht sofort beginnt
- Maximal 5 Transfers pro Applikation in der Queue
- Maximaler Upload von 5 MB
- Maximaler Download von 20 MB über GSM, 100 MB über WiFi auf Batterie
- Maximal 2 gleichzeitige Transfers pro Gerät
- Maximal 500 Transfers in der Queue (über alle Applikationen)

Interessant ist, dass ein Transfer abgebrochen und später nochmals versucht wird, falls minimale Übertragungsraten unterschritten werden (50 Kbps bei 3G bzw. 100 Kbps bei Wifi/USB). Für den Background File Transfer gibt es noch Bedingungen an die Applikation die beim Review für den Marketplace überprüft werden:

- File Transfers müssen über ein sichtbares UI Element vom Benutzer ausgelöst werden.
- Der Benutzer muss die Möglichkeit haben sich den aktuellen Zustand laufender Transfers anzeigen zu lassen.
- Es muss dem Benutzer eine Möglichkeit geboten werden über ein UI Element Transfers abbrechen zu können.

7.6. Fazit

Oberste prio sind Batterielaufzeit sowohl als Rechenleistung. Alles andere wird diesen Kriterien untergeordnet. Eine Applikation sollte immer auch ohne die Multitasking-Eigenschaften sinnvoll funktionieren können. Die Illusion von Multitasking ist für den Benutzer gerade so gut wie echtes Multitasking!

Teil II.

Code

8. Binding + Property Changed

Listing 8.1: Viewmodel + Umgang mit Property Changed

```
namespace Flashcards05.ViewModels
{
    public class DownloadViewModel : INotifyPropertyChanged
    {
       public event PropertyChangedEventHandler PropertyChanged;
        private void NotifyPropertyChanged(String propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (null != handler)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }

          public bool IsDownloadInProgress
        {
            get
            {
                return _isDownloadInProgress;
            }

            private set
            {
                if (value != _isDownloadInProgress)
                {
                    _isDownloadInProgress = value;
                    ProgressVisibility = _isDownloadInProgress ?
                       Visibility.Visible : Visibility.Collapsed;
                    NotifyPropertyChanged("IsDownloadInProgress");
                    NotifyPropertyChanged("ProgressVisibility");
                }
            }
        }
    }


    }
```

Listing 8.2: Binding

```
<ProgressBar x:Name="progress"
                        IsIndeterminate="{Binding IsDownloadInProgress}"
                        Visibility="{Binding ProgressVisibility}"
                        Grid.Row="0"
                        Style="{StaticResource
                           CustomIndeterminateProgressBar}"/>
```

8.1. Binding mit Converters (vom Internet - nicht abgedeckt von Denzler)

```
using System;
using System.Windows.Data;

namespace App.Converters
{
    public class BooleanToVisibilityConverter : IValueConverter
    {
        readonly Random _generator = new Random();

        public object Convert(object value, Type targetType, object
            parameter, System.Globalization.CultureInfo culture)
        {
            bool actualValue = (bool) value;
            if (actualValue) return Visibility.Visible;
            return Visibility.Collapsed;
        }

        public object ConvertBack(object value, Type targetType, object
            parameter, System.Globalization.CultureInfo culture)
        {
            return null;
        }
    }
}
```

Folgendes bei Namespace Deklarationen im View hinzufügen

```
xmlns:converters="clr-namespace:App.Converters"
```

Folgendes Nach der Namespace Deklaration im View

```
<phone:PhoneApplicationPage.Resources>
  <converters:BooleanToVisibilityConverter x:Name="MyConverter">
</phone:PhoneApplicationPage.Resources>

 <TextBlock x:Name="MyInvisibleTextBlock" Visibility="{Binding
    MyBooleanProperty, Converter={StaticResource MyConverter}"}
```

9. Layout

9.1. Grid Beispiel

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
             <RowDefinition Height="Auto"/>
         </Grid.RowDefinitions>
         <Image x:Name="MyImage" Grid.Row="0" Height="200"
            Source="{Binding Image}" />
         <TextBlock Grid.Row="1" x:Name="question" Text="{Binding
            Question}" Style="{StaticResource labelStyle}"
            VerticalAlignment="Top" Margin="6,6,0,0"
            HorizontalAlignment="Left" Width="444" TextWrapping="Wrap" />
         <TextBlock Grid.Row="2" HorizontalAlignment="Left"
            x:Name="answer" Text="{Binding Answer}"
            Style="{StaticResource labelStyle}" VerticalAlignment="Top"
            Width="444" Margin="6,106,0,0" TextWrapping="Wrap" />
    </Grid>
```

10. Isolated Storage

Listing 10.1: Saveable object Example

```
namespace Flashcards05.Models
{
//Contract for Saveable Class
    [DataContract]
    public class Card
    {

        public Card(string question, string answer, string imgpath)
        {
            Question = question;
            Answer = answer;
            ImagePath = imgpath;

        }
//Annotation for saveable property, must be public.
        [DataMember]
        public string Question { get;  set; }
        [DataMember]
        public string Answer { get;  set; }
        [DataMember]
        public string ImagePath { get; set; }

        public BitmapImage actualImage { get; set; }
    }
```

Listing 10.2: Loading and Saving

```
namespace Flashcards05.PersistentData
{
    public class Storage : IStorage
    {
        public T Load<T>(string filename, string directoryname) where T :
            class
        {
            T retVal = null;
            String _filepath = System.IO.Path.Combine(directoryname,
                filename);
            IsolatedStorageFile storage =
                IsolatedStorageFile.GetUserStoreForApplication();
            if (storage.FileExists(_filepath))
                using (var sourceStream =
                    storage.OpenFile(_filepath, FileMode.Open))
                {
                    DataContractSerializer _mySerializer = new
                        DataContractSerializer(typeof (T));
                    retVal = (T) _mySerializer.ReadObject(sourceStream);
                }
```

```
        return retVal;
    }

    public bool Save<T>(string filename, string directoryname, T
        serializableObject) where T : class
    {
        IsolatedStorageFile storage =
            IsolatedStorageFile.GetUserStoreForApplication();
        storage.CreateDirectory(directoryname);

        string _filepath = System.IO.Path.Combine(directoryname,
            filename);
        try
        {
            using (IsolatedStorageFileStream file =
                storage.CreateFile(_filepath))
            {
                DataContractSerializer _mySerializer = new
                    DataContractSerializer(typeof (T));
                _mySerializer.WriteObject(file, serializableObject);
            }
        }
        catch (Exception e)
        {
            MessageBox.Show("Save object " + e.Message);
            storage.DeleteFile(_filepath);
            return false;
        }
        finally
        {
        }
        return true;
    }

    public void SaveImage(StreamResourceInfo info, string basepath,
        Card card)
    {
        using (var ms = new MemoryStream())
        {
            StreamResourceInfo imgFile = App.GetResourceStream(info,
                new Uri(card.ImagePath, UriKind.Relative));
            var bm = new BitmapImage();

            bm.SetSource(imgFile.Stream);
            var wbm = new WriteableBitmap(bm);
            wbm.SaveJpeg(ms,wbm.PixelWidth,wbm.PixelHeight,0,100);
            Save(card.ImagePath, basepath + App.ImageFolder,
                ms.ToArray());
        }


    }

    public BitmapImage LoadImage(string filename, string directoryname)
    {
        BitmapImage retVal = null;
        var bytearray = Load<byte[]>(filename, directoryname);
```

```
                using (var ms = new MemoryStream(bytearray, 0,
                    bytearray.Length))
                {
                    retVal = new BitmapImage();
                     retVal.SetSource(ms);
71              }
                return retVal;

            }
        }
76  }
```

11. App Lifecycle

```
public App()
        {
            // Global handler for uncaught exceptions.
            UnhandledException += Application_UnhandledException;

            // Standard Silverlight initialization
            InitializeComponent();

            // Phone-specific initialization
            InitializePhoneApplication();

            // Show graphics profiling information while debugging.
            if (System.Diagnostics.Debugger.IsAttached)
            {
                // Display the current frame rate counters.
                Application.Current.Host.Settings.EnableFrameRateCounter =
                    true;

                // Show the areas of the app that are being redrawn in each
                    frame.
                //Application.Current.Host.Settings.EnableRedrawRegions =
                    true;

                // Enable non-production analysis visualization mode,
                // which shows areas of a page that are handed off GPU with
                    a colored overlay.
                //Application.Current.Host.Settings.EnableCacheVisualization
                    = true;

                // Disable the application idle detection by setting the
                    UserIdleDetectionMode property of the
                // application's PhoneApplicationService object to Disabled.
                // Caution:- Use this under debug mode only. Application
                    that disables user idle detection will continue to run
                // and consume battery power when the user is not using the
                    phone.
                PhoneApplicationService.Current.UserIdleDetectionMode =
                    IdleDetectionMode.Disabled;
            }

        }

        // Code to execute when the application is launching (eg, from
            Start)
        // This code will not execute when the application is reactivated
        private void Application_Launching(object sender,
            LaunchingEventArgs e)
        {
         //   IsolatedStorageFile.GetUserStoreForApplication().Remove();
        }
```

```
        // Code to execute when the application is activated (brought to
            foreground)
        // This code will not execute when the application is first launched
43      private void Application_Activated(object sender,
            ActivatedEventArgs e)
        {
            // Ensure that application state is restored appropriately
            if (!App.ViewModel.IsDataLoaded)
            {
48
                App.ViewModel.LoadData();
            }
        }

53      // Code to execute when the application is deactivated (sent to
            background)
        // This code will not execute when the application is closing
        private void Application_Deactivated(object sender,
            DeactivatedEventArgs e)
        {
            // Ensure that required application state is persisted here.
58      }

        // Code to execute when the application is closing (eg, user hit
            Back)
        // This code will not execute when the application is deactivated
        private void Application_Closing(object sender, ClosingEventArgs e)
63      {
        }

        // Code to execute if a navigation fails
        private void RootFrame_NavigationFailed(object sender,
            NavigationFailedEventArgs e)
68      {
            if (System.Diagnostics.Debugger.IsAttached)
            {
                // A navigation has failed; break into the debugger
                System.Diagnostics.Debugger.Break();
73          }
        }

        // Code to execute on Unhandled Exceptions
        private void Application_UnhandledException(object sender,
            ApplicationUnhandledExceptionEventArgs e)
78      {
            if (System.Diagnostics.Debugger.IsAttached)
            {
                // An unhandled exception has occurred; break into the
                    debugger
                System.Diagnostics.Debugger.Break();
83          }
        }

        #region Phone application initialization

88      // Avoid double-initialization
```

```
        private bool phoneApplicationInitialized = false;

        // Do not add any additional code to this method
        private void InitializePhoneApplication()
        {
            if (phoneApplicationInitialized)
                return;

            // Create the frame but don't set it as RootVisual yet; this
               allows the splash
            // screen to remain active until the application is ready to
               render.
            RootFrame = new PhoneApplicationFrame();
            RootFrame.Navigated += CompleteInitializePhoneApplication;

            // Handle navigation failures
            RootFrame.NavigationFailed += RootFrame_NavigationFailed;

            // Ensure we don't initialize again
            phoneApplicationInitialized = true;
        }

        // Do not add any additional code to this method
        private void CompleteInitializePhoneApplication(object sender,
           NavigationEventArgs e)
        {
            // Set the root visual to allow the application to render
            if (RootVisual != RootFrame)
                RootVisual = RootFrame;

            // Remove this handler since it is no longer needed
            RootFrame.Navigated -= CompleteInitializePhoneApplication;
        }

        #endregion
    }
}
```