Implementierung einer Android-Applikation zur komfortablen Fernsteuerung eines Zielrechners

Senad Licina, Pascal Rost, Christopher Schewe

28. September 2011

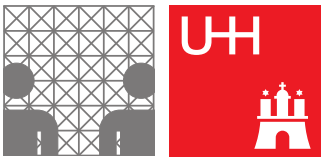
Projektbericht im Projekt

„Android Programmierung“

im Sommersemester 2011

Dozenten: Julian Kunkel

Projekt im Sommersemester 2011
Arbeitsbereich Wissenschaftliches Rechnen (WR)
Department Informatik
Universität Hamburg



Inhaltsverzeichnis

Kurzfassung

Das Ziel des Projekts ist die Erstellung einer Android-Applikation zur Fernsteuerung eines Zielrechners. Dabei sollen sowohl Linux- als auch Windows-Systeme unterstützt werden. Es ist vorgesehen, dass sowohl der Maus-Cursor als auch Tastatureingaben mit dem Programm gesteuert werden können. Ebenfalls soll es möglich sein, vordefinierte oder vom Benutzer erstellte Befehle (Unix, Windows) oder auch Makros an den Zielrechner zu senden, wo diese ausgeführt werden. Für die auf dem Desktop häufig angewandten Handlungsschritte, wie beispielsweise dem Anzeigen aller aktiven Anwendungen oder dem Wechseln zwischen offenen Fenstern, sollen im Programm leicht auszulösende Aktionen existieren. Die Benutzeroberfläche der Anwendung soll sich dem jeweiligen Arbeitskontext anpassen lassen. Beispielhafte Anwendungskontexte sind der Umgang mit einem Multimediaprogramm wie VLC und das „Surfen“ im Internet mittels eines Webbrowsers. Es sollen Profile für unterschiedliche Netzwerkkonfigurationen zur Verbindung des Mobiltelefons mit dem Zielsystem anlegbar sein. Programmkonfigurationen und Profile sollen exportierbar und zwischen unterschiedlichen Telefonen austauschbar sein. Wenn möglich, sollen alle Einstellungen des Programms sowie das Erstellen von Befehlen etc. auch mittels der Server-Applikation auf dem Zielsystem möglich sein (wenn eine Server-Applikation notwendig ist). Durch einen Synchronisationsprozess kommen die Daten auf das Telefon. Wenn möglich soll neben TCP/IP und SSH auch eine Verbindungsmöglichkeit mittels Bluetooth existieren.

1. Vorgehensweise

Für die Realisierung unseres Projektes haben alle Gruppenmitglieder die Entwicklungsumgebung Eclipse verwendet. Die verwendete Programmiersprache ist systemabhängig Java. Zusätzlich wurde notwendigerweise ebenfalls das Eclipse-Plugin für Android Development benutzt. Der Quellcode unseres Projektes wurde in einem extra für uns angelegten Redmine (git-repository) auf den Servern des „Arbeitsbereichs Wissenschaftliches Rechnen“ gehostet. Jedes Gruppenmitglied besaß ein Android-Handy, so dass jeder Teilnehmer zu jedem Zeitpunkt die Möglichkeit hatte, den Code auszuchecken, zu kompilieren und direkt auf seinem eigenen Gerät zu testen. Gearbeitet wurde teils gemeinschaftlich, teils jeder für sich. Zu Beginn wurde die Programmierarbeit und der Zuständigkeitsbereich zu jeder der drei grundlegenden Programm-Komponenten: Server-Applikation, Android-Applikation (GUI) und Android-Applikation (backend, Persistierung) auf jeweils einen Projektteilnehmer aufgeteilt, wobei prinzipiell nach dem Dogma der „Collective Code Ownership“ gearbeitet wurde, was bedeutet, dass trotz (fester) Zuständigkeiten jeder den Quell-Code des anderen zwecks Reviews und Anpassung sowie Bug-Fixing bearbeiten durfte und sich in diesem nach Möglichkeit auch auskennen musste. Als Name für unsere Applikation haben wir uns auf „Remoid“ geeinigt.

2. Design

Wie bereits im Punkt „Realisierung“ beschrieben, lässt sich die Software in drei Hauptteile gliedern. Diese drei Programmteile sind das Frontend, also die GUI des Programms, das Backend, bei dem es um die Persistierung der Einstellungen, Befehle, und Profile geht, sowie die Server-Applikation, welche auf dem Handy angestoßene Aktionen auf dem Zielrechner ausführt, welches allerdings auch eine in dem Android-Programm implementierte Komponente verwendet, damit eine Verbindung zwischen dem Handy und dem Zielsystem hergestellt werden kann.

3. Terminologie

Auf den folgenden Seiten werden bestimmte Begriffe verwendet werden, welche Teil einer festgelegten Terminologie innerhalb unseres Programms sind. Diese Terminologie wird innerhalb des Programms mittels eines Tooltip- und Hilfesystems den Benutzern nahe gebracht. Damit auch Nicht-Benutzer folgen können, werden hier alle wichtigen Begriffe einmal kurz erläutert. Der „Workspace“ ist die Hauptarbeitsfläche des Benutzers, mit welcher er Befehle und Aktionen an den Zielrechner senden kann und die somit den größten Teil der Interaktion unseres Programms darstellt. Ein Workspace kann mit einer beliebigen Anzahl an sogenannten „Screens“ gefüllt werden. Im Workspace wird zu jeder zeit genau ein Screen angezeigt und es kann zwischen den anderen bereits hinzugefügten Screens gewechselt werden. Ein Screen hat ein beim Hinzufügen festgelegtes „Layout“ (intern View) welches zum einen festlegt, wie viele „Buttons“ in welcher Anordnung auf dem Screen zur Verfügung stehen und zum anderen bestimmt, ob ein sogenanntes

„Touchpad“ auf dem Screen existiert. Sowohl Buttons als auch das Touchpad dienen dem Senden von Befehlen und Aktionen zum Server. Screens kann man als arbeitskontext-spezifische Arbeitsumgebungen betrachten. Die Buttons eines Screens können umbenannt und bestimmte sogenannte „Commands“ an sie gebunden werden. Commands sind vom Benutzer festlegbare Zeichenketten, welche auf dem Zielsystem durch den Server interpretiert und ausgeführt werden. Ein Command kann beispielsweise „firefox“ sein, um auf einem Linux-System den Webbrowser Firefox zu starten.

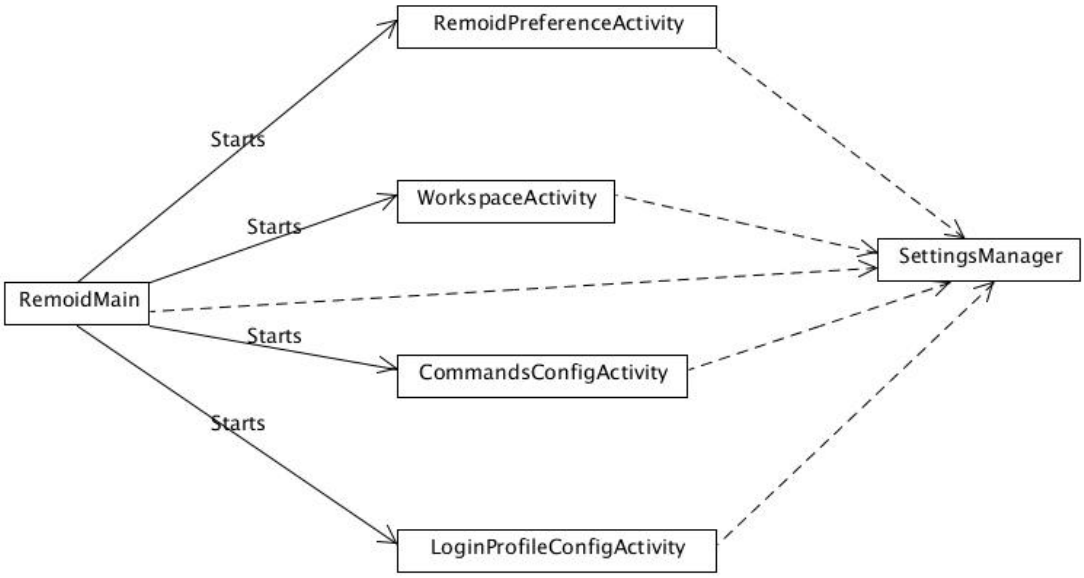


Abbildung 1: Stark vereinfachtes Klassendiagramm bzgl. der Activities

4. Implementation

4.1. Das Frontend - Die Android Application

Das Frontend ist der Teil des Android-Programms, welcher den internen Zustand des Systems grafisch wiedergibt und die Einstellungsmöglichkeiten, die möglichen Aktionen, das Hilfesystem und die Bedienung des Programms im Allgemeinen abbildet. Der Einstiegspunkt des Programms ist die Klasse „RemoidMain“. Ein instanziertes Objekt dieser Klasse visualisiert das Startmenü, mit verschiedenen „Buttons“, von dem aus man Login-Profile erstellen kann („LoginProfileConfigActivity“), Kommandos erstellen und verwalten („CommandsConfigActivity“), Programmeinstellungen tätigen („RemoidPreferenceActivity“) und das eigentliche Programm, den „Workspace“, im „Offline-Modus“ oder mit Verbindung zum Zielrechner starten kann („WorkspaceActivity“). Man kann anhand des Klassendiagrammes (Abb. 1 auf Seite 5) deutlich erkennen, dass alle Klassen Instanzen der Klasse „SettingsManager“ benutzen. Objekte von „SettingsManager“ bieten Operationen an, um zum einen den Zustand der aktuellen Einstellungen zu manipulieren, diese werden hauptsächlich in den Programmeinstellungen („RemoidPreferencesActivity“) verwendet, und zum anderen Operationen („applySettings()“), um die Einstellungen tatsächlich umzusetzen. Letztere werden immer zu Beginn der Instanzierung bzw. des Starts des jeweiligen Unterprogramms benutzt. Die Implementationsdetails der weniger komplexen Klassen bzw. Unterprogramme „RemoidPrefenceActivity“, „LoginProfileConfigActivity“ und „CommandsConfigActivity“ werden im Folgenden ausgelassen. Es ist jedoch anzumerken, dass in diesen Subprogrammen (ausgenommen der Aktivität für die Einstellungen, „RemoidPrefenceActivity“) ein Großteil der Interaktion mit dem „Backend“ der Android-Applikation ablaufen. So müssen natürlich alle eingefügten Profile und Kommandos persistiert werden. Mehr Informationen über die Persistierung gibt es in einem späteren Teil dieses Dokuments. Ebenfalls nicht im Klassendiagramm enthalten ist die Klasse „ConnectionManager“. Ein Objekt der Klasse „RemoidMain“ hält eine Instanz auf ein Objekt dieser Klasse, um sich zu dem Server zu verbinden. Im Folgenden wird also der Teil des Programms, in dem der interessante Teil der GUI und der Bedienung liegt, analysiert, der sogenannte „Workspace“, welcher durch die Klasse „WorkspaceActivity“ modelliert ist.

Objekte der Klasse „WorkspaceActivity“ nutzen den bereits genannten „ConnectionManager“, welches eine Klasse ist, die sowohl das Singleton-Entwurfsmuster[1] als auch das Zustands-Entwurfsmuster[2] (für Offline- und Online-Modi) implementiert, um Tastendrücke des Virtuellen-Keyboards an den Server zu schicken. Ebenfalls wird eine Instanz der Klasse „ViewFlipper“ verwendet, welches eine von der Android-API bereitgestellte Klasse ist[3], welche unter anderem dafür zuständig ist, den Effekt, der beim Wechseln der einzelnen Screens zu sehen ist, zu erzeugen. Dafür benötigt sie die Views der einzelnen Screens. „View“ ist ebenfalls eine von der Android-API bereitgestellte Klasse[4]; über

[1] Das Singleton-Pattern in der englischen Wikipedia
[2] Das Zustands-Pattern in der englischen Wikipedia
[3] Die Klasse ViewFlipper in der Android-API
[4] Die Klasse View in der Android-API

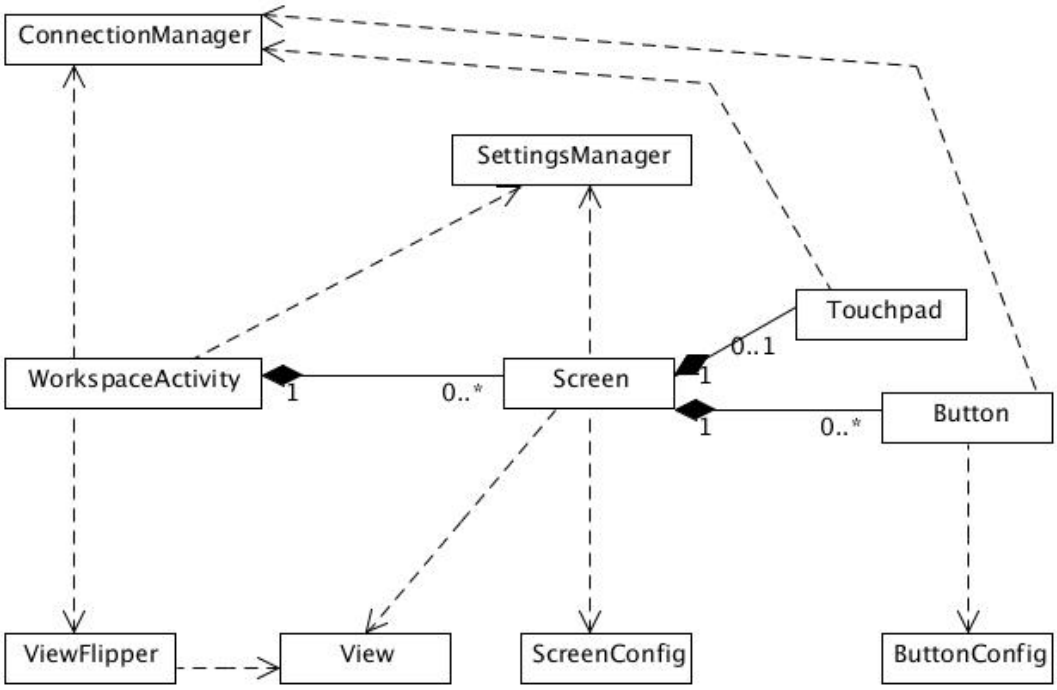


Abbildung 2: Stark vereinfachtes Klassendiagramm ausgehend von der WorkspaceActivity

Instanzen der Klasse „View“ kann man auf vorher per XML definierte GUI-Elemente zugreifen und mit diesen arbeiten. Jeder Screen hat eine View. Im derzeitigen Stand des Programms gibt es eine Handvoll an Views, im Programmkontext „Layout“ genannt, welche man jeweils einem Screen zuordnen kann. Ein Workspace kann keinen oder beliebig viele Screens haben. Jeder Screen hat eine Instanz der Klasse „ScreenConfig“ und entweder eine oder keine Instanz der Klasse „Touchpad“, sowie keinen oder beliebig viele Instanzen der Klasse „Button“. Sowohl Objekte der Klasse „Button“ als auch „Touchpad“ verwenden ebenfalls die Instanz des ConnectionManagers, um Daten an den Server zu übertragen. Im Falle des „Touchpad“ werden Koordinaten der Finger des Benutzers und eventuelle Gesten übertragen (Anmerkung: in der aktuellen Version gibt es noch keine Gesten, die auf einem Touchpad ausgeführt werden können). Im Falle „Button“ werden die an den jeweiligen Button gebundenen Kommandos an den Server übertragen.

4.2. Backend - Persistierung der Daten

Die Daten, die gespeichert werden müssen, sind die Login-Profile des Benutzers, seine selbst erstellten Befehle, der Zustand des Workspaces mit den Anordnungen der Screens, die Zustände der Buttons auf den Screens, also welche Befehle an sie gebunden sind und wie sie vom Nutzer benannt wurden, sowie der jeweils aktuelle Zustand der Programmeinstellungen. Um in einem Android-Programm Daten zu persistieren, gibt es mehrere Möglichkeiten. Zur Speicherung des Zustands eines jeden Java-Programms bietet sich prinzipiell die sogenannte „Serialisierung“ an, ein Vorgang, bei dem der aktuelle Zustand eines Programms, also die Zustände aller Objekte und ihrer Beziehungen untereinander, in einer Datei gespeichert wird und zu einem späteren Zeitpunkt wieder ausgelesen werden kann. Für diesen Vorgang bietet die Standard Java API einfache Schnittstellen an. Die Android-Entwicklerplattform bietet ebenfalls eine derartige Schnittstelle für die Programmierer an, jedoch ist diese zum Zeitpunkt der Backend-Implementierung nachweislich sehr langsam gewesen. Andere Möglichkeiten wären unter anderem die Verwendung einer Open Source-Bibliothek, mit welcher der Zustand unseres Programms mittels der JSON-Syntax in Dateien festgehalten werden würde, sowie Bibliotheken, welche den Zustand auf XML abbilden und ebenfalls in einer Datei speichern würden. Die beste und auch der Struktur unserer internen Daten am nächsten kommende Lösung, ist die Verwendung der von der Android-API bereitgestellten Schnittstelle zur Nutzung der ebenfalls auf jedem Android-System zur Verfügung stehenden SQLite-Datenbank. In unserem Programm speichern wir also die Daten in einer SQLite-Datenbank, wobei die Persistierung der Programm-Einstellungen mittels einer speziellen Schnittstelle direkt vom Android-System selbst übernommen wird[5].

Damit die Front- und Backend-Entwickler unabhängig voneinander arbeiten konnten, haben wir uns zuerst einmal auf ein Interface geeinigt, welches zur Speicherung und Wiederherstellung der Daten genutzt wird. Es bietet alle nötigen Operationen an, um Screens, Buttons, Profile und Commands speichern, laden und entfernen zu können. So konnte der Frontend-Entwickler, bis die Datenbank-Funktionalität implementiert war,

[5] Die SQLite-Schnittstelle von Android

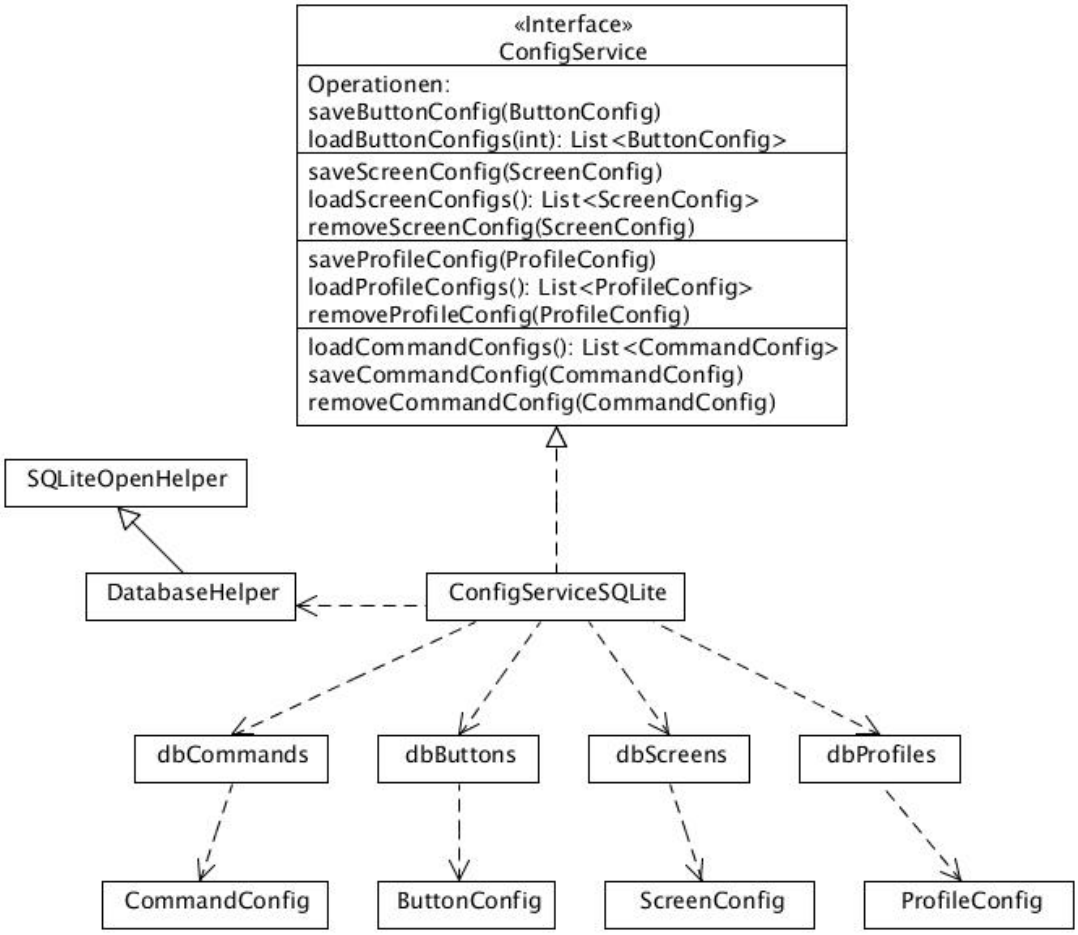


Abbildung 3: Vereinfachtes Klassendiagramm, welches die Implementationsdetails auf Klassen-Ebene darstellt

mit einer selbst erstellen Dummy-Klasse arbeiten, welche dieses Interface implementierte. Die tatsächlich genutzte Klasse für alle backend-spezifischen Operationen ist nun die „ConfigServiceSQLite“, die prinzipiell alle Aufgaben an die Klassen „dbCommands“, „dbButtons“ etc. weiterleitet, welche direkt mit den von Android angebotenen Schnittstellen sehr nah an der Datenbank arbeiten. Um mit der Datenbank arbeiten zu können, war es ebenfalls nötig, eine Klasse zu erstellen („DatabaseHelper“), die von der Android-Klasse „SQLiteOpenHelper“ erbt und bestimmte Operationen implementiert, die den Datenbankzugriff steuern. Unter anderem wird in dieser Klasse festgelegt, auf welchen Tabellen gearbeitet wird und wie die einzelnen Tabellen und Felder der Datenbank aufgebaut sind. Zu erwähnen ist noch, dass in den „CommandConfig“, „ButtonConfig“ etc. Klassen durch schlaue Implementierung und Nutzung privater innerer Klassen innerhalb der „db“-Klassen keinerlei datenbank-spezifische Details auftauchen (wie etwa der künstlich erzeugten Primärschlüssel). Dadurch ist das Backend komplett vom Frontend entkoppelt und es kann bei Bedarf ohne große Probleme von einem Datenbank-Backend zu beispielsweise einem XML- oder Serialisierungs-Backend gewechselt werden, ohne dass die Implementation des Frontends, also sogar der „Config“-Klassen, angepasst werden muss.

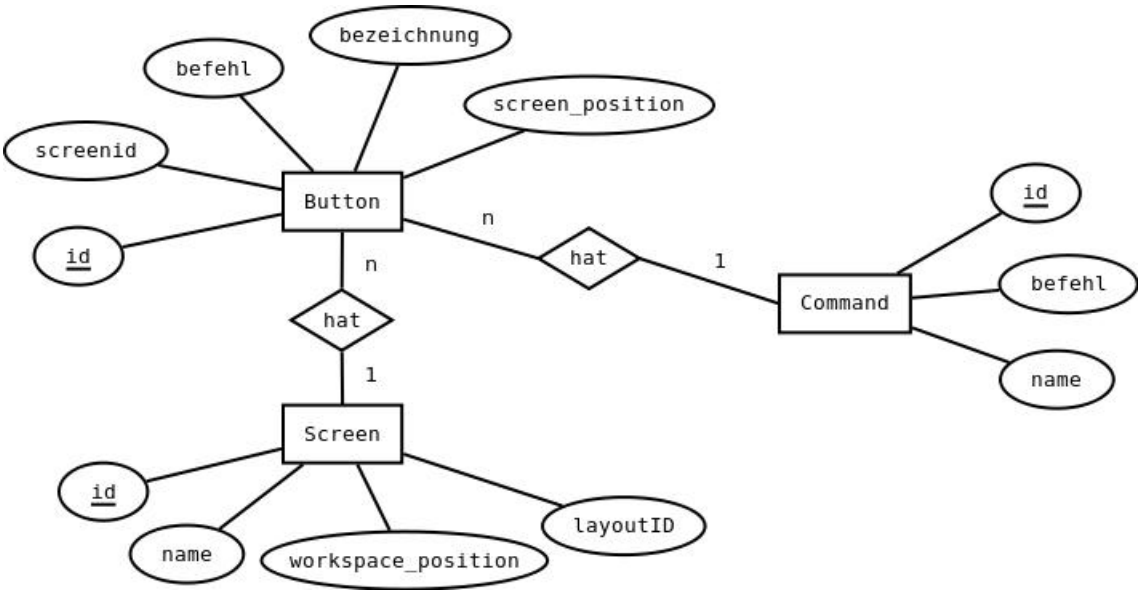


Abbildung 4: Ein ER-Diagramm welches den Ist-Zustand der Strukturierung unserer Daten in der Datenbank zeigt

Wie man in der Abbildung 4 sieht, werden derzeit keine Fremdschlüssel verwendet, obwohl dies sinnvoll und sehr einfach zu realisieren wäre (einen entsprechenden Eintrag findet man auch in den Tickets des Redmines). Jede Entität enspricht in der Datenbank einer Tabelle, mit Feldern, welche den Attributen der jeweiligen Entität entsprechen. Die „id“ ist in der Datenbank ein automatisch erzeugter Primär-Schlüssel. Damit ohne Fremdschlüssel die Zusammenhänge der einzelnen Daten nachvollzogen werden können, werden diese redundant, beispielsweise als String, in den entsprechenden Feldern gespei-

chert. (Der Befehl eines Buttons ist der Befehl eines Commands und nicht die ID des selbigen.)

5. Serverapplikation und Netzwerk

Dem Netzwerk von Remoid und der Applikation des Zielrechners liegt eine Server-Client-Architektur zugrunde, wobei der Zielrechner als Server und die Android-Geräte als Clients dienen. Dadurch besteht die auch die Möglichkeit, sich mit mehreren Clients gleichzeitig zu einem Server zu verbinden. Wir haben bewusst dies ermöglicht, um Remoid flexibler und offener für verschiedene Einsatzkontexte zu gestalten.

5.1. Netzwerkkomponente auf dem Android Device

Die zentrale Komponente des Netzwerk-Codes auf dem Android-Gerät sind die Klassen *ConnectionManager, Connection* und *ConnectionConfiguration*. Das ConnectionConfiguration Interface bietet eine einheitliche Schnittstelle zum Verwalten der Konfigurationsdetails einer Netzwerkverbindung. In der aktuellen Version von Remoid stellt UnsecureConnectionConfiguration die einzige Implementation des Interfaces dar. Dieses besitzt lediglich die Informationen für Ziel-IP und Port, sowie einen benutzerdefinierten Namen für die Verbindung und ist, wie der Name schon vermuten lässt, für unsichere Netzwerkverbindungen. Die Klasse Connection stellt die eigentliche Verbindung von der Android App zur Serverapplikation dar. Sie bietet Operationen rund um die Verbindungsmöglichkeiten im Netzwerk, sowie Operationen zum Ausführen der diversen Aktionen an. Die Klasse ConnectionManager stellt wie die Connection-Klasse Operationen rund um die Netzwerkverbindung und Ausführung diverser Aktionen an. Genau gesagt, deligiert sie diese Operationen sogar an ein Exemplar der Connection-Klasse weiter. Die Klasse ist nach dem Entwurfsmuster des Singletons entworfen, um so sicher zu stellen, dass die Android Applikation jederzeit die gleiche und damit richtige Netzwerkverbindung nutzt. Zusätlich bietet die Klasse auch noch zwei Operationen an, um den Offline-Modus ein-, bzw auszuschalten.

Desweiteren gibt es die Klassen *ConnectedState*, sowie *OfflineState*. Beide Klassen implementieren das Interface *ConnectionState* und sind hauptsächlich für das Wechseln zwischen Offline-Modus und Online-Modus verantwortlich. Einen Offline-Modus haben wir deshalb implementiert, damit man auch dann, wenn man keine Verbindung zu einem Server hat, seine Konfigurationen (Netzwerkverbindungen, Commands und Workspace) einsehen und bearbeiten kann.

Schlussendlich gibt es noch die Klassen *MouseClick* und *MouseMovement*, sowie das Interface *Command* und die beiden Implementationen *DefaultCommand* und *MouseCommand*. Die beiden ersten Klassen dienen zum Ausführen der Mausbewegung sowie von Mausklicks. Während das MouseCommand lediglich eine Wrapper-Klasse für Exemplare von MouseClick ist, stellt die Klasse DefaultCommand den Zentralen das Werkzeug dar, um (Kommandozeilen-)Befehle auf dem Server auszuführen.

5.2. Netzwerkkomponente Server

In der Server Applikation befinden sich im net Package die Klassen *ConnectionManager, Connection, ServerConnectionConfiguration, UnsecureServerConnectionConfigura-*

tion, sowie *NetWorkMessagePrefix.* Auf diese Klassen wird hier nicht genauer eingegangen, da sie die gleiche oder aber ähnliche (im Falle von (Unsecure) ServerConnectionConfiguration) Funktionsweise haben, wie ihre Pendants in der Android-Applikation. Die wirklich interessante Klasse im Server ist die *InputProcessor.* Diese verarbeitet den einkommenden String und führt je nach Inhalt des Strings dann die entsprechenden Aktionen aus.

5.3. Netzwerkübertragungen

Damit nun eine auf dem Android Gerät gewählte Aktion auf dem Server ausgeführt werden kann, muss durch die beiden Netzwerkkomponenten eine Verbindung aufgebaut werden. Diese Netzwerkverbindung ist eine simple Java-Socket-Verbindung, die über TCP/IP läuft. Diese Verbindung ist in den jeweiligen Connection-Klassen gekapselt. Als 'übertragungsprotokoll' haben wir uns für einfache Strings entschieden, um so auch die Möglichkeit zu lassen, den Server in anderen Programmiersprachen zu entwickeln. Als alternative würde sich natürlich das Java hauseigende Remote-Procedure-Call Framework RMI (Remote-Method-Invocation) anbieten. In der aktuellen Version von Remoid ist die Netzwerkverbindung völlig ungesichert, weswegen es nur im Heimnetzwerk, welches selbst gesichert sein sollte, genutzt werden sollte.

6. Ist-Zustand

Abbildung 5: Remoid nach Programmstart, man befindet sich im Haupt-Auswahlmenu

Zum Ende des Projektes befindet sich unser Programm in einem Zustand, in dem die meisten der in den Anforderungen genannten Features zwar verwendet werden können, einige jedoch unvollständig, unzureichend oder instabil funktionieren. Prinzipiell können Commands und Login-Profile erstellt, manipuliert und entfernt werden, Einstellungen wie Vollbild, Offline-Modus und Sperrung des Workspaces getätigt werden, sowie mittels „Login“ bei korrekt eingestelltem Server und Login-Profil zum Zielrechner verbunden werden. Es ist ebenfalls möglich, sämtliche Anpassungen, die man im Programm vorgenommen hat, mittels „export“ auf die SD-Karte des Telefons zu exportieren und sie analog mittels „import“ wieder zu importieren.

Im Workspace funktioniert das Hinzufügen und navigieren zwischen den Screens, es existieren drei verschiedene Layouts, die der Benutzer für einen Screen verwenden kann. Buttons lassen sich umbenennen und vorher erstellte Commands lassen sich verschiedenen Buttons zuordnen. Die Steuerung des Maus-Cursors auf dem Zielrechner funktioniert, Commands können auf den Zielsystemen ausgeführt werden. Zusätzlich existieren einige Standard-Commands, welche nicht löschbar sind und jederzeit an die Buttons gebunden werden können. Dieses sind Commands zum Starten der virtuellen Tastatur und zum Senden von linken und rechten Mausklicks an den Zielrechner.

Die Übertragung von Tastendrücken von der virtuellen Tastatur des Android-Telefons zum Zielsystem funktioniert bisher nur teilweise, bestimmte Zeichen und Großschreibung funktionieren noch nicht. Es exisitert ein Tooltip und hilfsmenü-basiertes Hilfesystem, welches dem Benutzer einen leichten Einstieg in das Arbeiten mit „Remoid“ bieten soll.

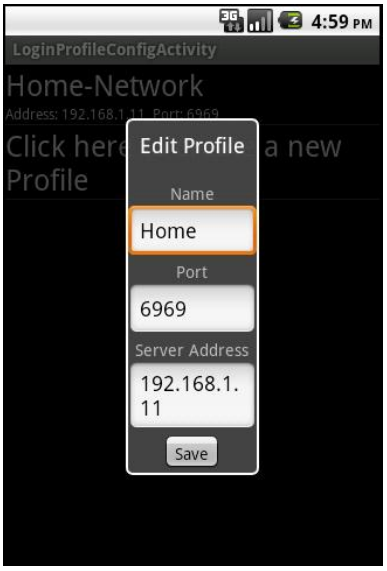


Abbildung 6: Editieren eines vorher erstellten Login-Profils

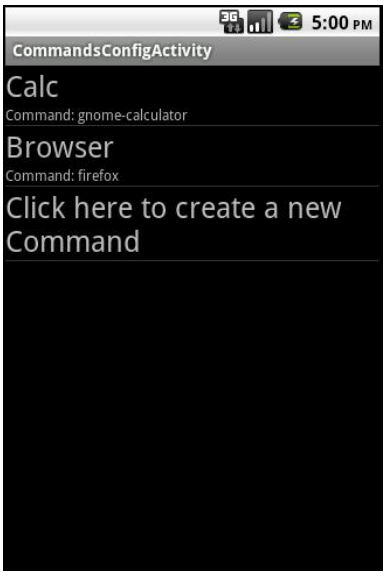


Abbildung 7: Übersicht über erstellte Commands

Abbildung 8: Ein Workspace in dem noch keine Screens angelegt wurden

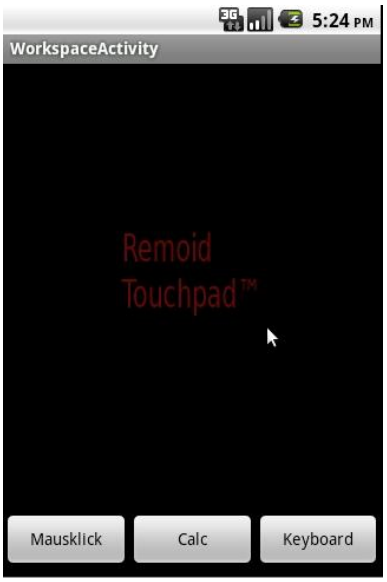


Abbildung 9: Ein Screen mit Touchpad und drei modifizierten Buttons

7. Verwendung/Installation

Um Remoid auf dem Handy zu installieren, benötigt man zwei Komponenten, zum einen die Server-Applikation, welche auf dem Zielrechner gestartet werden muss und zum anderen Remoid selbst. Die Server-Applikation benötigt eine installierte Java-Runtime-Environment auf dem Zielrechner und kann als lauffähige „jar"-Datei auf unserer Redmine-Projekt-Seite unter der Rubrik „Files" heruntergeladen werden. Es wird empfohlen, den Server in einer Konsole zu starten (Linux), um gegebenenfalls Programm- und Debug-Informationen bzw. Feedback sehen zu können. Die App, Remoid, ist derzeit noch nicht im Android-Market erhätlich und muss ebenfalls vom Redmine heruntergeladen werden. Danach kann das Programm wie jedes andere auf dem Handy installiert werden. Wichtig hinzuzufügen ist, dass man derzeit keine Serverseitigen Konfigurationen tätigen kann und der default Port des Servers „6969" ist. Um Remoid im Android-Emulator starten zu können muss ein Android-SDK[6] installiert sein, Eclipse mit dem ADT-Plugin[7] vorliegen und der Quellcode aus unserem Repository ausgecheckt sein. Es wurden keine weiteren Bibliotheken verwendet.

8. Soll-Zustand

Um den gewünschten Soll-Zustand zu erreichen fehlen noch Maus-Gesten, sowie die Unterstützung von betriebsystemeigenen Shortcuts, allgemeinen Medienabspiel-Funktionen und eine vollwertige Implementierung der Keyboard-Funktion. Desweiteren könnte das Design der Datenbank eine Überarbeitung gebrauchen. Auch die Netzwerkkomponente könnte eine Überarbeitung gebrauchen, z.B. in Form einer Implementierung des Command-Patterns. Zusätzlich wäre die Möglichkeit einer sicheren Verbindung wünschwert, genauso wie Verbindungen über SSH (die dann keine Mausunterstützung haben würde) oder Bluetooth (das Android Gerät direkt als Human-Interface-Device am Zielsystem anmelden). Die Server-Applikation sollte Einstellungs-Möglichkeiten bieten und dazu eine GUI bereitstellen.

In der verbesserten Version des ER-Diagramms (siehe Abbildung 10 auf Seite 18), kann man erkennen, dass keine künstlich erzeugten IDs mehr verwendet werden. Stattdessen werden echte Primär- und Sekundärschlüssel und verwendet.
Ein wichtiges Feature, welches derzeit noch fehlt, ist die Automatische Server-Suche und ein automatischer Verbindungsaufbau. Einige Apps von anderen Entwicklern im Android Market können dies. So kann in sicheren Netzwerken, wo keine Gefahr für einen Missbrauch der Funktion besteht ohne ein Login-Profil gearbeitet werden, welches vor allem das erstmalige Starten und Bedienen unserer Anwendung erleichtert.

[6] Link zur Download-Seite des Android-SDK für verschiedene Systeme
[7] Link zur Eclipse-ADT Seite

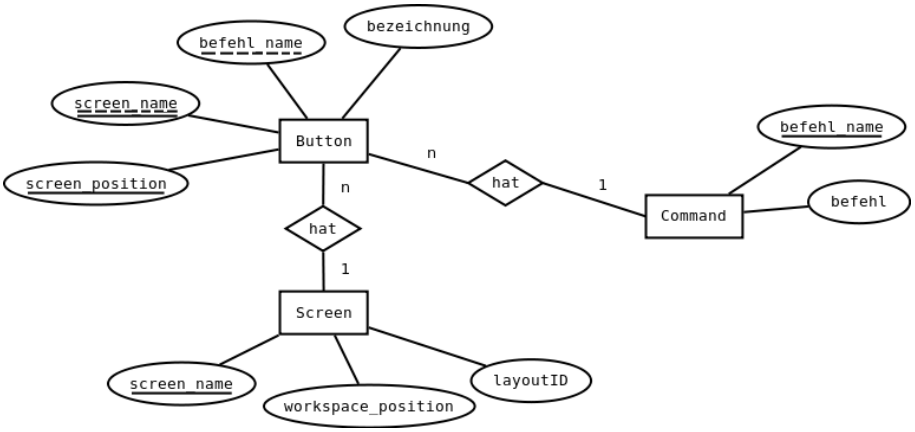


Abbildung 10: Ein ER-Diagramm welches den Ist-Zustand der Strukturierung unserer Daten in der Datenbank zeigt

9. Alternativen

Auf dem Android-Market gibt es eine einige Applikationen die das gleiche wie unsere App versuchen, einen Zielrechner möglichst bequem und über viele Anwendungen hinweg fernsteuern zu können. Ausserdem gibt es sehr viele Apps, welche für die Fernsteuerung ganz bestimmter Anwendungen auf dem PC entwickelt wurden. Als eines der belibtesten Programme zur Steuerung vieler Anwendungen sei hier einmal die „Unified Remote“[8] genannt. Der Vorteil dieser App ist, dass sie eine starke Verschlüsselung, Bluetooth-Verbindung und Multitouch-Funktionen für die Maussteuerung anbietet. Ein Nachteil ist, dass der Server für diese Anwendung nur unter Windows läuft und die Vollversion der App Geld kostet. Anstatt auf große Anpassungsfähigkeit und selbsteinstellungs möglichkeiten zu setzen, wie wir es mit unserem Programm vorhaben, bietet diese App für jede Aufgabe eine festgelegte Fernbedienung (vergleichbar mit den Screens aus unserer App) an, beispielsweise für den Umgang mit dem Dateisystem oder dem Chrome-Webbrowser. Unser Programm punktet hier mit starker individualiserbarkeit und der Möglichkeit shell-Kommandos direkt auf den Server zu schicken, was besonders interessant für Linux-Profis ist. Beispiele für spezialisierte Apps sind zum einen „VLC Remote“[9], mit welcher das Programm VLC ohne zusätzliche Serversoftware vom Smartphone aus bedient werden kann und „YouTube Remote“ [10] von Google, mit welcher auf einem PC laufende YouTube-Videos ausgewählt und gesteuert werden können.

[8] http://www.unifiedremote.com/
[9] http://hobbyistsoftware.com/vlc-more.php
[10] „YouTube Remote“ auf AppBrain