UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

ESPECIALIZAÇÃO EM TECNOLOGIA JAVA

CLAYTON EDUARDO KUHN

ELABORAÇÃO DE UM PROTÓTIPO DE APLICATIVO PARA ACOMPANHAMENTO DE REQUISIÇÕES DE TÁXI

MONOGRAFIA DE ESPECIALIZAÇÃO

CURITIBA

2012

CLAYTON EDUARDO KUHN

ELABORAÇÃO DE UM PROTÓTIPO DE APLICATIVO PARA ACOMPANHAMENTO DE REQUISIÇÕES DE TÁXI

Monografia de especialização apresentada ao Curso de Especialização em Tecnologia Java da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do título de especialista.

Orientador: Professor Dr. João Alberto Fabro

CURITIBA

2012

AGRADECIMENTOS

Aos professores do Curso de Especialização Java que transmitiram o conhecimento com muita qualidade. Ao professor João Alberto Fabro pela ajuda e conselhos na confecção do documento. A todos que contribuíram de forma direta e indireta no desenvolvimento deste trabalho.

RESUMO

A mobilidade computacional vêm crescendo em um ritmo bastante acelerado. Isso têm gerado um interesse bastante grande em serviços móveis por parte das empresas. Utilizando redes de comunicação móvel como 3G e WiFi, as empresas podem oferecer aos usuários serviços que são acessados de dispositivos móveis como smartphones. Esses serviços móveis oferecem uma facilidade para o usuário acessá-los e se tornam um atrativo aos olhos de quem utiliza. Uma área que pode oferecer muitos serviços desse tipo é a área de transporte urbano. As empresas de transporte de ônibus, metrô e táxi podem usufruir da mobilidade computacional para oferecer serviços aos usuário e reduzir seus custos, além de ser uma forma de engajamento do usuário junto ao serviço devido a facilidade do mesmo em acessar e acompanhar os serviços de transporte. As centrais de Táxi podem ser automatizadas para receberem solicitações de atendimento através de dispositivos móveis e fazer o despacho automatizado dos táxis para atendimento dos clientes. Isso traz às empresas de Táxi uma redução bastante grande nos custos operacionais e pode reduzir o tempo de atendimento aos usuários. Para os usuários do serviço de táxi traz uma facilidade muito grande em utilizar o serviços pois utiliza o conceito de autosserviços, ou seja, o usuário utiliza o serviço sem intermédio de uma central telefônica. O que antes era uma possibilidade remota, hoje com o advento das redes móveis e da mobilidade computacional é possível. A alta conectividade permite a integração de várias tecnologias, formando um grande provedor de serviços.

Palavras-chave: Dispositivos móveis. Computação móvel. Autosserviços. Redes de comunicação móvel.

ABSTRACT

Mobility computing is growing at a rapid pace. This has generated a rather large interest in mobile services by businesses. Using mobile communication networks such as 3G and WiFi, enterprises can offer users services that are accessed from mobile devices like smartphones. These services provide a mobile ease for the user to access them and become an attraction to users. One area that can offer many mobile services is the area of urban transport. Transport companies of bus, subway and taxi can take advantage of mobile computing to provide services to users and reduce their costs as well as being a form of user engagement with the service due to ease of access and even monitor the transport services. The táxi companies can be automated to receive service requests via mobile devices and make the automated taxi dispatch to serve the customers. This brings to a taxi company rather large reduction in operating costs and can reduce the time of service to users. For taxi users brings a great ease to use the service because it uses the concept of self-service, ie, the user use the service without the intermediary of a central telephone. What was once a remote possibility today with the advent of mobile computing and networking is possible. The high connectivity allows integration of various technologies, forming a large service provider.

Keywords: Mobile devices. Mobile computing. Self-service. Mobile communication networks.

LISTA DE SIGLAS

AJAX	Asyncronous Javascript and XML
API	Application Program Interface
EJB	Enterprise Javabeans
HTTP	Hypertext Transfer Protocol
JAX-WS	Java API for XML WebServices
JPA	Java Persistence API
JEE	Java Enterprise Edition
JME	Java Mobile Edition
JSE	Java Standard Edition
JSON	Javascript Object Notation
JVM	Java Virtual Machine
ORM	Object Relational Mapping
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SDK	Software Development Kit
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structuted Query Language
UML	Unified Modeling Language
XML	eXtensible Markup Language
WSDL	WebService Definition Language

Sumário

1. Introdução

Atualmente o sistema de táxi brasileiro possui alguns problemas, o que pode prejudicar a experiência do usuário. As centrais telefônicas podem estar sobrecarregadas devido à quantidade de requisições. O despacho do táxi pode não estar sendo otimizado o quanto poderia. Os problemas citados podem causar grandes transtornos aos usuários desse serviço, que precisam de um sistema confiável que os leve aos lugares desejados dentro do tempo esperado. Com isso, quanto mais automatizado e otimizado for o serviço, a experiência e confiabilidade do usuário no sistema de táxi irá aumentar e consequentemente os usuários irão procurar mais por esse serviço.

1.1. Solução

Uma possível solução para esses problemas seria a automatização e otimização do sistema de táxi. Exemplos de automatização seriam: reduzir a carga das centrais telefônicas, diminuindo o tempo necessário para solicitar um táxi; diminuir o tempo de despacho do táxi, reduzindo o tempo de espera pelo serviço do usuário. Esse trabalho apresenta uma proposta de solução destes problemas através de um sistema computacional que permite aos usuários e taxistas se comunicarem através de aplicativos desenvolvidos para seus smartphones.

1.2. Objetivos

Os objetivos desse trabalho estão voltados para o desenvolvimento de uma plataforma que possibilite a automatização e otimização do uso do serviço de táxi. Para atingir os objetivos do trabalho, a plataforma se concentra em prover os seguintes serviços:

- Requisição do serviço de táxi através de um aplicativo de smartphone, sem uso da central telefônica.
- Despacho automatizado de táxi. Sistema otimizado para buscar um táxi que esteja livre e mais próximo do usuário sem intervenção de um atendente da central telefônica.
- Acompanhar o andamento da requisição através do smartphone. Todas as alterações na solicitação serão visualizadas pelo usuário.

Para o trabalho em questão, o desenvolvimento do aplicativo de smartphone utilizará como base a plataforma Android. O aplicativo oferecerá aos usuários as seguintes funcionalidades:

- Fazer uma solicitação para o serviço de táxi utilizando somente 1 clique(*One Click to Service*).
- Acompanhar as alterações do andamento solicitação:
    - *Status* da solicitação
    - Observações da central de atendimento
    - Veículo e placa que irão atender o usuário
    - Distância e tempo estimado de chegada
- Cancelar uma solicitação iniciada.

O servidor é responsável por receber as requisições do aplicativo Android e realizar os serviços:

- Criar solicitação de atendimento do usuário.
- Receber e atualizar coordenadas de táxis.
- Enviar atualização da solicitação para o usuário.
- Cancelar/Encerrar solicitações.
- Despachar solicitação para um táxi.

1.3. Justificativa

A popularização dos smartphones têm proporcionado novos nichos de mercado a serem explorados e ainda ampliado a possibilidade de atendimento de serviços diversos que auxiliem o público geral na execução das atividades diárias. O serviço de táxi em várias cidades está defasado e com problemas para atendimento do público.

Somente aumentar a frota ajuda a amenizar o problema porém não é a única solução. Otimizar o uso do táxi utilizando funcionalidades como 3G e GPS pode ajudar a melhorarmos o serviço de táxi e consequentemente aumentarmos a confiabilidade do usuário no serviço. Além disso, se torna um atrativo para o usuário pela facilidade em solicitar e acompanhar o seu atendimento.

1.4. Estrutura do Trabalho

O trabalho em questão é composto de estudos e aplicação de várias tecnologias de mercado, tendo como resultado um sistema destinado aos usuários e operadoras do serviço de táxi.

O capítulo 1 foi uma abordagem geral do trabalho, seus objetivos, problemas envolvidos e descrição do cenário atual.

O capítulo 2 cita algumas referências de trabalhos com temas semelhantes. A seguir, são apresentadas as tecnologias utilizadas para o desenvolvimento do trabalho. Entre elas, podemos citar: Android SDK, JAX-WS, EJB 3.1 e JPA.

O capítulo 3 mostra o detalhamento do trabalho. Neste capítulo é apresentado como cada componente do sistema é integrado através da modelagem UML.

O capítulo 4 apresenta as conclusões e considerações finais sobre o trabalho, além de possíveis trabalhos futuros, e a seguir são apresentadas as referências bibliográficas.

2. Revisão Bibliográfica

2.1 Trabalhos correlatos

2.1.1 Problema de Atribuição de Tarefas

Em trabalho correlato, Pierobom (2012) fez um estudo sobre a utilização de algoritmos de Otimização por Nuvem de Partículas - Particle Swarm Optimization (PSO) para o Problema de Atribuição de Tarefas (*Task Assignment Problem* - TAP) .

“(...) A tarefa de otimização consiste em buscar soluções para um problema que conduzam a valores extremos de sua função objetivo em um determinado intervalo .”(Pierobom, 2011)

Dentre as aplicações práticas na área de TAP, podemos citar: programação de disciplinas em cursos universitários, alocação de espaço de armazenamento, carregamento de caminhões, entre outros. O trabalho em questão faz a utilização do algoritmo PSO para otimização de Alocação *Taxi-Cliente(PATC)*, que pode ser considerado um Problema de Atribuição de Tarefas.

O trabalho desenvolvido por Pierobom tem relação com o trabalho aqui apresentado pelo fato de serem sobre o Problema de Alocação Taxi-Cliente(PATC). Porém, as 2 abordagens possuem focos diferentes. O trabalho analisado tem um foco na otimização global do problema e na busca da solução ótima utilizando o algoritmo PSO. O trabalho descrito nesse documento tem foco mais na automatização da alocação e solicitação de táxi. Claro que a otimização também é buscada, porém de forma local e não global.

2.1.2 Sistema de chamados de Táxi para Smartphones

Em trabalho semelhante, Muraro (2011) desenvolveu um protótipo para solicitar atendimento de táxi através de um smartphone. O protótipo desenvolvido consiste em 3 módulos:

- Aplicação para smartphones: aplicativo desenvolvido em Android e possibilita ao usuário solicitar um Táxi.
- Taxímetro: controlador responsável por calcular o preço total da solicitação além de possuir um módulo feito em Python para comunicação com o servidor. O taxímetro possui GPS e acesso a rede GSM.
- Sistema gerenciador: módulo servidor responsável por tratar as requisições do smartphone e taxímetro. Possui um WebService exposto para acesso além de interface com banco de dados.

A arquitetura do protótipo pode ser vista na imagem abaixo.

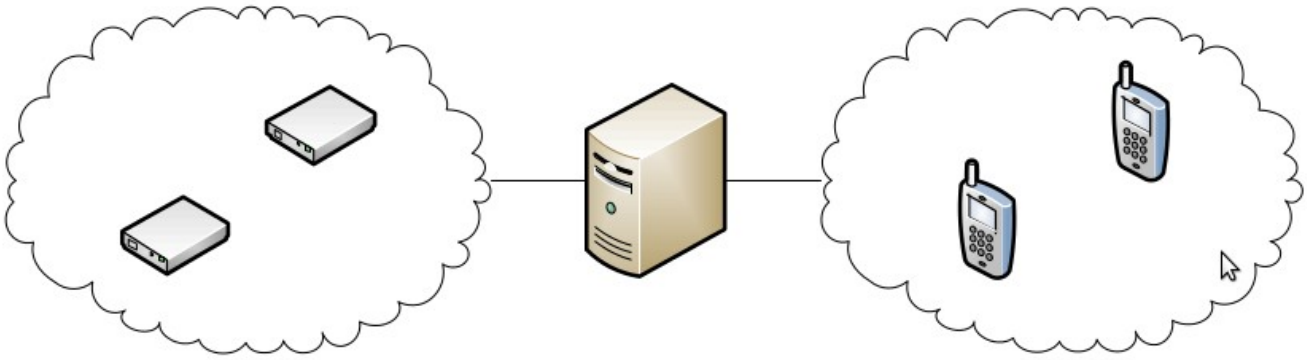

Figura 1: Arquitetura do protótipo
Fonte: Muraro, 2011.

A ideia do projeto é automatizar a solicitação de táxi pelo usuário. O funcionamento básico do protótipo segue os passos abaixo:

1. Passageiro abre chamado pelo smartphone.
2. Servidor busca veículo disponível mais próximo.
3. Servidor envia associa o motorista ao chamado.
4. Motorista aceita o chamado.
5. Motorista se aproxima do passageiro.

6. Servidor envia sinal ao passageiro.
7. Motorista chega até o passageiro.
8. Servidor libera o motorista.

O trabalho desenvolvido por Muraro têm relação com trabalho aqui apresentado. Ambos são protótipos para automatizar a solicitação de Táxi. A diferença está nas funcionalidades apresentadas e o fato do protótipo aqui proposto utiliza um aplicativo de smartphone também para o taxista. Isso aumenta bastante o potencial do protótipo, já que um sistema como o Android oferece muitas vantagens como navegação, melhor interface visual, integração com redes sociais etc.

2.2 Tecnologias

Desde o seu lançamento, a tecnologia Java está cada vez mais presente no mercado e na área acadêmica. A plataforma Java possui uma comunidade bastante ativa, que ajuda no desenvolvimento da linguagem bem como das especificações que cercam a plataforma Java. A plataforma java possui uma clara separação em 3 níveis:

- JSE(*Java Standard Edition*): plataforma que contém a especificação da linguagem Java. Aqui fazem parte a linguagem, as bibliotecas disponíveis aos desenvolvedores e outros elementos como JVM(*Java Virtual Machine*) e *Garbage Collector*.
- JEE(*Java Enterprise Edition*): plataforma para desenvolvimento de sistemas corporativos. Utiliza a JSE como base para os serviços disponibilizados. Aqui fazem parte diversas API's e especificações como EJB, JPA.
- JME(*Java Mobile Edition*): plataforma para desenvolvimento para dispositivos móveis. Utiliza uma versão simplificada da API Java.

Para o projeto em questão, foi utilizada a JSE e JEE na versão 6. A JME foi substituída pelo *Android SDK* que será discutido posteriormente.

A linguagem Java é do tipo Interpretada. O processo de compilação

transforma o código fonte em *bytecodes* que são executados pelo elemento principal da plataforma: a JVM.

A JVM(Java Virtual Machine) é responsável por executar toda aplicação Java. Ela faz o controle de todo o ciclo de vida de uma aplicação Java e provê todo os controles necessários para a aplicação.

Devido a presença da JVM da plataforma Java, as aplicações desenvolvidas em Java são portáveis, ou seja, uma vez desenvolvida pode ser executada em qualquer Sistema Operacional que possua uma implementação da JVM. Atualmente os principais Sistemas Operacionais possuem JVM(*Windows*, *Linux* e *MacOS*). Essa característica se torna um diferencial importante na plataforma Java em relação as demais. Outro elemento importante da plataforma é o *Garbage Collector*. Em Java a tarefa de alocação/desalocação da memória para os objetos criados é da JVM. Ela utiliza o *Garbage Collector* para varrer os objetos criados e caso eles não sejam mais necessários, o mesmo é desalocado e a memória é liberada. Esse processo facilita bastante o desenvolvimento de aplicações, pois tira essa responsabilidade do desenvolvedor. Em linguagens que não possuem o GC(*Garbage Collector*) é bastante comum ocorrer *Memory Leaks* devido à não coleta de algum objeto da memória. Com o GC isso é amenizado. Outra vantagem da JVM é a otimização que a mesma realiza nos *bytecodes* gerados.

2.2.1 RPC

As chamadas remotas a procedimentos (*Remote Procedure Call* - RPC) representam um conceito que permite a execução de uma rotina em um processo externo(em outra aplicação/máquina/dispositivo). O processo que executa a rotina externa possui apenas uma interface para o processo remoto, que seria uma espécie de espelho do processo remoto. Toda a execução da rotina ocorre remotamente. Atualmente esse conceito é bastante utilizado em Sistemas Distribuídos, cada vez mais as aplicações são executadas de forma distribuída e com RPC é possível fazer a integração de várias aplicações distribuídas na rede.

Existem várias tecnologias que implementam o conceito de RPC, entre

elas podemos citar:

- Corba(*Common Object Request Broker Architecture*)
- Java RMI(*Remote Method Invocation*)
- *WebServices*
- EJB(*Enterprise Javabeans*)

Para o projeto em questão, foi utilizado RPC na forma de *WebServices* para a comunicação de dispositivos móveis com um servidor.

2.2.2 WebServices

Com *WebServices* é possível executar uma rotina externa através de algum protocolo de rede. Desde a sua idealização, os *WebServices* se tornaram referência para integração de sistemas distribuídos. Os *WebServices* também são referência para implementação de SOA(*Service Oriented Architecture*).

SOA é um novo paradigma para a construção de sistemas. Com SOA, a ideia é a criação de vários serviços que juntos formam uma aplicação. É uma arquitetura bastante flexível e que busca o reuso de componentes. Atualmente está sendo usado por muitas empresas no mercado que buscam a excelência em seus negócios. Porém, vale ressaltar que SOA é um conceito bastante amplo e que *WebServices* é apenas uma forma de prover SOA(SHEVAR, 2006).

Para o projeto em questão, foi utilizado *WebServices* sobre SOAP para a comunicação de dispositivos móveis com um servidor.

2.2.2.1 SOAP

O protocolo de acesso simples a objetos(*Simple Object Access Protocol* - SOAP) é um protocolo de comunicação baseado em XML que permite o transporte de informações sobre HTTP. Ele é utilizado para fazer a comunicação entre sistemas que não necessariamente foram desenvolvidos com a mesma linguagem. Pelo fato de SOAP utilizar XML e HTTP, a comunicação entre sistemas é facilitada pois qualquer linguagem que permite a comunicação com HTTP pode utilizar SOAP para comunicação com outros sistemas. Uma

mensagem SOAP é um documento XML que contém os elementos que definem a mensagem:

- *Envelope*: identifica que se trata de uma mensagem SOAP.
- *Header*: informações de cabeçalho.
- *Body*: contém o corpo da mensagem. É o elemento que contém as informações principais da mensagem.
- *Fault*: contém informações de erro. Quando ocorre um erro no *WebService*, as informações e detalhes do erro são enviadas nesse elemento.

Para o projeto em questão, o SOAP foi usado como protocolo de comunicação entre os aplicativos móveis e o servidor.

2.2.2.2 WSDL

A linguagem de definição de serviços web(*WebServices Definition Language* – WSDL) é um documento XML que define a estrutura de um WebService. O objetivo do WSDL é descrever as operações expostas pelo WebService a fim de criar um contrato de como a comunicação entre o cliente e o WebService deve ocorrer. O WSDL possui as informações necessárias para que um cliente possa invoca um método remoto:

- Tipos: define as estruturas de dados usados pelas operações.
- Operações: define as ações/operações expostas. Dentro de cada operação, são especificadas as entradas e saídas esperadas.
- Endpoint: define a URL onde o serviço está exposto.

2.2.2.3 REST

A transferência de estado representacional(*Representational State Transfer* - REST) é um estilo de arquitetura para sistemas distribuídos. Ao longo dos últimos anos surgiu como uma alternativa ao SOAP para desenvolvimento de *WebServices*. Ao contrário do SOAP, o REST não é um protocolo de comunicação e sim um estilo de arquitetura. Ele utiliza HTTP para a comunicação entre sistemas distribuídos.

O REST foi proposto por Fielding(2000), tendo sido concebido para a comunicação de sistemas *hypermedia* distribuídos, isto é, sistemas no qual texto, áudio, gráficos e outras mídias são armazenados e interconectados através de *hyperlinks*. A *World Wide Web* é um exemplo de um sistema *hypermedia*.

O elemento principal na arquitetura REST é o recurso. A ação a ser realizada sobre o recurso é informada pelos verbos HTTP(*Post, Put, Get* e *Delete*). Com isso, podemos realizar operações CRUD(*Create*, *Retrieve*, *Update* e *Delete*) sobre os recursos. A tabela a seguir representa o mapeamento entre os verbos HTTP e as operações CRUD.

Verbo HTTP	**Significado em termos CRUD**
POST	Criar um novo recurso
GET	Lê um recurso
PUT	Atualiza um recurso
DELETE	Exclui um recurso

Ao associar os verbos HTTP com os recursos(URI) você forma uma expressão lógica de comportamento. Ex: GET recurso X, DELETE recurso Y etc. Uma grande vantagem do REST é que os recursos podem ser representados de diversas formas. Por exemplo, um recurso pode ser obtido em qualquer formato que ele possa ser representado(XML, JSON, TXT, áudio, vídeo etc).

Para o trabalho em questão, o REST foi usado para consumir recursos do *Google Directions*, que possui toda sua API exposta via REST.

2.2.2.4 JAX-WS

A API Java para *WebServices* XML(*Java API for XML WebServices* - JAX-WS) é uma API para a criação de *WebServices* que utiliza XML/SOAP como forma de comunicação e faz parte da JEE. Com a JAX-WS é possível transformar uma classe com diversas operações em um *WebServices* e expô-lo na rede para que outras aplicações possam utilizar através de RPC. Desde a versão 5 da JEE, a JAX-WS utiliza anotações para a definição dos *WebServices* e suas operações.

Anotações foram criadas a partir da versão 5 da JEE para simplificar a criação de componentes como *WebServices*, EJB e *Servlets* etc. Exemplo de uso de anotações para a definição de um *WebServices* com JAX-WS:

```
@WebService(portName="CalcularFreteWSPort",serviceName="CalcularFreteWSService")
public class CalcularFreteWS{

	@EJB
	private FreteEJB freteEjb;

	@WebResult(name="valorFrete")
	public double calcularFrete(String cepOrigem, String cepDestino, String tipo) {
		return freteEjb.calcularFrete(cepOrigem, cepDestino, tipo);
	}
}
```

A anotação *@WebService* define a classe como sendo um *WebService*. Todas as operações de escopo *public* da classe serão expostas como operações do *WebService*. Quando a aplicação é implantada no servidor de aplicação, o mesmo identifica que se trata de um *WebService* e cria um *Endpoint* para a execução do Webservice. O *Endpoint* é o endereço onde o serviço pode ser localizado. Junto com o *Endpoint*, fica o WSDL, que é o contrato que especifica o *WebService*.

Com o WSDL, é possível criar uma aplicação cliente para a execução das operações do *WebService*. Essa aplicação cliente recebe o nome de *Stub*. O *Stub* é a aplicação que encapsula a invocação do *WebService* remoto. Em geral são utilizadas ferramentas que geram o código do *Stub* automaticamente, a partir do WSDL. O *Stub* possui todo o código necessário para abrir sockets HTTP, enviar a mensagem SOAP, obter a resposta e transformar em objetos. Entre as responsabilidades do Stub, estão:

- *Marshalling*: transforma o objeto de entrada do cliente em XML(*eXtensible Markup Language* – Linguagem de Marcação Extensível) para ser enviado pela rede.
- Transporte: envia a requisição e obtém a resposta utilizando mensagens SOAP.
- *Unmarshalling*: transforma o XML de retorno em objeto e devolve para o

cliente.

O protocolo SOAP é utilizado pelo JAX-WS para trafegar as informações do *WebService*.

Para o projeto em questão, o JAX-WS foi utilizado para desenvolvimento do *WebService* utilizado pelo aplicativo móvel para comunicação com o servidor.

2.2.2.5 Ksoap

O Ksoap é uma biblioteca desenvolvida em Java que permite que aplicativos móveis se comuniquem com *WebServices*. O Ksoap foi inicialmente desenvolvido para ser utilizado com JME porém recentemente foi adaptado para ser usado em aplicativos Android(ksoap2-android).

2.2.3 EJB

O *Enterprise Javabeans - EJB* é uma tecnologia Java que permite a criação de componentes de negócio que podem ser usados pelas aplicações Java. Desde a versão 5 do JEE, o EJB é configurado através de anotações:

- @Stateless
- @Stateful
- @Remote
- @Local
- @Singleton
- @Schedule

Existem 2 tipos de EJB: *Stateless* e *Stateful*. Em aplicações OO temos o conceito de estado do Objeto. O estado do Objeto é representado pelo valor de seus atributos. No EJB do tipo *Stateless*, o estado não é mantido no decorrer da utilização do EJB pelos clientes. Já o EJB do tipo *Stateful* mantém o estado do EJB no decorrer da utilização do mesmo pelo cliente. O EJB do tipo *Stateful* consome mais recursos do servidor, pois é necessário manter um objeto EJB específico para cada cliente que o utiliza. Isso significa que se tivermos 100 clientes em uma aplicação que utiliza *Stateful*, 100 objetos EJB serão necessários

para servir aos clientes. Já no caso do EJB do tipo *Stateless*, o servidor de aplicação mantém um pool de objetos EJB que são usados pelos clientes. Dessa forma, quando um cliente precisa utilizar o EJB, o servidor de aplicação busca um objeto EJB no pool e retorna para o cliente.

O EJB é uma tecnologia que implementa o RPC. Existem 2 tipos de escopo de acesso ao EJB: Remoto e Local. No tipo Local, o acesso é local na aplicação e exige que o cliente esteja na mesma JVM que o EJB. No tipo Remoto, a interface do EJB pode ser exposta e utilizada por um cliente que esteja em outra JVM e até em outra rede. Esse comportamento é definido pelas anotações @Remote e @Local. Existe ainda 1 tipos especial de EJB: Singleton.

Um EJB do tipo Singleton possui apenas uma instância/objeto para toda a aplicação. Pode ser usado para serviços que não mantém estado e que podem ser executados de forma concorrente. A configuração do EJB do tipo Singleton é feita utilizando a anotação @Singleton na classe do EJB.

Em aplicações corporativas, podemos ter a necessidade de ter métodos que são executados periodicamente. Em EJB, isso pode ser feito utilizando a anotação @Schedule. Essa anotação deve ser feita nos métodos que devem ser executados periodicamente. Essa anotação permite que seja informada as informações de quando o método deve ser executado: dia, hora, minuto e segundo.

O EJB provê uma série de serviços que podem ser usados, como por exemplo transações e persistência com JPA(Java Persistence API).

Sobre as transações, elas podem ser controladas pelo servidor de aplicação ou de forma manual. Se forem controladas pelo servidor de aplicação, caso ocorra um erro na invocação, o mesmo automaticamente desfaz as alterações realizadas na base de dados. Caso contrário, realiza a persistência na base de dados. Se o controle de transação for feito manualmente, a lógica do método EJB deve controlar o momento em que a transação deve ser efetivada ou desfeita.

Para o projeto em questão, foi utilizado EJB do tipo *Stateless* e com acesso *Local* para prover os principais métodos de negócio envolvidos na

aplicação. Também foram utilizados métodos para execução periódica para a execução do Despachador de Táxi. O controle de transação é realizado pelo servidor de aplicação.

2.2.4 ORM

O ORM(*Object Relational Mapping*) é uma técnica para fazer o mapeamento entre duas formas de representação diferentes: Orientação a Objetos x Banco de Dados Relacional. Ao longo dos anos, o uso de banco de dados relacionais em aplicações OO(Orientadas a Objetos) não se mostrou simples e transparente. Era preciso muito código para fazer o mapeamento entre as classes e tabelas na base de dados e isso tomava muito tempo no desenvolvimento de aplicações. Além disso, era necessário utilizar código SQL dentro da aplicação OO, implicando em uma dependência muito forte da aplicação com o banco de dados. Esses e outros fatores levaram ao surgimento de ferramentas de ORM.

2.2.4.1 Hibernate

O Hibernate foi a primeira ferramenta de ORM a se tornar popular na comunidade Java. Podemos dizer que foi por causa do Hibernate que o ORM tomou força a ponto de surgir a J*ava Persistence API* - JPA. O Hibernate foi criado pela empresa JBoss e foi desenvolvido em Java. Foi a primeira ferramenta de ORM a ser usada em larga escala pelo mundo corporativo.

Nas primeiras versões do Hibernate, o mapeamento de Classes para o mundo Relacional era feito utilizando XML. Depois da popularização do uso de Anotações para configuração de componentes em Java, o Hibernate também aderiu a idéia, o que simplificou o trabalho de mapeamento.

No Hibernate, cada classe e seus atributos são mapeados para tabelas e colunas. O Hibernate permite o mapeamento dos relacionamentos de Um-para-Um, Um-para-Muitos, Muitos-para-Muitos e Muitos-para-Um. Também permite que sejam utilizados conceitos de OO como polimorfismo e herança.

Para o trabalho em questão, foi utilizado Hibernate para persistência de

Objetos em base de dados.

2.2.4.2 JPA

A API Java para persistência(*Java Persistence API* – JPA) é a especificação do Java para a utilização de ORM. Com o sucesso e a popularização do Hibernate, várias outras ferramentas de ORM surgiram e como consequência disto a comunidade Java se viu motivada a criar uma especificação para o ORM dentro da plataforma Java. Com isso, surgiu a JPA, que veio com o objetivo de padronizar as ferramentas de ORM.

A ideia por traz do JPA era tratar a persistência de maneira padronizada, de forma que qualquer ferramenta de ORM pudesse ser usada sem alterar o código fonte das aplicações. O JPA também trouxe a transparência na utilização de banco de dados. As implementações de JPA foram desenvolvidas para trabalhar com a maioria dos bancos de dados do mercado. Dessa forma, através de configuração específica da implementação de JPA podemos informar qual banco de dados será usado. Isso ajuda a diminuir o acoplamento das aplicações com o banco de dados, trazendo mais flexibilidade para as aplicações. O JPA hoje é um sucesso na comunidade e já está na versão 2.0.

A configuração do mapeamento é feita através de anotações nas classes que devem ser persistidas. Cada classe que deve ser persistida deve ter uma anotação *@Entity*. O JPA trabalha com o conceito de *Configuration By Exceptio*n. Esse conceito simplifica a configuração do mapeamento. Caso não seja informado o nome da tabela na base de dados, é utilizado o nome da classe. Caso não seja informado/anotado os atributos da classe, assume-se que os nomes de colunas serão iguais aos nomes dos atributos da classe. Esse conceito faz com que o tempo de configuração do mapeamento seja bastante reduzido. O JPA permite polimorfismo e herança. Os relacionamentos entre as classes são mapeados pelas anotações:

- @OneToMany
- @ManyToOne
- @OneToOne

- @ManyToMany

O elemento principal do JPA é o EntityManager. Ele é responsável por realizar as operações de persistência:

- *find*: busca entidades na base de dados, já fazendo a transformação para Objeto.
- *persist*: faz a persistência de uma entidade na base de dados.
- *merge*: atualiza a entidade na base de dados.
- *refresh*: atualiza a entidade com as informações da base de dados.
- *remove*: remove a entidade da base de dados.

Ele também é responsável por obter a conexão com o banco de dados e abrir transações. O JPA possui uma configuração em XML que deve ser fornecida para seu funcionamento. Essa configuração é definida no arquivo *persistence.xml* e provê as informações de:

- *Persistence Provider:* caminho para a classe que implementa o JPA.
- Informações para conexão com a base de dados: usuário, senha, string de conexão.
- Parâmetros específicos para a ferramenta de ORM.

Exemplo de uma entidade mapeada com JPA:

```
@Entity
@SequenceGenerator(sequenceName="SEQ_CORRIDA", name="SEQ_CORRIDA")
public class Corrida implements Serializable {
        @Id
      @GeneratedValue(generator="SEQ_CORRIDA",strategy=GenerationType.SEQUENCE)
        private Long id;

        @ManyToOne
        @JoinColumn(name = "ID_USUARIO")
        private Usuario usuario;

        @ManyToOne(cascade = CascadeType.MERGE)
        @JoinColumn(name = "ID_TAXISTA")
        private Taxista taxista;

        @Column(nullable = false)
        @Temporal(TemporalType.TIMESTAMP)
        private Calendar dataSolicitacao;

        @Column(nullable = false)
        @Enumerated(EnumType.STRING)
```

```
        private StatusCorrida status;
        ... ...
}
```

Para o trabalho em questão, foi utilizado JPA na versão 2.0 juntamente com o Hibernate para a camada de persistência.

2.2.5 JSON

O *JavaScript Object Notation* - JSON é um formato leve para transferência de dados entre aplicações distintas. Embora o nome indique ser para JavaScript, o JSON atualmente possui uma abrangência bem maior. O JSON é uma alternativa ao formato XML e tem a vantagem de ser mais leve, ou seja, o a carga de dados para transferência de informações é menor devido ao JSON ser um formato reduzido se comparado ao XML.

O JSON é bastante utilizado para troca de informações em requisições AJAX e recentemente tem sido muito utilizado em arquiteturas do tipo REST como formato de envio e recebimento de mensagens.

Atualmente o JSON pode ser usado em praticamente qualquer linguagem. Existem muitas API's desenvolvidas para várias linguagens. Para Java existem dezenas de API's que fazem a leitura e transformação de JSON para objetos Java. Devido a força que JSON vem adquirindo recentemente, uma JSR foi proposta para padronizar as operações de parse e transformações sobre formato JSON na plataforma Java.

Exemplo de um objeto representado em JSON:

```
{
  user: {
    id: 123456,
    name: "Clayton Kuhn",
    username: "claytonedk",
    phone: "4133332222"
  }
}
```

Exemplo de um array representado em JSON:

```
{
   "notas" : [
      {"nome":"João", "nota1":8, "nota2":6, "nota3":10 },
      {"nome":"Maria", "nota1":5, "nota2":9, "nota3":8 }
```

```
  ]
}
```

A principal vantagem do JSON em relação ao XML é o formato reduzido que ele utiliza.

Para o projeto em questão, o JSON foi utilizado como formato de dados nas chamadas da API de mapas do Google.

2.2.6 Google Directions

A *Google Directions* é uma API para cálculo de rotas e distâncias. Ele calcula as rotas disponíveis entre dois pontos.

O *Google Directions* fornece a API em forma de serviços REST. A entrada do serviço é passada como parâmetros GET na URL e o retorno pode ser em formato XML ou JSON.

Exemplo de request:

http://maps.googleapis.com/maps/api/directions/json?origin=-25.435054,-49.269331&destination=-25.473692,-49.215317&sensor=true&units=metric&language=pt-BR

Onde:
origin = coordenadas do ponto de origem
destination = coordenadas do ponto de destino
units = unidade de medida
language = linguagem a ser usada nas informações de resposta

Para o trabalho em questão, foi utilizado o Google Directions para calcular a distância entre o táxi e o cliente.

2.2.7 Android

O Android é um Sistema Operacional para dispositivos móveis. Foi inicialmente desenvolvido e concebido pela Google e atualmente é mantido pela Open Handset Alliance, que é formada por várias empresas com interesse na plataforma Android, como Intel, Google, Samsung, Motorola etc.

2.2.7.1 Plataforma Android

O Android é uma plataforma baseada em linux e o desenvolvimento para o mesmo é feito utilizando linguagem Java em conjunto com o Android SDK. Por utilizar Java, o Android possui uma máquina virtual chamada Dalvik. A Dalvik possui um funcionamento um tanto diferente da máquina virtual Java para Desktop. Primeiramente, ela não utiliza diretamente as classes compiladas(.class). A Dalvik faz mais um passo para transformar o .class em .dex, que é o formato otimizado de compilação para o Android. Outro diferencial da Dalvik é que ela é executada por aplicação, ou seja, cada aplicação em execução possui uma instância da Dalvik em execução. Com isso, o Android consegue isolar as aplicações e impedir que algum comportamento inesperado nas aplicações possam interferir em outras. Porém para que fosse possível a utilização dessa abordagem, a Dalvik chegou em um nível de otimização bastante elevado.

O Android provê diversas API's para o desenvolvimento de aplicações móveis. Como por exemplo, podemos citar:

- SQLite: Banco de dados gratuito e bastante leve que está incluso no Android.
- Google API's: API's do Google para trabalhar com mapas, buscas, GPS.
- WebKit: Framework usado para renderização de conteúdo HTML.
- Multimídia: API's para manipular e tratar conteúdo multimídia.
- Hardware: API's para tratar os diversos componentes de um dispositivo móvel, como por exemplo câmera, GPS e acelerômetro.
- OpenGL: API para criação de efeitos e jogos utilizando OpenGL.

2.2.7.2 Visão Geral da Arquitetura

O Android é uma plataforma composta pelo sistema operacional baseado em linux e um conjunto de bibliotecas e aplicativos que o acompanham. Na figura 3 podemos ver as principais camadas da plataforma Android.

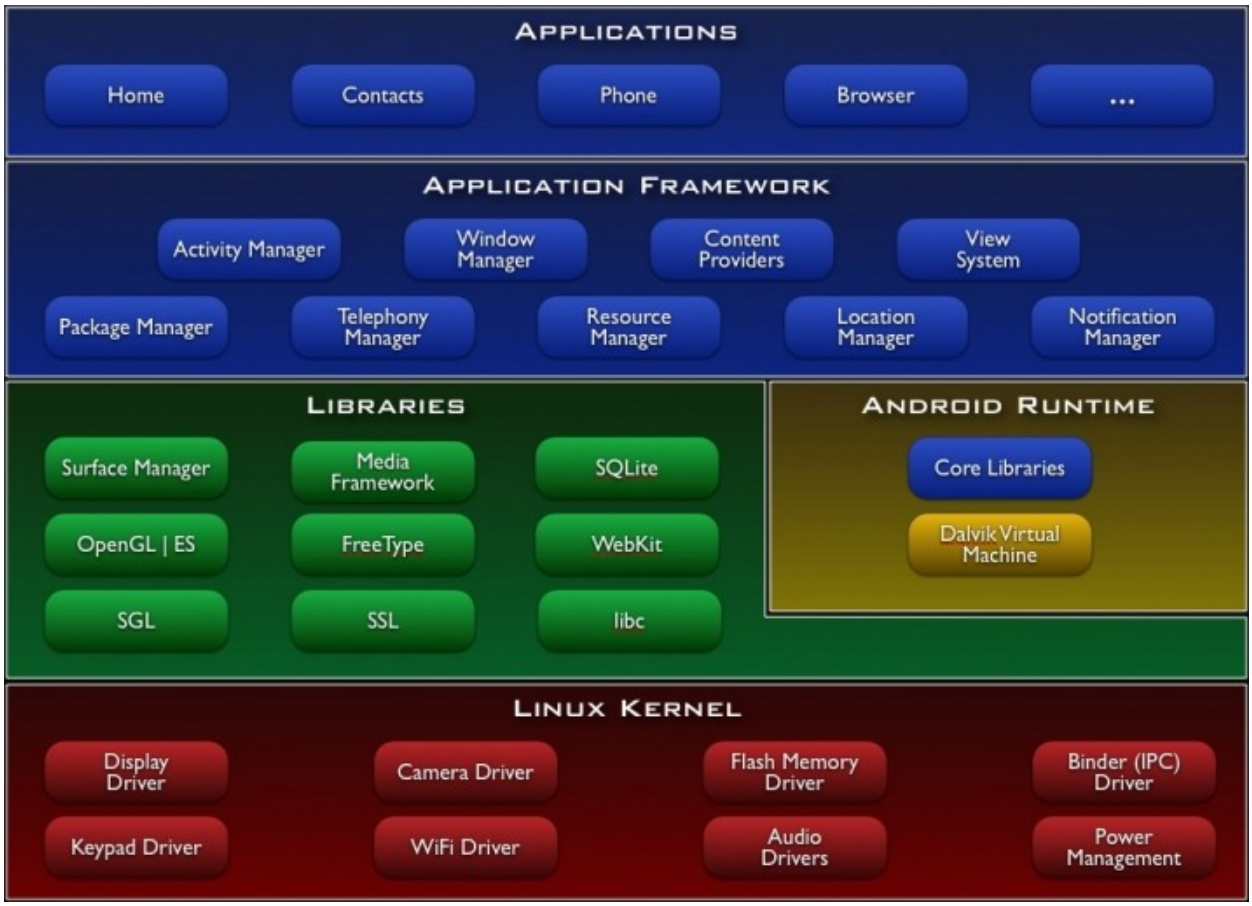

Figura 2: Arquitetura da plataforma Android.
Fonte: Conhecendo o Android.

Applications: camada onde são executadas os aplicativos do Android, tanto as nativas como Navegador, Contatos, Calculadora etc., quanto aplicativos de terceiros como o protótipo aqui proposto.

Application Framework: camada onde estão os serviços utilizados pelos aplicativos Android, como por exemplo o *Activity Manager*(ciclo de vida da aplicação), *Location Manager*(localização por GPS ou Rede), *Content Providers*(provedores de conteúdos, ex: Contatos).

Libraries: camada onde estão algumas API's desenvolvidas em C/C++ e que oferecem recursos como renderização 3D(OpenGL ES), banco de dados(SQLite), WebKit(renderização de conteúdo Web).

Android Runtime: camada onde estão um conjunto de *core libraries* que provém a maioria das funcionalidades disponíveis nas *core libraries* do Java. Também contem a Dalvik VM, que provê todo o ambiente necessário para a execução de

uma aplicação *Android*.

Linux Kernel: camada onde estão os serviços de baixo nível do Sistema Operacional *Android*. Contém serviços essenciais como segurança, gerenciamento de memória, gerenciamento de processos e *drivers* de dispositivos.

2.2.7.3 Android SDK

O *Android SDK* provê as ferramentas necessárias para o desenvolvimento de aplicativos *Android*. Está disponível para *Windows*, *Linux* e *MacOS*. Entre as ferramentas incluídas no *Android SDK* podemos citar o depurador, bibliotecas, emulador de *smartphone*, documentação, códigos de exemplo e tutoriais.

2.2.7.4 Componentes de um Aplicativo Android

Figura 3: Componentes de um Aplicativo Android
Fonte: Conhecendo o Android.

AndroidManifest: arquivo XML existente em todos os aplicativos Android. Possui informações sobre as Activities, Services bem como as permissões necessárias para o funcionamento do aplicativo.

Activities: representam as telas da aplicação, onde é necessário interação com usuário. Geralmente possui uma *view* associada, que contém o layout da tela. São responsáveis por tratar os eventos gerados pela tela bem como coordenar o fluxo do aplicativo.

Services: são serviços que são executados em segundo plano. Não possuem

uma *view* associada, já que não possuem interface gráfica. Normalmente são usados para tarefas que demandam um tempo de execução grande e que não realizam interação direta com usuário. Ex: Aplicativo de música executando em segundo plano.

Broadcast Receivers: são componentes que tratam eventos gerados pelo Android. Os eventos podem ser nativos ou gerados por aplicativos. Como exemplos de eventos, podemos citar recebimento de uma ligação ou SMS, ou carga da bateria abaixo de determinado nível.

Content Providers: são componentes que compartilham conteúdo entre aplicativos Android. Como exemplo, podemos citar o gerenciamento de contatos nativo do Android, que oferece um *Content Provider* que pode ser usado para buscar e atualizar contatos.

Intents: são mensagens geradas para o Android que tem por objetivo gerar uma ação no sistema. Como exemplo, podemos citas a troca de *Activity*, utilizar um aplicativo externo para visualizar um determinado conteúdo, entre outros.

Views: são componentes usados para definir os elementos gráficos exibidos na tela. Como exemplos, podemos citar botões, caixas de texto e mapas.

2.2.7.5 Ciclo de Vida de um Aplicativo Android

O componente base de um aplicativo *Android* é a *Activity*. Ela é associada a um layout de tela e controla o fluxo da aplicação conforme as ações do usuário. A figura 5 mostra o ciclo de vida de uma *Activity*.

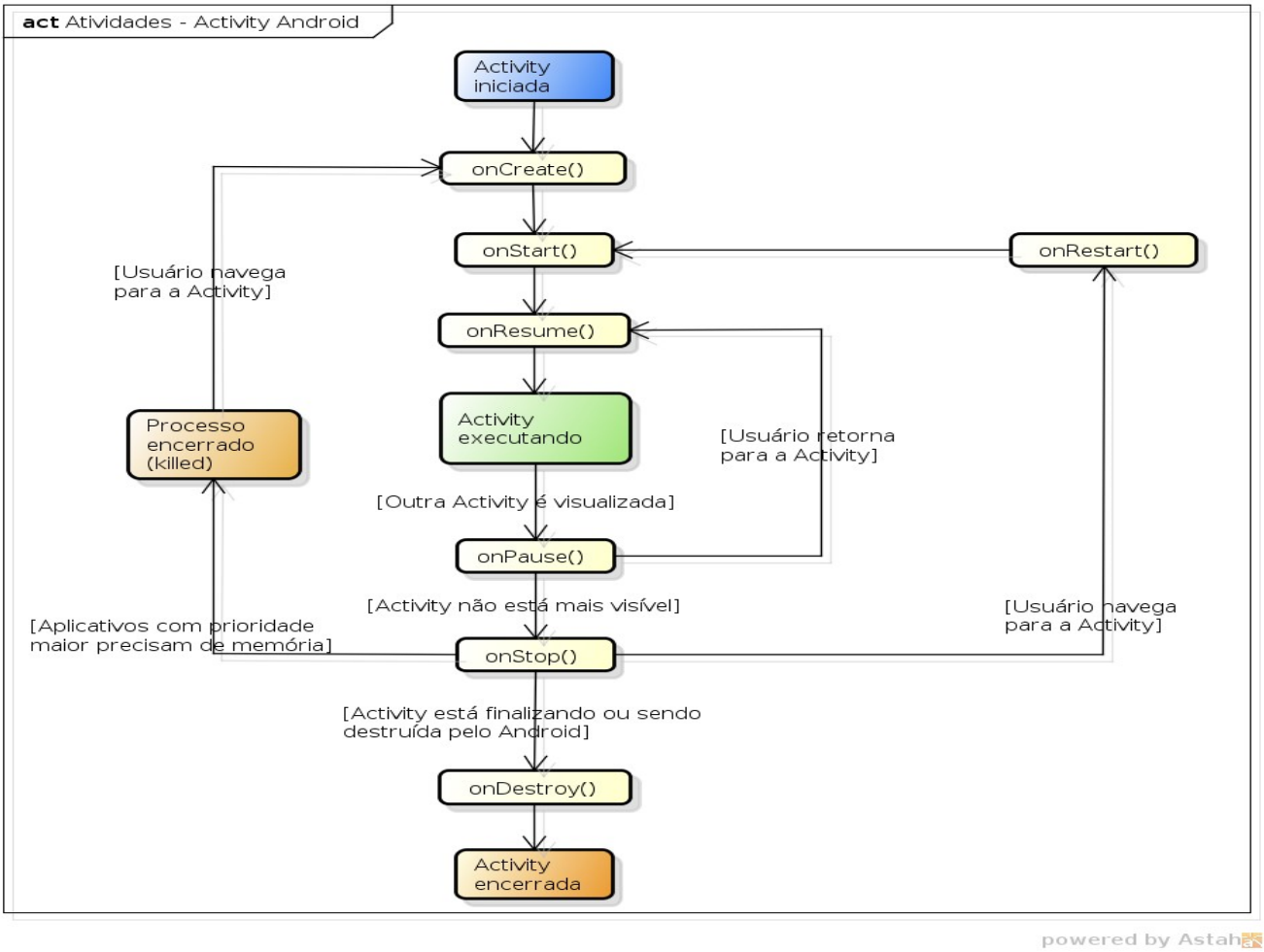

Figura 4: Ciclo de vida de uma *Activity*
Fonte: Autoria própria

Para cada fase do ciclo de vida de uma *Activity*, o Android oferece um método que pode ser sobrescrito e realizar os tratamentos necessários. A seguir serão listados e descritos os métodos dos ciclos de vida de uma *Activity*.

onCreate()	Executado na primeira vez que uma *Activity* é criada. Aqui devemos inicializar a parte estática de uma *Activity*, como por exemplo associar com um layout.
onRestart()	Executado quando uma *Activity* parou de executar e foi executada novamente.
onStart()	Executado logo após a *Activity* se tornar visível ao usuário.
onResume()	Executado quando a *Activity* inicia a interação com o usuário.
onPause()	Executado quando o Android está iniciando outra *Activity*. É usado para parar a execução de processos que consomem

	CPU, memória e rede. Deve ter execução rápida pois a outra *Activity* só será iniciada quando a atual ser paralisada.
onStop()	Executado quando a *Activity* não está mais visível ao usuário.
onDestroy()	Executado antes da *Activity* ser destruída. É a chamada final que a *Activity* receberá. Pode ser executado devido a uma chamada ao método finish() ou quando o Android necessita de memória e finaliza a *Activity*.

Como citado acima, em alguns casos o Android pode precisar de memória e finalizar automaticamente uma *Activity*. Nesse caso, quando a *Activity* for iniciada novamente, será como se ela nunca tivesse existido. Isso pode ser ruim pois sempre desejamos que o usuário veja a mesma tela de aplicação que estava aberta na última vez que ele a visualizou. Para esses casos, o Android oferece dois métodos para salvar e restaurar o estado de uma *Activity*.

protected void onSaveInstanceState(Bundle outState): executado antes de uma *Activity* ser finalizada e removida da memória(*onDestroy*()). Armazena os dados necessários no objeto *outState.*

protected void onRestoreInstanceState(Bundle savedInstanceState): executado quando uma *Activity* é criada(*onCreate*()). O objeto *savedInstanceState* contém os dados salvos no método *onSaveInstanceState*().

Esses métodos foram utilizados nas *Activities* do aplicativo móvel, para guardar sempre o último estado de cada *Activity*. Dessa forma, se o Android finalizar uma *Activity* do aplicativo móvel, quando voltarmos para o aplicativo o último estado válido será restaurado na tela.

3. Desenvolvimento do protótipo

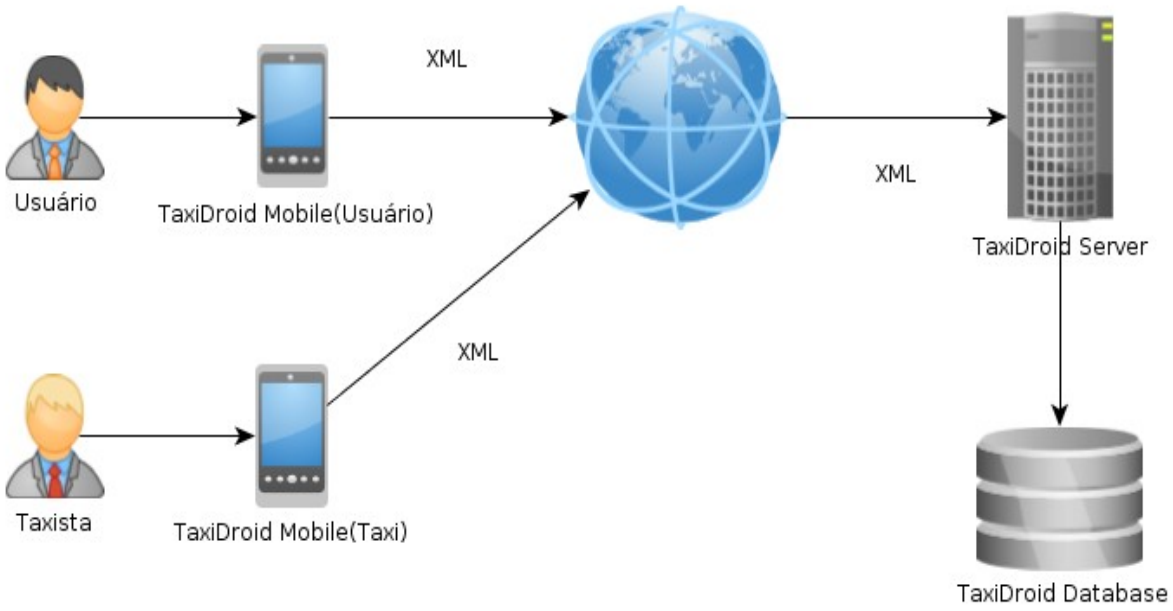

Figura 5: Arquitetura geral do protótipo.

Para o desenvolvimento do protótipo proposto, foi utilizada a tecnologia Java JEE 6 para o servidor e Android para o cliente mobile. Dessa forma, o protótipo foi desenvolvido para funcionar em *smartphones Android* que tenham algum tipo de conexão com a rede através de tecnologias como Wi-Fi ou 3G.

Na Figura é apresentada a arquitetura geral do protótipo. Nela, temos os principais componentes da arquitetura:

Componente	Descrição
TaxiDroid Mobile(Usuário)	Aplicativo utilizado pelo Usuário para realizar as solicitações de atendimento para a central de táxi.
TaxiDroid Mobile(Taxista)	Aplicativo utilizado pelo Taxista para receber as solicitações de atendimento dos usuários.
TaxiDroid Server	Servidor que oferece todos os serviços utilizados pelos aplicativos dos Usuários e Taxistas. É responsável por tratar o ciclo de vida das solicitações.
TaxiDroid Database	Base de dados que contém todas as informações dos usuários, taxistas e solicitações de atendimento.

3.1 Requisitos

Requisito	Descrição
1. Realizar login	O sistema deve oferecer autenticação do usuário/taxista através de login. Deve ser validado usuário e senha e o status do usuário/taxista no sistema.
2. Informar posição no mapa	O sistema deve permitir ao usuário que o mesmo possa informar sua posição exata no mapa. Esse requisito é para os clientes que não possuem GPS no celular, sendo que a posição não será tão exata.
3. Solicitar atendimento	O sistema deve permitir ao usuário que após obtida a localização seja possível solicitar um táxi pelo celular. Outra opção é utilizar um local salvo nos favoritos.
4. Acompanhar atendimento	O sistema deve permitir ao usuário/taxista acompanhar o status de sua solicitação de atendimento. Dessa forma, de tempos em tempos o sistema deve buscar atualizações da solicitação de atendimento.
5. Cancelar atendimento	O sistema deve permitir ao usuário/taxista o cancelamento de um atendimento solicitado, desde que a solicitação não esteja encerrada. Deve ser informado o motivo do cancelamento.
6. Salvar favorito	O sistema deve oferecer ao usuário a opção de guardar o local de origem nos favoritos. Após salvo, nas próximas solicitações de atendimento o local favorito deve constar para seleção.
7. Encerrar atendimento	O sistema deve oferecer ao taxista a opção de encerrar o atendimento quando o mesmo está atendendo um cliente.
8. Encerrar expediente	O sistema deve oferecer ao taxista a opção de encerrar o expediente. Essa ação desconecta o taxista na central e impede que o mesmo receba despachos de novos atendimentos.
9. Aceitar atendimento	O sistema deve oferecer ao taxista a opção de aceitar um atendimento despachado.
10. Recusar atendimento	O sistema deve oferecer ao taxista a opção de recusar um atendimento despachado. O recuso pode ser pelo motivo de o expediente estar se encerrando.

3.3 Casos de Uso

Um caso de uso representa uma interação que um Ator realiza com o Sistema. O objetivo do diagrama de Casos de Uso é mostrar as possibilidades de interação do usuário com o sistema. A figura 7 demonstra o diagrama de casos de uso referente ao aplicativo móvel.

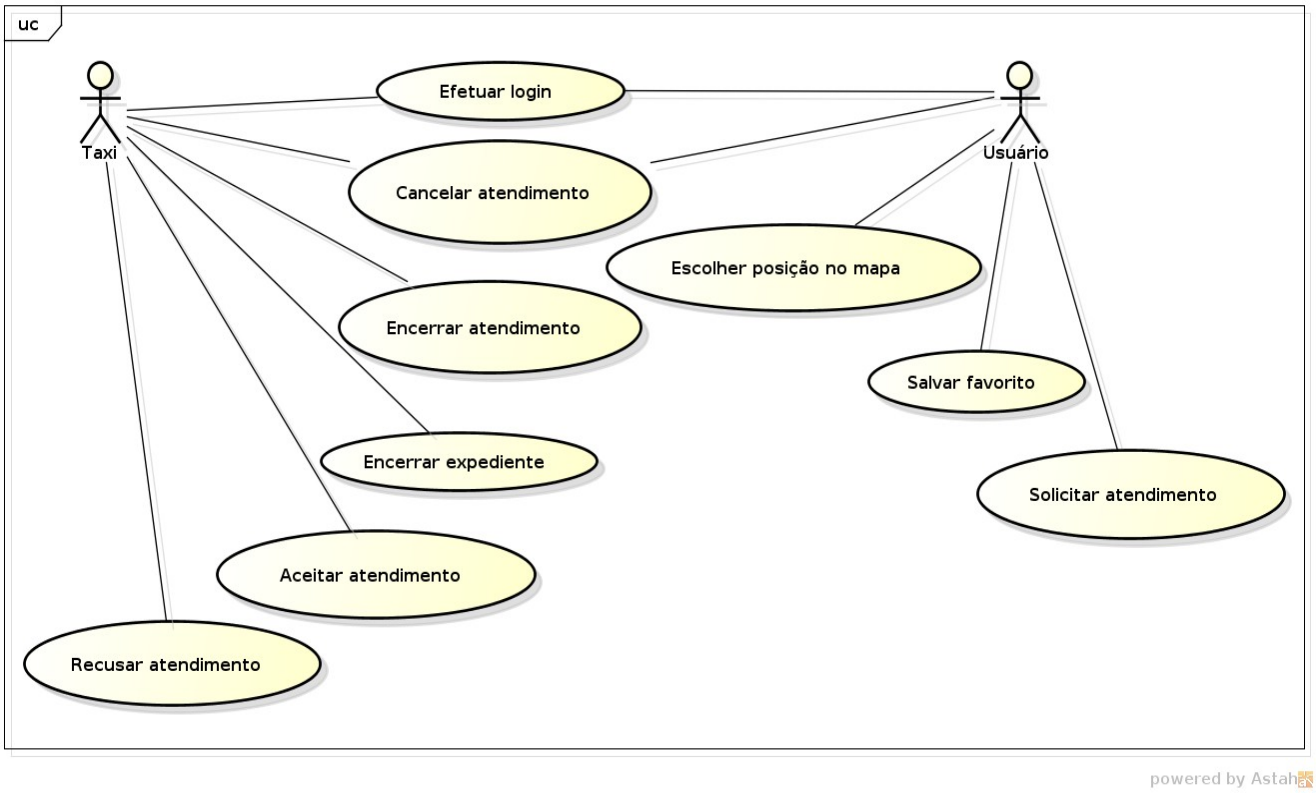

Figura 6: Casos de Uso do aplicativo móvel.

Descrição	Autenticação do usuário no sistema
Fluxo	1. Usuário/Taxista inicia o seu aplicativo móvel. 2. Aplicativo abre com a tela de login. 3. Usuário preenche informações de usuário e senha e aperta o botão “Entrar”. 4. Sistema valida o usuário/taxista e redireciona para a tela principal.
Pré condições	*Smartphone* deve possuir alguma conexão de rede.
Pós condições	Usuário/Taxista autenticado no sistema.

Descrição	Solicitar atendimento

Fluxo	1. Usuário aperta botão “Nova solicitação”. 2. Usuário aguarda posição ser obtida da rede ou GPS. 3. Usuário informa o número do local e se necessário preenche informações de complemento e observações. 4. Usuário aperta o botão “Solicitar”.
Fluxo alternativo	1. Usuário aperta botão “Nova solicitação”. 2. Usuário aguarda posição ser obtida da rede ou GPS. 3. Caso a localização obtida não seja exata, usuário aperta botão “Mapa”. 4. Usuário seleciona a rua no mapa e clica em “Selecionar”. 5. Usuário informa o número do local e se necessário preenche informações de complemento e observações. 6. Usuário aperta o botão “Solicitar”.
Fluxo alternativo 2	1. Usuário escolhe um local salvo nos favoritos. 2. Usuário aperta botão “Selecionar”.
Pré condições	1. *Smartphone* deve possuir alguma conexão de rede. 2. Usuário deve estar autenticado no sistema.
Pós condições	Atendimento solicitado. Usuário redirecionado para a tela de acompanhamento da solicitação.

Descrição	Cancelar atendimento
Fluxo	1. Usuário aperta botão “Cancelar”. 2. Usuário seleciona o motivo de cancelamento.
Pré condições	1. *Smartphone* deve possuir alguma conexão de rede. 2. Usuário deve estar autenticado no sistema. 3. Usuário deve possuir um atendimento em andamento.
Pós condições	Solicitação cancelada. Usuário redirecionado para a tela inicial.

Descrição	Salvar favorito
Fluxo	1. Solicitação é encerrada pelo Taxista. 2. Usuário recebe notificação do encerramento e a opção se deseja salvar o local nos favoritos. 3. Usuário escolhe opção “Sim”. 4. Usuário informa o nome do local. 5. Usuário aperta botão “Salvar”.
Pré condições	1. *Smartphone* deve possuir alguma conexão de rede. 2. Usuário deve estar autenticado no sistema. 3. Solicitação deve ser encerrada pelo Taxista.

Pós condições	Local salvo nos favoritos e disponível para seleção futura.

Descrição	Cancelar atendimento
Fluxo	1. Taxista aperta botão “Cancelar”. 2. Taxista seleciona o motivo de cancelamento.
Pré condições	1. *Smartphone* deve possuir alguma conexão de rede. 2. Taxista deve estar autenticado no sistema. 3. Taxista deve possuir um atendimento em andamento.
Pós condições	Solicitação cancelada. Tela do taxista reiniciada para receber novas requisições.

Descrição	Encerrar atendimento
Fluxo	1. Taxista chega até o local de destino do usuário. 2. Taxista aperta botão “Encerrar”.
Pré condições	1. *Smartphone* deve possuir alguma conexão de rede. 2. Taxista deve estar autenticado no sistema. 3. Taxista deve possuir um atendimento.
Pós condições	Solicitação encerrada. Tela do taxista reiniciada para receber novas requisições.

Descrição	Encerrar expediente
Fluxo	1. Taxista aperta botão de menu do smartphone. 2. Taxista aperta botão “Encerrar expediente”.
Pré condições	1. *Smartphone* deve possuir alguma conexão de rede. 2. Taxista deve estar autenticado no sistema.
Pós condições	Taxista desconectado no sistema. Aplicação é encerrada.

Descrição	Aceitar / Recusar atendimento
Fluxo	1. Taxista recebe despacho de atendimento. 2. É oferecida a opção de aceitar ou recusar o atendimento. 3. Taxista escolhe a opção.
Pré condições	1. *Smartphone* deve possuir alguma conexão de rede. 2. Taxista deve estar autenticado no sistema. 3. Taxista deve ter recebido o despacho de um atendimento.
Pós condições	1. Taxista aceita o atendimento e a solicitação muda para o status “EM ATENDIMENTO” no sistema.

	2. Taxista recusa o atendimento e a solicitação é enfileirada novamente no sistema.

3.4 Diagrama de Sequência

O diagrama de sequência têm por objetivo demonstrar as interações que ocorrem entre os componentes de um sistema. A figura 7 mostra as interações que ocorrem durante uma operação de solicitação de atendimento.

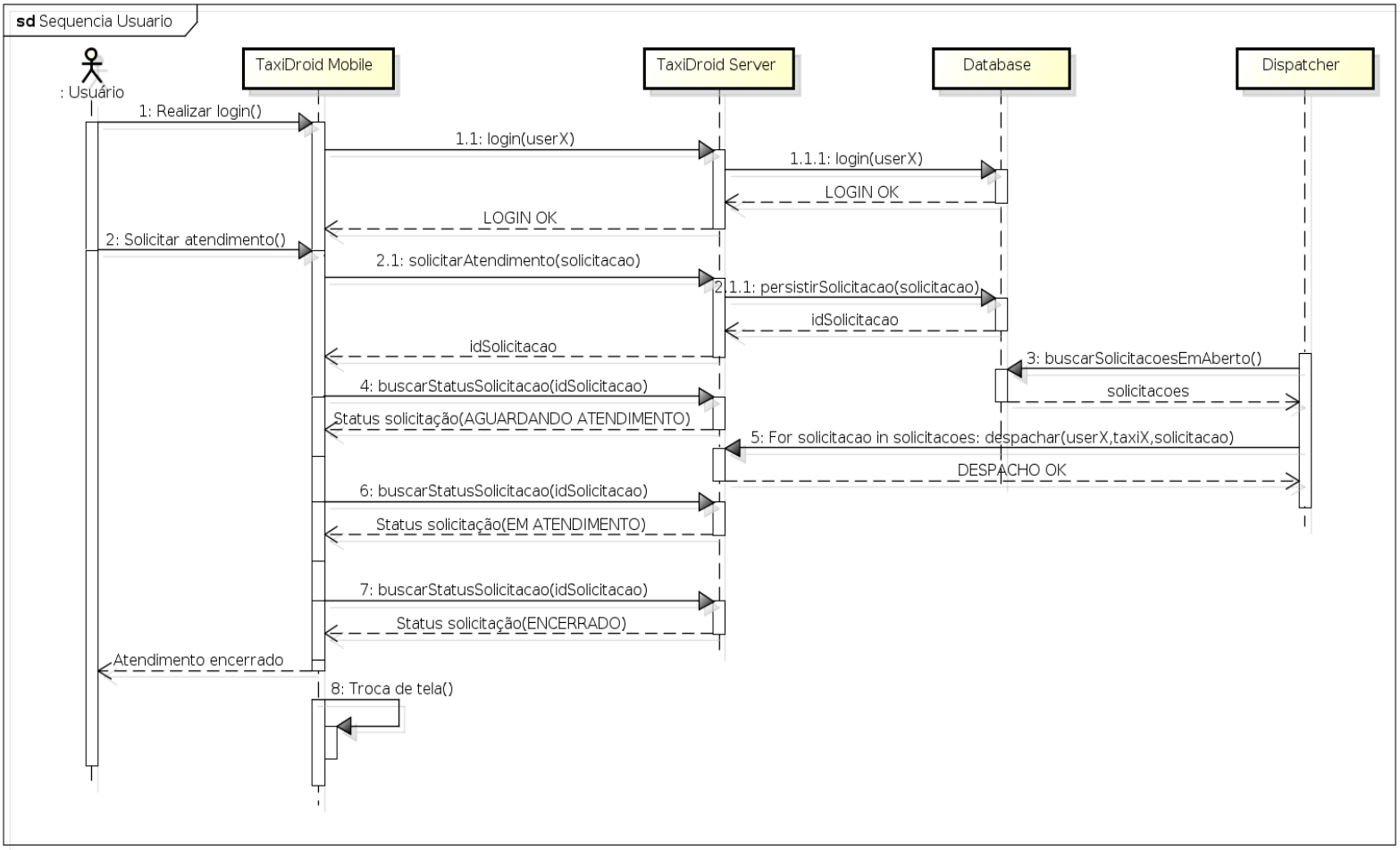

Figura 7: Diagrama de Sequência de uma solicitação de atendimento.

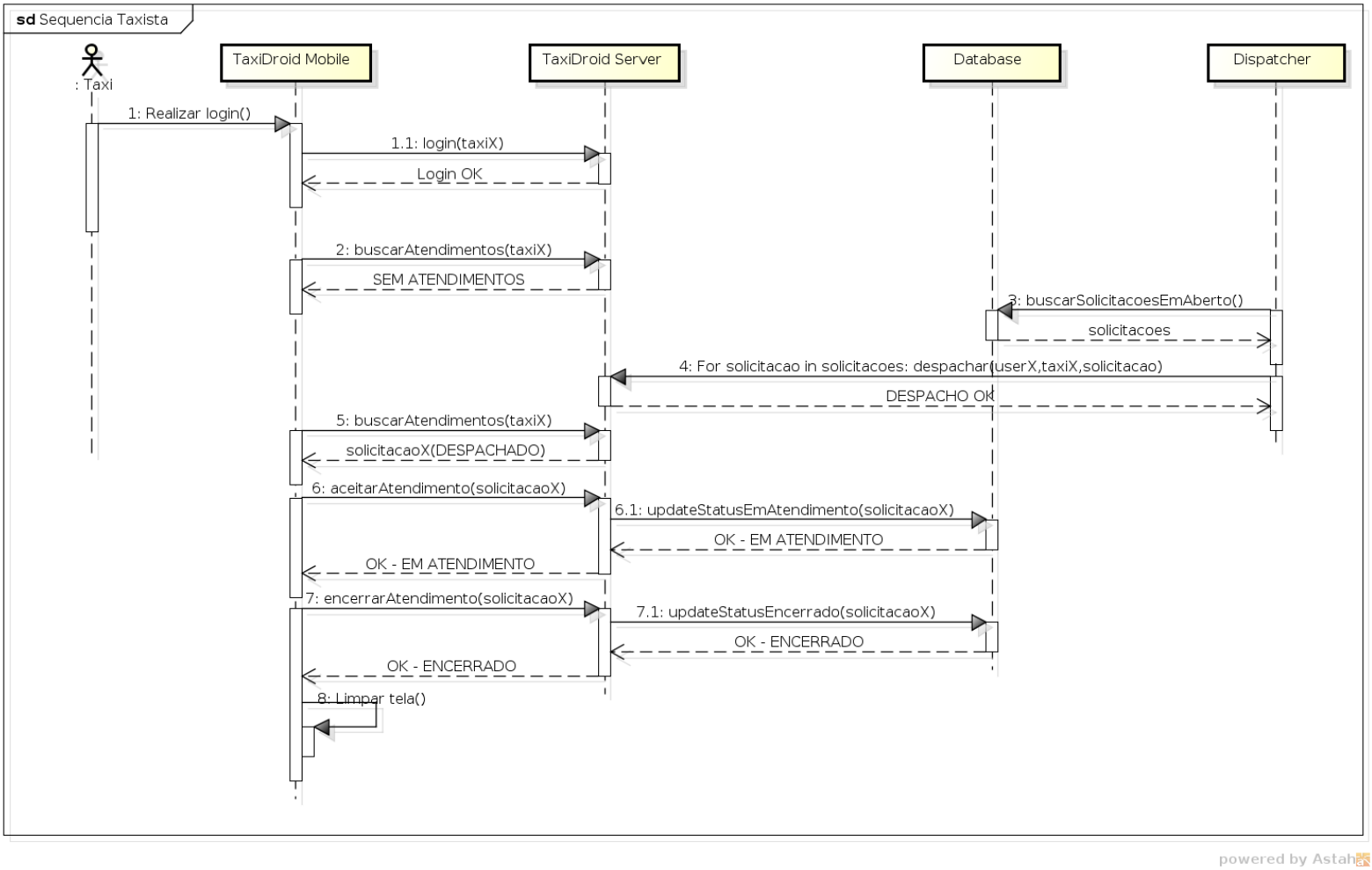

Figura 8: Diagrama de Sequência de um atendimento do Taxista.

3.5 Implementação

3.5.1 Servidor

Na implementação do servidor, foi utilizada a plataforma JEE na versão 6. A seguir temos o diagrama de classes do servidor.

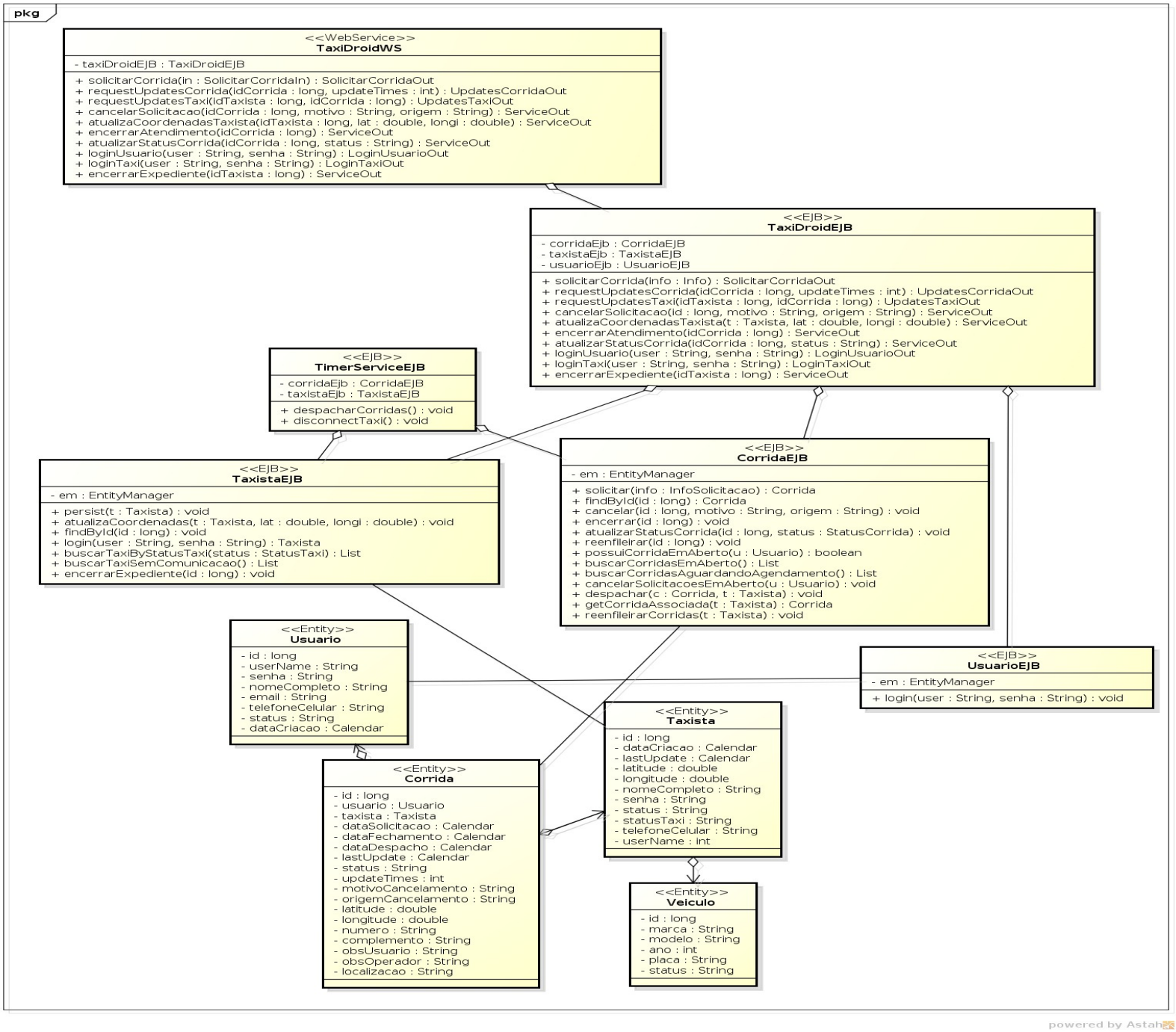

Figura 9: diagrama de classes do servidor

O diagrama acima representa as principais Classes utilizadas na camada servidor. Essas classes provém os serviços necessários para as operações no sistema.

3.5.1.1 Entidades

Quando se utiliza JPA, as entidades representam Classes que são mapeadas para persistência na base de dados. Para o desenvolvimento do protótipo, foram criadas 4 entidades:

- **Usuário**: Classe do tipo entidade *JPA* que representa um usuário no sistema. É uma entidade que possui mapeamento de persistência na base de dados.
- **Corrida**: Classe do tipo entidade *JPA* que representa uma solicitação de atendimento no sistema. É uma entidade que possui mapeamento de persistência na base de dados.
- **Taxista**: Classe do tipo entidade *JPA* que representa um taxista no sistema. É uma entidade que possui mapeamento de persistência na base de dados.
- **Veículo**: Classe do tipo entidade *JPA* que representa um veículo associado a um taxista no sistema. É uma entidade que possui mapeamento de persistência na base de dados.

3.5.1.2 **WebServices**

Para o desenvolvimento do protótipo, tínhamos o requisito de Sistema Distribuído pois o aplicativo móvel teria que se comunicar com o servidor através da rede de internet. Com isso, utilizamos *WebService* SOAP para a comunicação do aplicativo móvel com o servidor. Para o desenvolvimento do protótipo , foi criado 1 *WebService* com 10 operações:

- **TaxiDroidWS**: Classe do tipo *WebService* que expõe as operações utilizadas pelo aplicativo móvel. Essa classe apenas expõe as operações em forma de serviços. Ela utiliza os EJB's para realizar as operações de negócio.

3.5.1.3 EJB

Os EJB's são componentes a nível de servidor que foram desenhados para tratar as regras de negócio de um sistema. Eles oferecem diversas facilidades como segurança, transações e persistência. No protótipo desenvolvido, o *WebService* funciona apenas como um *Facade* expondo os métodos para o aplicativo móvel. Quem realiza todo o trabalho são os EJB's.

Para o desenvolvimento do protótipo, foram criados 5 EJB's responsáveis por realizar as operações.

- **TaxistaEJB**: Classe do tipo EJB *Stateless* que é responsável pelas operações de negócio envolvendo a entidade Taxista.

Método	Descrição
Persist	Insere ou atualiza uma entidade do tipo Taxista na base de dados.
AtualizaCoordenadas	Atualiza as coordenadas do taxista na base de dados.
FindById	Busca um Taxista com o ID informado.
Login	Realiza o login, validando se o usuário e senha do Taxista estão corretos.
BuscarTaxistaByStatusTaxi	Busca taxistas na base de dados com status do táxi informado.
BuscarTaxistaSemComunicacao	Busca taxistas na base que estão sem comunicação a mais de X tempo.
EncerrarExpediente	Atualiza o status do táxi para DESCONECTADO, encerrando o expediente.

- **CorridaEJB**: Classe do tipo EJB *Stateless* que é responsável pelas operações de negócio envolvendo a entidade Corrida.

Método	Descrição
Solicitar	Insere uma entidade do tipo Corrida na base de dados.
FindByID	Busca uma Corrida com o ID informado.
Cancelar	Cancela uma Corrida na base de dados.
Encerrar	Encerra uma Corrida na base de dados.
AtualizarStatusCorrida	Atualiza o status de uma Corrida na base de dados.
Reenfileirar	Reenfileira a Corrida, atualizando o status para AGUARDANDO ATENDIMENTO.
PossuiCorridasEmAberto	Verifica se o usuário possui alguma corrida em aberto.
BuscarCorridasEmAberto	Retorna as corridas do usuário que estejam em aberto.
CancelarSolicitacoesEmAberto	Cancela todas as corridas em aberto para o usuário.
Despachar	Despacha uma corrida para o taxista informado.
GetCorridaAssociada	Retorna a corrida associada ao Taxista, caso exista.
ReenfileirarCorridas	Reenfileira as corridas de um Taxista, atualizando o status para AGUARDANDO ATENDIMENTO.

- **UsuarioEJB**: Classe do tipo EJB *Stateless* que é responsável pelas

operações de negócio envolvendo a entidade Usuario.

Método	Descrição
Login	Realiza o login, validando se o usuário e senha do Usuário estão corretos.

- **TaxiDroidEJB**: Classe do tipo EJB *Stateless* que é responsável por centralizar as operações da aplicação sobre as entidades de Corrida, Usuario e Taxista. Trabalha como um *Facade* para as operações nas entidades.
- **TimerServiceEJB**: Classe do tipo EJB *Timer* que é responsável por executar duas tarefas agendadas:

Método	Descrição
DespacharCorridas	Responsável por despachar corridas que estão aguardando atendimento.
DisconnectTaxi	Responsável por desconectar no sistema Taxistas que estão a mais de X minutos sem comunicação.

3.5.1.4 Diagrama de Estados

O diagrama de estados têm por objetivo demonstrar as transições de estados possíveis entre as entidades de um sistemas. A seguir serão demonstrados os diagramas de estados das entidades Atendimento e Taxista.

3.5.1.4.1 Estados Atendimento

A entidade Corrida no sistema representa uma solicitação feita pelo usuário. Ela possui um ciclo de vida bem definido. Como exemplos do ciclo de vida, podemos ter os seguintes cenários:

1) Cenário de encerramento:

1. Cliente solicita atendimento(AGUARDANDO ATENDIMENTO).
2. Sistema despacha a solicitação para o taxista A(status DESPACHADO).
3. Taxista A recusa o atendimento, pois está encerrando o expediente(status AGUARDANDO ATENDIMENTO).
4. Sistema despacha a solicitação para o taxista B(status DESPACHADO).
5. Taxista B aceita o atendimento(status EM ATENDIMENTO).
6. Taxista B atende o cliente e encerra a solicitação(status ENCERRADO).

2) Cenário de cancelamento:

1. Cliente solicita atendimento(AGUARDANDO ATENDIMENTO).

2. Sistema demora para despachar, pois não existe táxi disponível.
3. Cliente cancela a solicitação, pois vai utilizar outra forma de locomoção(status CANCELADO).

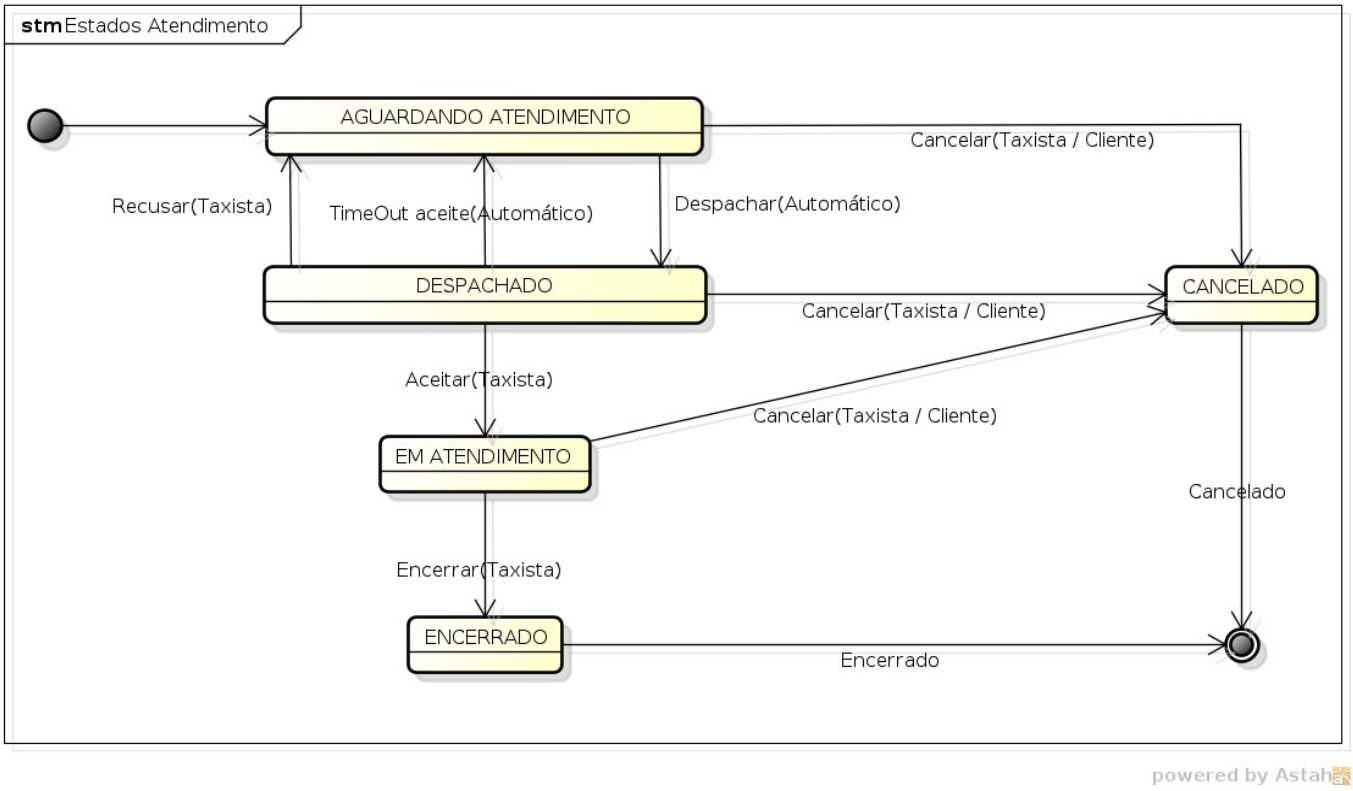

Figura 10: Estados de um Atendimento.

3.5.1.4.2 Estados Táxi

A entidade Taxista possui um status que representa a situação dele no momento. Quando o taxista realiza login no sistema, o status inicial é LIVRE. Quando uma solicitação é despachada para o mesmo, o status é atualizado para OCUPADO. Quando o taxista encerra o expediente ou quando perde comunicação com a central(ex: sem rede), o status é atualizado para DESCONECTADO.

Quando o taxista está com status LIVRE, ele pode receber um despacho de atendimento.

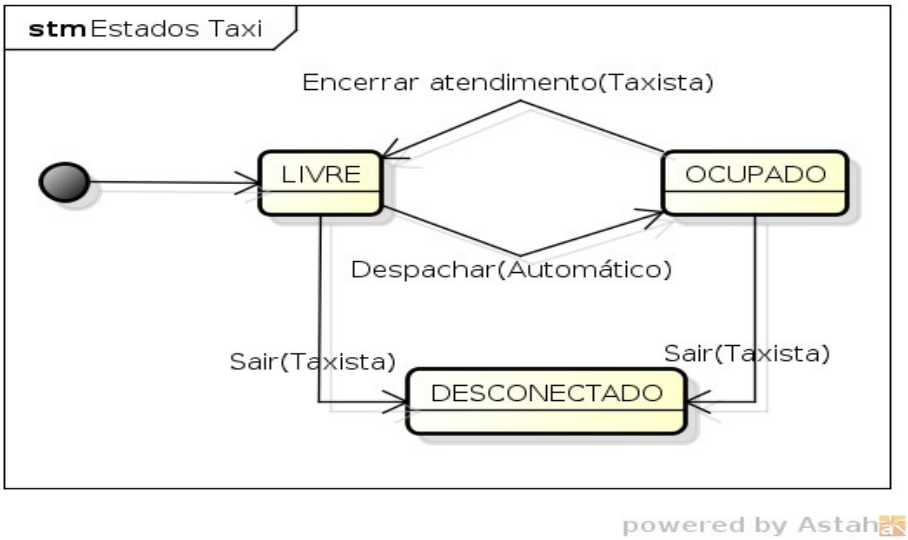

Figura 11: Estados de um Taxista.

3.5.2 Aplicativo Móvel

3.5.2.1 Android Library

Para o desenvolvimento do aplicativo móvel foi utilizado o *Android SDK*. Juntamente com ele vem um conjunto de bibliotecas desenvolvidas que provém serviços aos desenvolvedores *Android*. Para o desenvolvimento do aplicativo móvel, foram utilizadas diversas classes do *Android* para prover serviços como localização e paralelismo de tarefas.

3.5.2.1.1 AsyncTask

Em aplicativos que possuem interação com usuário, tarefas que consomem muito tempo para executarem podem tornar a aplicação pouco responsiva em alguns momentos. Isso se quando essas tarefas rodam dentro da mesma *Thread* onde a UI está executando. Para tratar esse problema, o Android possui a classe *AsyncTask*. Essa classe deve ser utilizada sempre que tivermos alguma tarefa que pode demorar para executar, como por exemplo uma tarefa que utiliza recursos de rede, disco etc. Uma AsyncTask deve prover a implementação de no mínimo 2 métodos:

- protected T doInBackground(T... params): contém o código para realizar a tarefa desejada.
- protected void onPostExecute(T result): invocado ao final da execução do método doInBackground e pode ter um resultado como entrada.

T pode ser uma classe de qualquer tipo e esses tipos são fornecidos na criação da *AsyncTask*.

3.5.2.1.2 LocationManager

Em aplicativos que trabalham com informações de posição geográfica, é necessário utilizar algum dispositivo como GPS ou a rede para obter as coordenadas. No *Android*, A classe *LocationManager* oferece serviços de localização geográfica obtidos através do GPS e rede. Para utilizá-la é necessário se registrar para receber as atualizações de localização. Geralmente isso é feito no método *onResume* e deve ser informado o tipo de serviço de localização a ser usado: GPS_PROVIDER(GPS) ou NETWORK_PROVIDER(Rede WiFi).

3.5.2.1.3 Geocoder

O processo de *geocoding* e *geocoding* reverso consiste respectivamente em transformar coordenadas em um endereço e um endereço em coordenadas. No *Android*, a classe *Geocoder* oferece esse tipo de serviço. A classe oferece alguns métodos, entre eles os 2 mais importantes:

- *getFromLocation*: busca uma lista de endereços com base nas coordenadas informadas.
- *getFromLocationName*: busca uma lista de endereços com base em um nome de local(rua, cidade, bairro etc).

3.5.2.1.4 MapActivity

Em aplicativos que trabalham com informações de posição geográfica, é necessário mostrar mapas com as informações demarcadas. Para isso, o *Android*

oferece a classe *MapActivity*, uma *Activity* especial para trabalhar com mapas em um aplicativo Android.

3.5.2.1.4.1 MapView

Componente gráfico que representa uma visualização de mapa. Contém alguns métodos para visualização de mapas como por exemplo mostrar controles de zoom, tipo de visão(rua ou satélite) etc.

3.5.2.1.4.2 MapController

Classe que provê controles sobre o *MapView*. Contém alguns métodos para controle do zoom do mapa, mostrar determinado ponto no mapa etc.

3.5.2.1.4.3 GeoPoint

Classe que representa um ponto geográfico. Possui informações de latitude e longitude.

3.5.2.1.5 TimerTask

Classe que representa a execução de uma tarefa periodicamente. Implementa a interface *Runnable* e representa uma *Thread* que é executada de tempos em tempos. Possui o método *public void run()* que contém o código a ser executado na *Task*.

3.5.2.1.6 Timer

Em alguns aplicativos é necessário a execução de um determinado método/processo periodicamente. Para isso, o Android oferece a classe *Timer* que permite a execução recorrente de uma *TimerTask*. Possui vários métodos para agendamento de tarefas recorrentes ou de execução única.

- void cancel(): cancela o *Timer* e a execução de todas as tarefas agendadas.
- void schedule(TimerTask task, Date when, long period): agenda uma tarefa

para execução recorrente, após a data informada e com intervalo de execução informado.

- void schedule(TimerTask task, long delay, long period): agenda uma tarefa para execução recorrente, após o tempo informado(em ms) e com intervalo de execução informado(em ms).
- void schedule(TimerTask task, Date when): agenda uma tarefa para execução única na data informada.
- void schedule(TimerTask task, long delay): agenda uma tarefa para execução recorrente, após o tempo informado(em ms).

3.5.2.1 Aplicativo Usuário

3.5.2.1.1 Diagrama de Atividades

O diagrama de atividades têm por objetivo demonstrar um fluxo de atividades dentro de um sistema. A figura 13 representa uma diagrama com as atividades que o usuário pode realizar no aplicativo móvel, desde o login até finalizar o aplicativo.

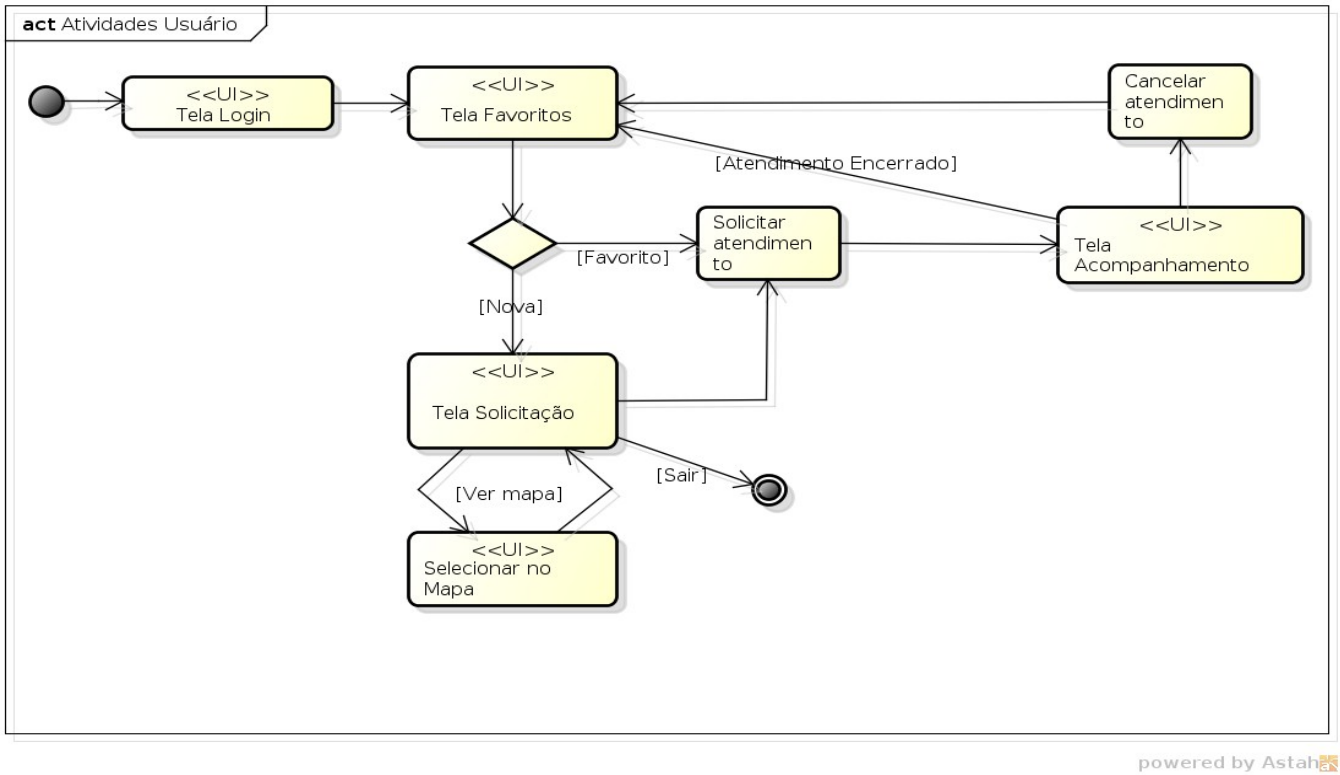

Figura 12: Atividades do Usuário no Aplicativo

3.5.2.1.2 Diagrama de Classes

O diagrama de classes têm por objetivo demonstrar as classes e as relações entre as classes de uma aplicação orientada a objetos(OO). A figura 14 mostra o diagrama de classes do aplicativo móvel para usuários. As classes estão divididas em três pacotes:

- **Usuário**: classes desenvolvidas para o aplicativo móvel do Usuário.
- **Compartilhada**: classes desenvolvidas e que são utilizadas no aplicativo móvel do Usuário e do Taxista.

- **Android**: classes que fazem parte da biblioteca Android.

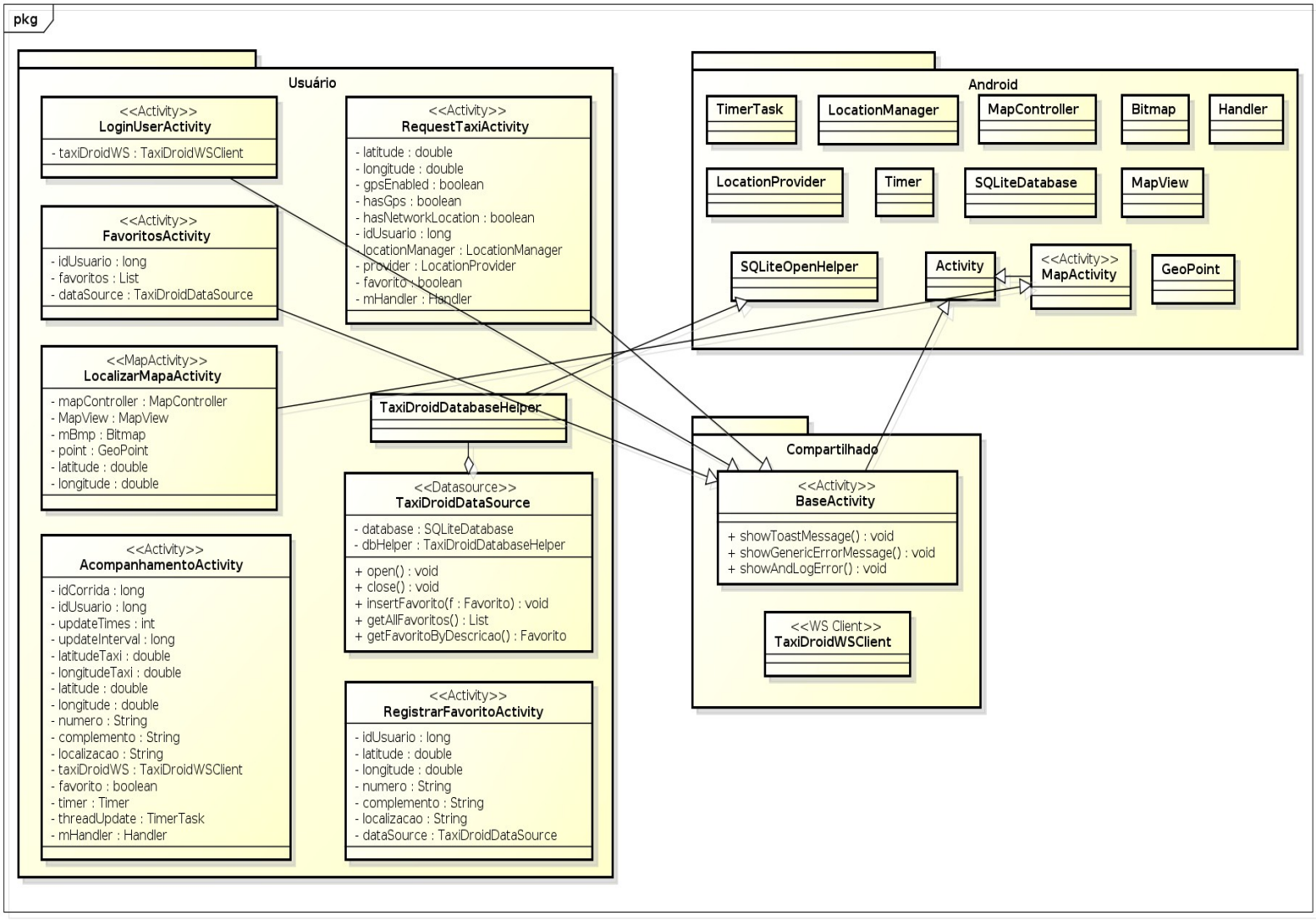

Figura 13: Diagrama de Classes – Aplicativo móvel do Usuário

3.5.2.1.3 Login

A *Activity* de login é a primeira tela do aplicativo. Nela o usuário entra com as informações de usuário e senha e a autenticação é feita no servidor. A figura 15 mostra a tela de login do usuário.

Figura 14: Tela de login.

A chamada remota ao método de login no servidor é feita através da chamada do método *WebService* login, utilizando a biblioteca *Ksoap*. Na resposta do *WebService*, temos um objeto genérico *SoapObject* que contém os atributos de retorno. Para o caso do método de login, o retorno contém a estrutura:

codigoRetorno	Campo do tipo *Integer* que representa o resultado da operação. Valores possíveis: • 1: Sucesso • -1: Erro
mensagemRetorno	Campo do tipo *String* que informa uma mensagem com o resultado da operação(sucesso ou erro). Ex: "Login realizado com sucesso".
IdUsuario	Campo do tipo *Long* que retorna o ID do Usuário no servidor, caso o login aconteça com sucesso.

Caso o login aconteça com sucesso, o usuário é redirecionado para a tela de Favoritos.

3.5.2.1.4 Favoritos

A *Activity* de Favoritos lista para o usuário os locais salvos na base de dados de favoritos. O usuário pode realizar duas ações nessa tela: selecionar um

favorito ou fazer uma nova solicitação. A figura 16 mostra a tela de favoritos.

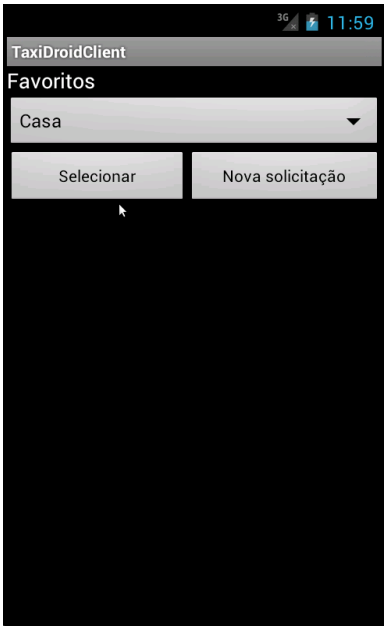

Figura 15: Tela de Favoritos.

Na inicialização da *Activity*, é utilizada a classe *TaxiDroidDataSource* para buscar os favoritos armazenados no celular e popular a listagem dos favoritos. Caso não exista nenhum favorito salvo, o botão “Selecionar” fica desativado. Qualquer ação tomada irá redirecionar o usuário para a *Activity* de Solicitação. Porém, caso seja selecionado um favorito, as informações do favorito localização, latitude, longitude, número e complemento são enviadas para a *Activity* de Solicitação e a requisição já é realizada automaticamente.

3.5.2.1.5 Solicitação de Atendimento

A *Activity* de Solicitação de Atendimento é responsável por capturar do usuário as informações do atendimento e fazer a solicitação no servidor. Caso na tela anterior tenha sido selecionado um favorito, o atendimento é solicitado automaticamente com os dados do Favorito. Caso contrário, os dados são obtidos do *smartphone* e da entrada do usuário. A figura 17 mostra a tela de Solicitação de Atendimento.

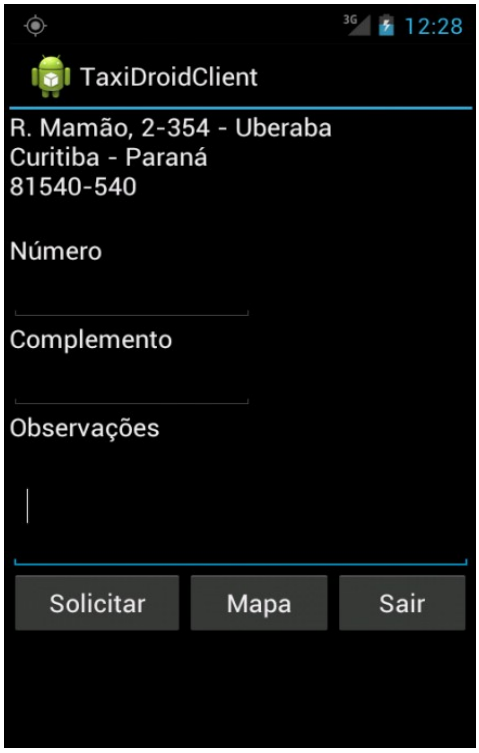

Figura 16: Tela de Solicitação de Atendimento.

A localização inicial do usuário é obtida através da classe *LocationManager* do Android. Para receber atualizações da localização, é necessário que a *Activity* implementar a interface *LocationListener* do Android e se registrar para obter atualizações na localização. A interface *LocationListener* provê o método *onLocationChanged* que é invocado de forma implícita pelo Android sempre que ocorre uma atualização na localização do dispositivo. Quando isso ocorre, as informações de latitude e longitude são atualizadas e a localização é atualizada na tela. Isso é feito utilizando a classe *Geocoder,* que realiza uma busca do endereço utilizando como parâmetros a latitude e longitude e atualiza a informação de localização com a resposta da busca.

Após a localização ser obtida do *GPS* ou rede, o botão “Solicitar” fica disponível. Porém, é necessário informar o número e se necessário o complemento. O número é necessário pois a *API* de localização não consegue identificar exatamente o número e sim um trecho de números. Se por um acaso a localização provida automaticamente não seja a localização exata, é possível

corrigir a localização selecionando no mapa. Isso pode ser feito pelo botão “Mapa”.

A chamada remota ao método de solicitar atendimento no servidor é feita através da chamada do método *WebService* solicitarCorrida, utilizando a biblioteca *Ksoap*.

No resultado da chamada, temos um objeto genérico *SoapObject* que contém os atributos de retorno do serviço. Para o caso do método de login, o retorno contém a estrutura:

codigoRetorno	Campo do tipo *Integer* que representa o resultado da operação. Valores possíveis: • 1: Sucesso • -1: Erro
mensagemRetorno	Campo do tipo *String* que informa uma mensagem com o resultado da operação(sucesso ou erro). Ex: "Solicitação processada com sucesso. Aguarde o atendimento".
idCorrida	Campo do tipo *Long* que retorna o ID da Solicitação de Atendimento no servidor, caso o a solicitação aconteça com sucesso.

3.5.2.1.5 Selecionar Localização no Mapa

A *Activity* de Selecionar Localização no Mapa permite ao usuário selecionar no mapa qual a sua localização. Para tal, é mostrado ao usuário um mapa e centralizado com a posição obtida anteriormente pelo *GPS* ou rede. A figura 18 mostra a tela de Selecionar Localização no Mapa.

Figura 17: Tela de Selecionar Localização no Mapa

Após escolher a localização e apertar o botão “Selecionar”, o usuário é redirecionado para a tela de Solicitação de Atendimento com a localização atualizada e pode seguir com a solicitação.

3.5.2.1.6 Acompanhamento da Solicitação de Atendimento

A *Activity* de Acompanhamento é responsável por mostrar o andamento da solicitação de atendimento ao usuário. As figuras 19 e 20 mostram a tela de acompanhamento inicial e após o despacho de um taxista.

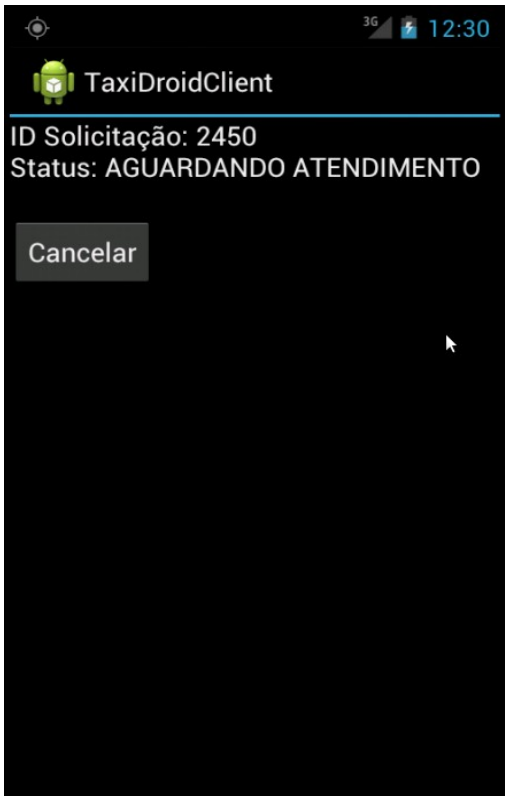

Figura 18: Aguardando atendimento

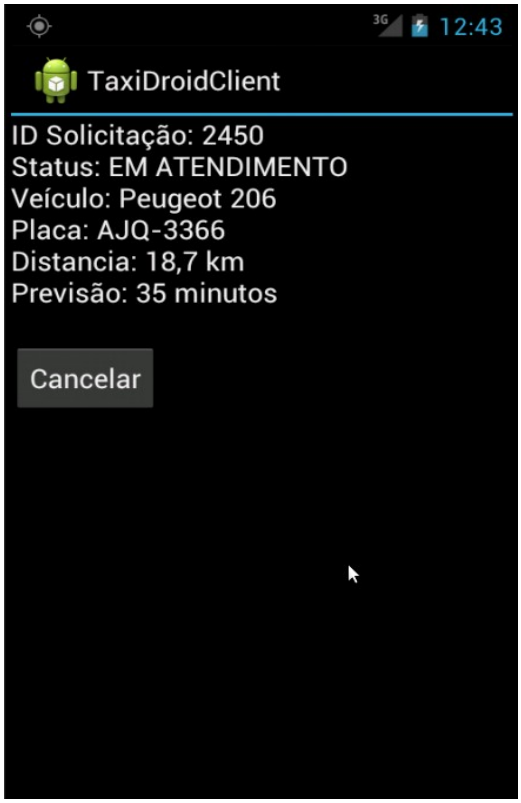

Figura 19: Em atendimento

Para a busca por atualizações da solicitação, é utilizando o mecanismo de pooling, que consiste no aplicativo ir até o servidor buscar atualizações para o atendimento. A cada X segundos(tempo padrão é 10 segundos) uma *TimerTask* é executada(agendada pelo *Timer*) e vai no servidor buscar atualizações para a solicitação de atendimento. A chamada remota ao método de buscar atualizações do atendimento no servidor é feita através da chamada do método *WebService* requestUpdatesCorrida, utilizando a biblioteca *KSoap*.

No resultado da chamada do *WebService* temos um objeto genérico *SoapObject* que contém os atributos da resposta do serviço. Para o caso do método de login, o retorno contém a estrutura:

codigoRetorno	Campo do tipo *Integer*. Valores possíveis: • 1: Sucesso • -1: Erro • 3: Sem atualizações • 4: Atendimento encerrado

	• 5: Atendimento cancelado
mensagemRetorno	Campo do tipo *String* que informa uma mensagem com o resultado da operação(sucesso ou erro). Ex: "Solicitação por atualizações realizada com sucesso.".
informacoes	Campo do tipo *String* que retorna as informações da solicitação de atendimento. Ex: ID Solicitação: 1 Status: EM ATENDIMENTO Veículo: Peugeot 206 Placa: AJQ-3366 Distância: 1,3 km Previsão: 5 minutos

Caso o retorno seja 3(Sem atualizações), nada é feito. Caso o retorno seja 1(Sucesso), as atualizações do atendimento são atualizadas para o cliente. Caso o retorno seja 4(Atendimento encerrado), é informado ao cliente que o atendimento foi encerrado e é oferecido a opção de salvar o local nos favoritos conforme a primeira figura. Caso o retorno seja 5(Atendimento cancelado), é informado ao cliente que o atendimento foi cancelado conforme a segunda figura.

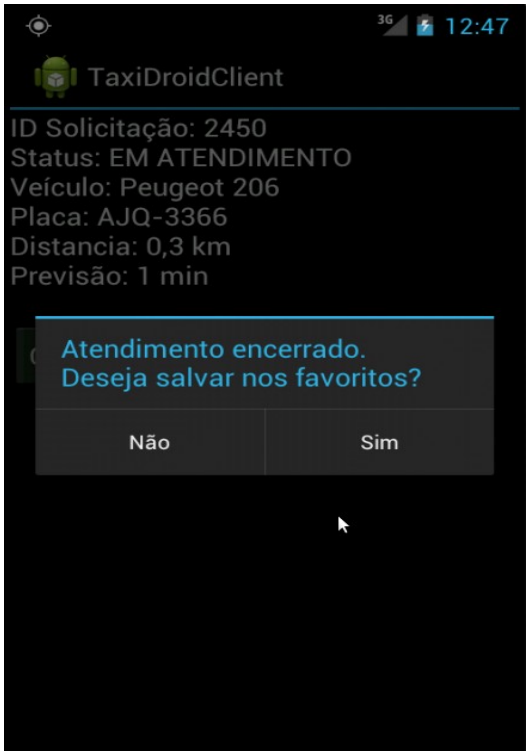

Figura 20: Atendimento encerrado

Figura 21: Atendimento cancelado

Quando o atendimento é encerrado, o usuário pode ser redirecionado para a tela de Salvar Favorito ou para a tela inicial do aplicativo dependendo da ação executada.

3.5.2.1.7 Registrar Favorito

A *Activity* de Registrar Favorito é responsável por obter do usuário o nome do favorito e armazenar em banco de dados *SQLite* no dispositivo. Para tal, foi criada uma tabela FAVORITOS que é utilizada par armazenas os favoritos de cada usuário. A tabela possui a seguinte estrutura:

COLUNA	TIPO
DESCRICAO	TEXT
LATITUDE	REAL
LONGITUDE	REAL
NUMERO	TEXT
COMPLEMENTO	TEXT
LOCALIZACAO	TEXT

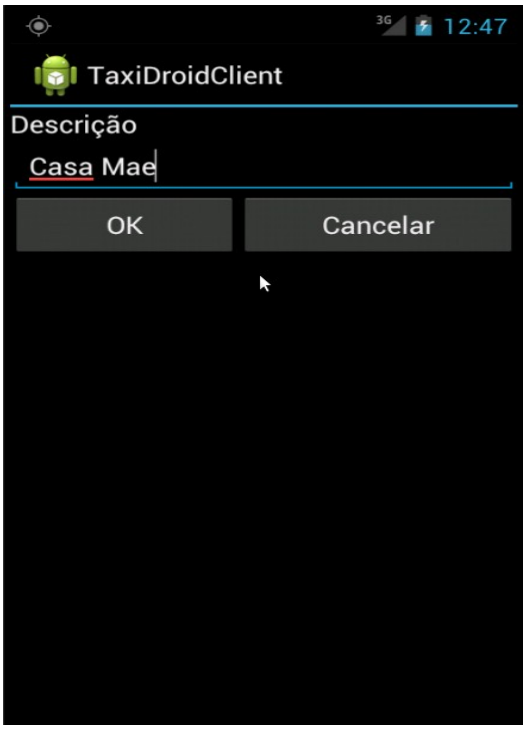

Figura 22: Tela de Registrar Favorito

Para lidar com a persistência no *SQLite*, foram criadas duas classes. A classe *TaxiDroidDatabaseHelper* herda da classe *SQLiteOpenHelper* do Android e é responsável por criar a base de dados no Android e atualizar quando necessário. Também é responsável por retornar instâncias do tipo *SQLiteDatabase*, classe responsável pelas operações na base de dados *SQLite*. Na classe *TaxiDroidDatabaseHelper* foram necessários a sobrescrita do construtor e de outros dois métodos. A seguir trecho de código da classe TaxiDroidDatabaseHelper.

```
public TaxiDroidDatabaseHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
}
@Override
public void onCreate(SQLiteDatabase database) {
        database.execSQL(DATABASE_CREATE);
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_FAVORITOS);
        onCreate(db);
}
```

O construtor recebe o nome da base de dados e da versão. Se a base de dados não existe, o método *onCreate* será usado para a criação. Caso ela exista mas em uma versão inferior a usada no construtor, o método *onUpgrade* é usado e irá destruir a base de dados existente e criar uma nova com as alterações na estrutura.

A classe *TaxiDroidDatasource* é uma espécie de implementação do *DAO(Data Access Object)* que provê os métodos para lidar com a persistência. Ela possui como atributo um objeto *SQLiteDatabase* e *TaxiDroidDatabaseHelper*. Na classe *SQLiteDatabase*, os métodos *open()* e *close()* são utilizados para abrir transações com a base de dados e fechar.

3.5.2.2 Aplicativo Taxista

3.5.2.2.1 Diagrama de Atividades

A figura 24 representa uma diagrama com as atividades que o usuário pode realizar no aplicativo móvel, desde o login até finalizar o aplicativo.

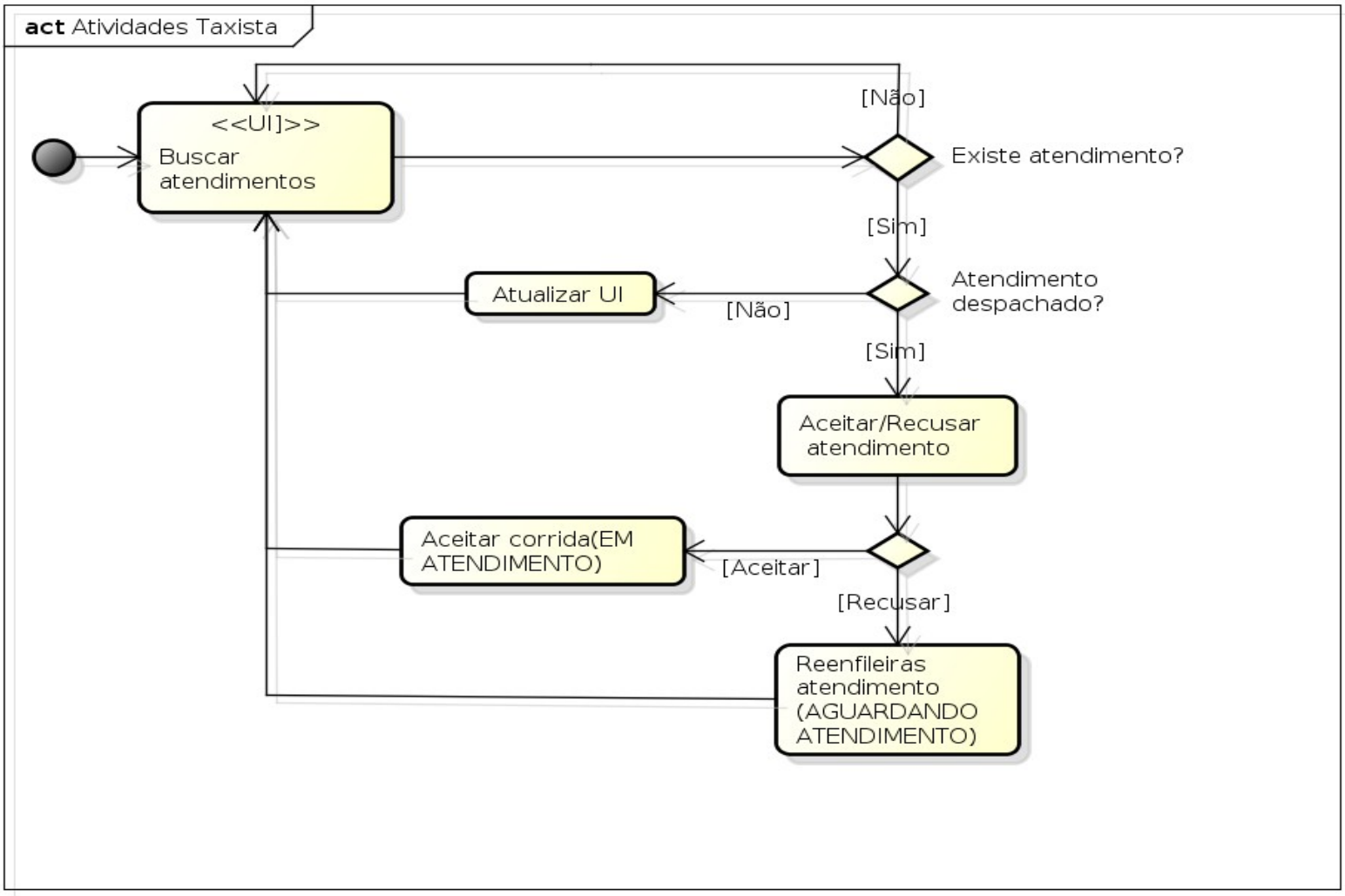

Figura 23: Diagrama de atividades do Taxista

3.5.2.2.2 Diagrama de Classes

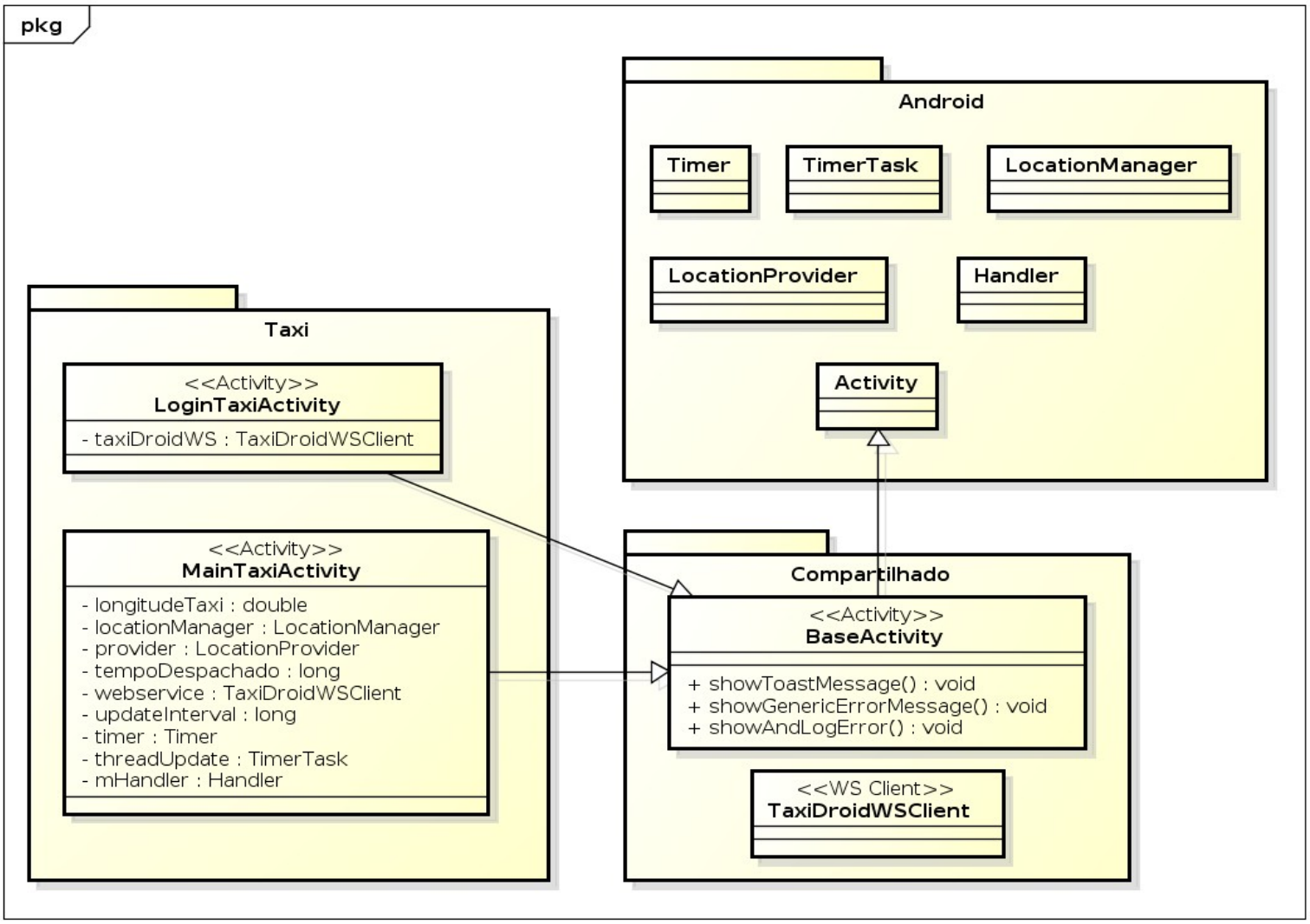

Figura 24: Diagrama de Classes – Aplicativo móvel do Usuário

3.5.2.2.3 Login

A *Activity* de login é a primeira tela do aplicativo. Nela o taxista entra com as informações de usuário e senha e a autenticação é feita no servidor. É semelhante à *Activity* de realizar login do aplicativo do Usuário. A diferença está apenas no método *WebService* invocado. Enquanto no aplicativo do Usuário é utilizado o método *loginUsuario*, no aplicativo do Taxista é utilizado o método *loginTaxi*.

3.5.2.2.4 Acompanhamento de solicitações

Além da *Activity* de login, o aplicativo do Taxista possui somente uma tela. Nessa tela ele recebe notificações sobre atendimentos a serem realizados.

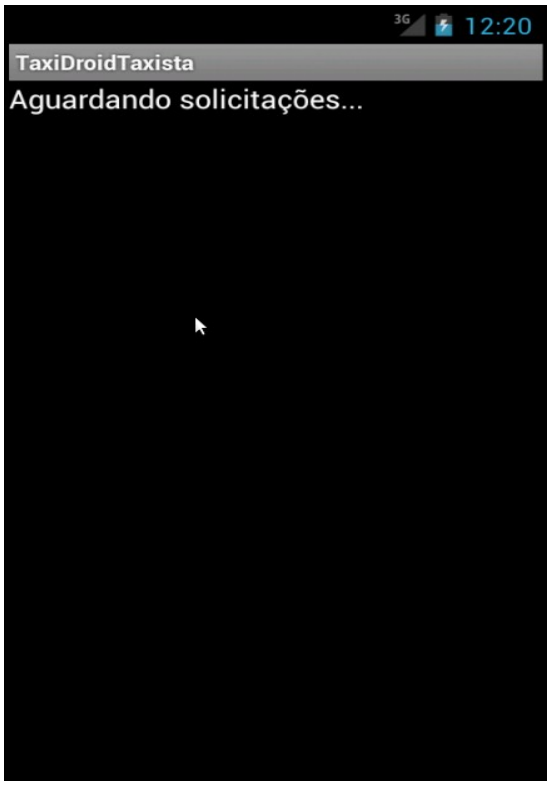

Figura 25: Aguardando solicitações

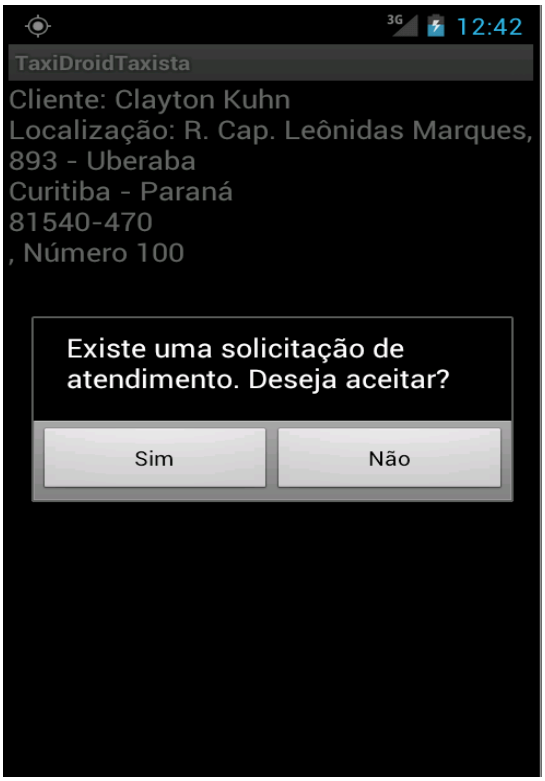

Figura 26: Despacho de uma solicitação

A primeira figura mostra a tela inicial de acompanhamento de solicitações. Quando uma solicitação é despachada ao taxista pelo servidor, a tela da segunda figura é mostrada, solicitando se o taxista aceita o atendimento ou não. Nesse caso, temos três fluxos que podem ser seguidos:

1. Taxista aceita o atendimento. Nesse caso, o status do atendimento é atualizado no servidor para EM ATENDIMENTO na entidade Corrida.
2. Taxista rejeita o atendimento. Nesse caso, o status do atendimento é atualizado no servidor para AGUARDANDO ATENDIMENTO na entidade Corrida. Com isso, o atendimento volta para a fila de despacho.
3. Time-out. Se o taxista não aceitar o atendimento em um minuto,

automaticamente o atendimento volta para a fila de despacho.

A chamada remota ao método de atualizar atendimento no servidor é feita através da chamada do método *WebService* atualizarStatusCorrida, utilizando a biblioteca *Ksoap*. Quando o taxista aceita o atendimento, a tela da figura 28 é mostrada.

Figura 27: Solicitação em atendimento

Ao finalizar o atendimento, o método *WebService* encerrarAtendimento é invocado no servidor utilizando a biblioteca *Ksoap*. Ao cancelar o atendimento, o método *WebService* cancelarSolicitacao é invocado no servidor utilizando a biblioteca *Ksoap*.

O botão “Navegar” permite utilizar o aplicativo de GPS do dispositivo para realizar a navegação até o cliente. Para tal, é utilizada uma *Intent* do tipo ACTION_VIEW e passado como parâmetro dados informando a utilização do *Google Navigation*. A seguir trecho de código usado para iniciar a navegação

usando o aplicativo GPS.

```
Intent intent = new Intent(android.content.Intent.ACTION_VIEW,
Uri.parse(String.format("google.navigation:q=%f,%f", latitude, longitude)));
startActivity(intent);
```

No Intent são passadas as informações de latitude e longitude do cliente. Após isso, o *Google Navigation* é iniciado com a rota traçada. A figura 29 mostra a tela de navegação.

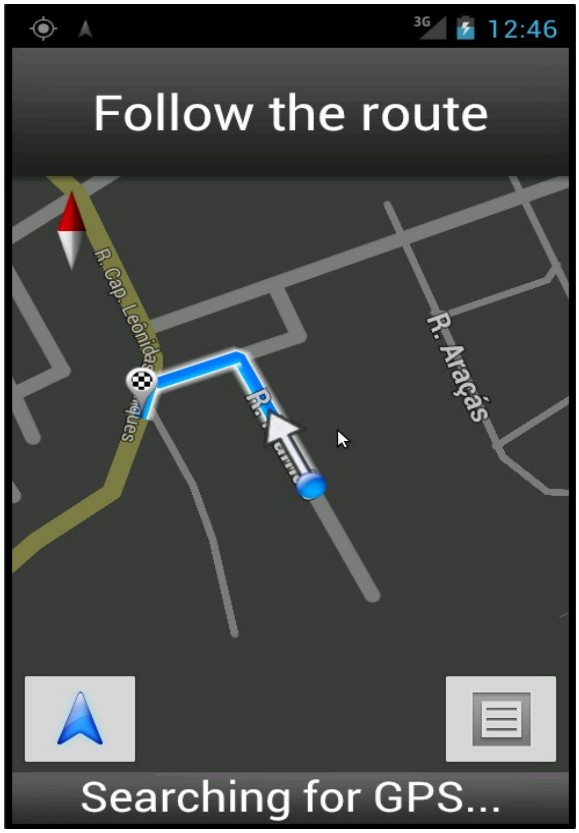

Figura 28: Navegação GPS até o cliente

4. Conclusões e Trabalhos Futuros

O Brasil ainda se encontra abaixo de outros países em questão de tecnologia e telecomunicações. Com isso, muitos serviços que poderiam ser melhorados com as tecnologias disponíveis continuam defasados. O potencial existente no uso de dispositivos móveis para melhorar serviços como transportes é muito grande.

O serviço de táxi possui muitos problemas atualmente como a alta demanda, baixa automatização das centrais etc. A partir dessa motivação, esse trabalho apresentou uma abordagem para melhora deste serviço utilizando alta conectividade através do uso de *smartphones*, propondo um protótipo desenvolvido utilizando a plataforma *Android* juntamente com *Java*. Esse protótipo busca a automatização e otimização do tempo necessário para solicitar um atendimento e o despacho do táxi até o cliente.

A escolha da plataforma *Android* foi motivada pelo fato de ser um projeto de código aberto e que está ganhando cada vez mais força no mercado, através da popularização dos smartphones e tablets. Também pela excelente qualidade da plataforma *Android*, tanto nas ferramentas disponíveis (IDE, emulador, *debugger*) que auxiliam bastante no desenvolvimento quanto nas documentações e *frameworks* disponíveis para auxiliar no desenvolvimento de aplicações.

O protótipo desenvolvido nesse trabalho cumpriu o objetivo proposto. Foi feita a automatização do despacho de táxis, sendo otimizada para buscar um táxi mais próximo do cliente e a automatização na solicitação de atendimento através do *smartphone*.

No trabalho desenvolvido, a plataforma *Android* foi bastante explorada. Diversas decisões tiveram que ser tomadas para o desenvolvimento do protótipo. Também foi necessário fechar um escopo para o protótipo, limitando inicialmente suas funcionalidades.

Com o desenvolvimento do trabalho, espera-se contribuir com as empresas que oferecem o serviço de táxi para que sejam automatizadas e utilizem a tecnologia existente hoje. A plataforma *Android* e outras plataformas móveis podem ser a porta de entrada para as empresas entrarem no mundo da conectividade móvel. Esse mundo têm muito a oferecer para as empresas e podem ajudar a otimizar os serviços e reduzir custos.

Como o trabalho desenvolvido obteve-se um protótipo com escopo fechado, sendo possível a inclusão de muitas melhorias e implementações no aplicativo, dentre as quais as principais seriam:

- Uso de *REST* ao invés de *SOAP*
    - O *REST(Representational State Transfer)* é uma abordagem proposta para a utilização de *WebServices* utilizando o protocolo *HTTP*. É extremamente leve, apresentando vantagens sobre a abordagem SOAP. Para o trabalho desenvolvido, o *REST* reduziria a carga de transferência de dados na comunicação com o servidor, reduzindo o uso de banda e otimizando o aplicativo.
- Pesquisa por endereços através de nome
    - Permitir ao usuário buscar o local onde deseja solicitar atendimento através do nome da rua etc.
- Utilização de uma plataforma de Mapas *open-source*
    - Utilizar uma plataforma de mapas que não seja proprietária, reduzindo custos de manter o aplicativo.
- Armazenar histórico de solicitações
    - Na tela inicial pós login, listar os 5 últimos atendimentos solicitados para facilitar e agilizar um novo atendimento.
- Encerramento automático do atendimento
    - Detectar quando o local de origem foi atingido. Para tal, é necessário

registrar no atendimento o local para o qual o cliente deseja ir. Com isso, por proximidade do local é possível encerrar automaticamente o atendimento.

- Agendamento de solicitação
    - Permitir ao usuário agendar um táxi para um dia e horário desejado.

5. Referências Bibliográficas

ABLESON W. Frank; COLLINS, Charlie; SEN, Robi. Android em Ação. Rio de Janeiro: Elsevier, 2012. 3. ed.

CANALLE, Anderson Luis. Empregando tecnologia Java, Android e Geoprocessamento em aplicativos móveis. Monografia de Especialização. UTFPR(Universidade Tecnológica Federal do Paraná), 2011.

FIELDING, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Tese de doutorado, Universidade da Califórnia, 2000.

GAERTNER. Smartphone Operating System Market by Year-End 2012. Disponível em: http://www.gartner.com/it/page.jsp?id=1622614 Acesso em: Maio/2012

GONÇALVES, Júlio César. USO DA PLATAFORMA ANDROID EM UM PROTOTIPO DE APLICATIVO COLETOR DE CONSUMO DE GAS NATURAL. Monografia de Especialização, UTFPR (Universidade Tecnológica Federal do Paraná), 2011.

GOOGLE. Installing the SDK. Disponível em: http://developer.android.com/sdk/installing/index.html Acesso em: Março/2012

GOOGLE. What is Android. Disponível em: http://developer.android.com/guide/basics/what-is-android.html Acesso em: Março/2012

GOOGLE. Activities. Disponível em: http://developer.android.com/guide/components/activities.html Acesso em: Março/2012

KSOAP, Biblioteca para comunicação com WebServices em Android. Disponível em: http://code.google.com/p/ksoap2-android/ Acesso em: Abril/2012

LECHETA, R. R. Google Android: Aprenda a criar aplicações para dispositivos móveis com o Android SDK. Novatec, 2a edição, 2010.

MARINHO, Simone. Aplicativos para celulares ajudam a chamar táxi. Disponível em: http://oglobo.globo.com/megazine/aplicativos-para-celulares-ajudam-chamar-taxi-5058221 Acesso em: Maio/2012

MURARO, Herbert Muniz. SISTEMA DE CHAMADOS DE TAXI PARA SMARTPHONES. Monografia de graduação do curso de Engenharia da Computação, Universidade Positivo, 2011.

PIEROBOM, Jean Lima. OTIMIZAÇÃO POR NUVEM DE PARTÍCULAS APLICADA AO PROBLEMA DE ATRIBUIÇÃO DE TAREFAS DINÂMICO. Dissertação de Mestrado Profissional em Computação Aplicada. UTFPR(Universidade Tecnológica Federal do Paraná), 2012.

SHEVAR, Amir. Are SOA and web-services synonymous? 2006. Disponível em: http://www.oreillynet.com/onjava/blog/2006/01/are_soa_and_webservices_synony.html Acesso em: Julho/2012

SOAP, Protocolo de Comunicação utilizado em WebServices. Disponível em: http://www.w3schools.com/soap/default.asp Acesso em: Maio/2012

SQLITE. About SQLite. Disponível em: http://www.sqlite.org/about.html , Acesso em: Maio/2012

TOSIN, Carlos. Conhecendo o Android. Disponível em: http://www.softblue.com.br/blog/home/postid/11/CONHECENDO+O+ANDROID Acesso em: Abril/2012