UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Departamento de Sistemas de Computação

SmartPendant – Um periférico para smartphones Android

Davi Diório Mendes

São Carlos - SP

SmartPendant – Um periférico para smartphones Android

Davi Diório Mendes

Orientador: Fernando Santos Osório

Monografia referente ao projeto de conclusão de curso dentro do escopo da disciplina SSC0670 – Projeto de Formatura I do Departamento de Sistemas de Computação do Instituto de Ciências Matemáticas e de Computação – ICMC-USP para obtenção do título de Engenheiro de Computação.

Área de Concentração: Sistemas Embarcados, Dispositivos Móveis

USP – São Carlos
05 de Novembro de 2015

"Em verdade vos digo que tudo o que ligardes na terra será ligado no céu, e tudo o que desligardes na terra será desligado no céu. "

(Mateus 18:18)

Dedicatória

A Deus, que me capacitou para chegar até aqui. Mesmo com todas as dificuldades do caminho e falhas presentes em nossas vidas, ainda assim, Ele com sua graça nos leva onde nunca pensamos chegar, nos dá o que nunca pensamos ter e nos ama de uma maneira que nunca pensamos ser possível.

Agradecimentos

A Deus por me trazer até aqui.

Aos meu pais por me ensinarem a importância do estudo neste mundo.

Aos professores Klauber Marcelli e Nilton Barrozo pelo incentivo e oportunidade que me deram.

A minha noiva Camila Peres que sempre me ouviu quando precisava conversar com alguém.

A esta universidade e seu corpo docente, que me proporcionou o conhecimento necessário para alcançar meus sonhos.

Ao meu orientador Prof. Dr. Fernando Osório pelo incentivo e suporte no desenvolvimento deste projeto.

Resumo

O *SmartPendant* é uma aplicação de sistemas emboarcados, envolvendo dispositivos móveis, computação distribuída e ubíqua, visando trazer maior comodidade na utilização de um *smartphone Android*. O sistema é composto de um pingente, que é um sistema embarcado implementado utilizando *Arduino*, uma plataforma de prototipagem eletrônica. O pingente é um dispositivo que se comunica com um software instalado no dispositivo móvel, no caso um *smartphone Android*. Ele vibra e pisca mediante notificações que cheguem ao celular, além de possuir quatro botões com ações personalizáveis frente a diversos contextos do *Android* como: em espera, tocando música, recebendo ligação. Este documento aborda sobre uma arquitetura de *software* que dê suporte para estas funcionalidades.

Sumário

Lista de Abreviaturas

I2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
LED	Light Emitting Diode
RF	Rádio Frequência
SDK	Software Development Kit
SPI	Serial Peripheral Interface
UML	Unified Modeling Language

Lista de Figuras

CAPÍTULO 1: INTRODUÇÃO

Este trabalho irá abordar o desenvolvimento de uma aplicação para gerenciar o uso de um periférico em *smartphones Android*. O periférico utilizado é o *SmartPendant*, um protótipo de pingente inteligente capaz de se comunicar com um dispositivo *Android.* A seguir serão apresentadas a contextualização e motivação deste trabalho, seguido dos principais objetivos propostos.

1.1. Contextualização e Motivação

O *SmartPendant* é um dispositivo vestível que agrega funcionalidades a um smartphone *Android*. Atualmente a indústria busca cada vez mais criar aplicações que envolvam dispositivos vestíveis (KIRK, 2014). Normalmente estes dispositivos possuem um cunho atlético ou de monitoramento físico (MORRIS, 2008; LAPINSKI, 2011).

Os *smartwatches* (relógios inteligentes) e as pulseiras inteligentes são os acessórios mais comuns de se encontrar no mercado. Normalmente eles possuem um cunho esportivo, ou de monitoramento físico, fazendo medições de batimento cardíaco, gravação de rotas tomadas e contagem de passos dados.

Atualmente vem sendo desenvolvido dispositivos vestíveis em forma de pingente. Um grande produto neste segmento é o ARC Pendant. Com segmento esportivo, ele possui um sistema de auxílio de navegação com GPS. Seis motores vibratórios em seu cordão informam ao usuário sobre a direção que deve ser tomada. Uma interface de voz recebe comandos do usuário por meio da fala. Possui monitoramento físico.

A motivação deste projeto é trazer uma nova interface ao smartphone convencional. Com um pingente vibrando em seu peito não há como perder uma ligação, mensagem ou e-mail importante. Sendo inclusive uma aplicação de acessibilidade quando levado em conta idosos e deficientes auditivos que, muitas vezes, perdem ligações por não estarem com o celular por perto. Os comandos contextuais através dos botões trazem discrição e conforto para utilizar o celular em momentos inapropriados, como rejeitar uma ligação durante uma reunião ou controlar o tocador de música no ônibus.

1.2. Objetivos

Este trabalho propõe uma arquitetura escalável para periféricos sem fio com uso voltado para *smartphones* e internet das coisas (IoT). Especificamente no projeto desenvolvido o periférico é um pingente e se conecta a dispositivos *Android*. Possui um caráter tanto de entrada como de saída de dados, fazendo uso de botões, sinal luminoso e de vibração. O periférico é um dispositivo *dummy* (sem conhecimento de aplicação), simplesmente recebe e executa comandos de atuação e envia eventos de entrada. O dispositivo mestre deve decidir o que fazer com os eventos de entrada do pingente e enviar comandos de atuação ao mesmo, conforme o contexto em que o mestre se encontra.

Como objetivo deste projeto, foi definido o desenvolvimento de um protótipo de pingente baseado no *Arduino*, juntamente com uma aplicação *Android*, em uma arquitetura escalável e adaptável para diversas necessidades. Além de um mero *wearable*, o *SmartPendant* possui uma característica de acessibilidade para deficientes auditivos, informando o usuário de notificações e ligações mesmo longe do celular, onde o usuário provavelmente não ouviria o dispositivo tocando. Este projeto deve servir como um ponto de partida para outros projetos, aproveitando a arquitetura base e somente incrementando com novas funcionalidade.

1.3. Organização do Trabalho

No capítulo 2 é apresentada uma revisão da bibliografia utilizada neste trabalho. Os tópicos *Android*, *Arduino*, sistemas embarcados, sistemas distribuídos e padrões de projetos serão abordados de forma a prover um melhor entendimento deste documento.

No capítulo 3 apresentamos o desenvolvimento do projeto. Decisões tomadas, pacotes e classes com suas respectivas funcionalidades, padrões de projetos aplicados e principais algoritmos utilizados.

No capítulo 4 concluímos este documento apresentando as contribuições geradas, as experiências do aluno neste projeto e no curso e propostas de trabalhos futuros.

CAPÍTULO 2: REVISÃO BIBLIOGRÁFICA

2.1. Considerações Iniciais

Neste capítulo serão apresentadas algumas bases para o entendimento deste documento. Inicialmente abordaremos o que é *Android* e o que é *Arduino*, os principais recursos deste projeto. Então veremos sobre sistemas embarcados e sistemas distribuídos, as principais áreas que este projeto envolve. Por fim veremos padrões de projeto, uma grande ferramenta de desenvolvimento visando resolução de problemas e reuso de código.

2.2. Android

O *Android* é um sistema operacional, baseado em Linux, para aparelhos celulares, criado por Andy Rubin. Em 2005, a *Android Inc* foi comprada pela Google. Desde então este SO é mantido pela gigante das buscas (JACKSON, 2011).

O *Android* possui um kit de desenvolvimento de software que permite a criação de aplicações para o SO. Através deste kit é possível trabalhar com criação de interfaces gráficas, processamento paralelo, comunicação interprocessos, uso de base de dados, entre outros. É uma plataforma completa de desenvolvimento, muito semelhante a desenvolver para um computador.

Um programa desenvolvido para *Android* é comumente chamado de aplicativo, ou aplicação. São escritos majoritariamente em Java, embora seja possível escrever módulos em C, ou C++, e incluí-los à aplicação Java. Outra linha de desenvolvimento são os *webviews*, onde se desenvolve o aplicativo como se estivesse desenvolvendo um site, atrelando-o a um *webview* que faz a renderização da aplicação.

2.3. Arduino

Massimo Banzi (2014, p. 17), cofundador do Projeto *Arduino*, nos traz a seguinte definição para sua plataforma:

> O Arduino é uma plataforma de computação física de fonte aberta para a criação de objetos interativos independentes ou em colaboração com softwares de computador. Ele foi projetado para artistas, designers e outros profissionais que queiram incorporar a computação física a seus projetos sem que para isso precisem ter se formado em Engenharia Elétrica.

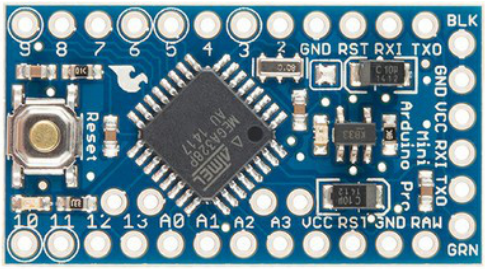
No trecho acima, Banzi utiliza o termo computação física. O que ele mesmo descreve como computação física em seu livro é o projeto de objetos que interagem com os humanos por meio de sensores e atuadores. Estes possuem seu comportamento controlado através de um *software*. Este *software* deve ser executado em um microcontrolador.

Sendo assim, o propósito do *Arduino* é permitir pessoas, que não possuem elevado conhecimento em eletrônica, desenvolver protótipos com alguma interação humana. Para isto basta conhecer como desenvolver para *Arduino*. Uma vez que esta plataforma é voltada para "leigos", seu método de desenvolvimento é bastante simplificado e de fácil aprendizado, embora seja em C++.

Um *software* desenvolvido para *Arduino* é comumente chamado de *sketch*, dado a característica de prototipação do *Arduino*. Além do ferramental disponibilizado pela *Arduino* para o desenvolvimento, ainda existem inúmeras bibliotecas de terceiros que podem ser incluídas no *sketch*.

O *Arduino* possui pinos de entrada e saída, tanto analógicos quanto digitais. Também possui barramentos de comunicação serial, SPI e I2C. Desta forma é possível conectar botões, LEDs, antenas *bluetooth*, *wifi*, módulos RF, motores de corrente contínua, enfim, o que for preciso para o protótipo. Na figura 1 temos Arduino Pro Mini, o *Arduino* utilizado neste projeto.

Figura 1 -- Arduino Pro Mini



Fonte: https://www.arduino.cc/en/uploads/Main/ArduinoProMini_Front_3v3.jpg

2.4 Sistemas Embarcados

Segundo White (2011, p.1) a definição de sistemas embarcados varia de pessoas para pessoas. Para alguém acostumado a lidar com servidores, programação para dispositivos móveis pode ser considerada como um desenvolvimento embarcado. Por outro lado, para aquele acostumado a trabalhar com microcontroladores de 8-bits, qualquer coisa com um sistema operacional não parece muito embarcado. Mas ela enuncia a definição: "um sistema embarcado é um sistema computadorizado construído propositadamente para sua aplicação", ou seja, computadores são de propósito geral, sistemas embarcados tem aplicação bem definida.

Analisando um *smartphone Android*, embora possua a aplicação bem definida de ser um celular, ainda assim possui uma vasta generalização proveniente do *"smart"* em sua categoria de produto. Toda a vasta gama de aplicativos voltados para este sistema operacional muitas vezes tornam um computador completamente dispensável para usuários comuns (que não necessitem de nenhum recurso avançado da computação). Serviços de e-mail, agenda, impressão, redes sociais, são mais que suficientes para a maioria dos usuários. Logo é de difícil análise sobre ser ou não um sistema embarcado. Alguns o consideram como, enquanto outros não.

Por outro lado, o *Arduino* é uma ferramenta para prototipação e desenvolvimento de sistemas embarcados. O *SmartPendant* é um exemplo disto. Ele possui processamento de dados, logo é um sistema computadorizado, mas sua construção é focada na

funcionalidade: quatro botões, antena *bluetooth*, um LED e um motor vibratório. Uma construção bem específica, para um sistema bem específico.

Um sistema embarcado que começa a aparecer são os sistemas ubíquos. Lyytinen (2002) diz que a computação ubíqua é termos computadores embarcados em nossos movimentos e interações naturais com o ambiente, tanto físico quanto social. Computação ubíqua, na prática, é poder de processamento em coisas corriqueiras do nosso dia a dia, extraindo informações do ambiente para que essas coisas se adequem ao ambiente.

2.4. Sistemas Distribuídos

Sistemas distribuídos é algo de difícil definição. Cada autor utiliza uma definição diferente, embora todas remetam que há computadores, ou outro dispositivo com capacidade de processamento, trabalhando em conjunto. Vamos tomar por definição: "Um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens. " (COULOURIS, 2007, p. 15).

Esta definição descreve bem o que se busca como sistema distribuído neste projeto. Embora não lidemos com computadores, tanto o *smartphone Android* como a plataforma *Arduino* são capazes de processar dados como um computador. Embora não estejam ligados em rede, estão ligados por uma comunicação *bluetooth*. Ou seja, o sistema apresentado nesta monografia está distribuído entre dois dispositivos: parte dele está em um dispositivo *Android* e parte dele está em um dispositivo *Arduino*. O sistema como um todo não funciona na ausência de uma destas partes.

2.5. Padrões de Projeto

Erich Gamma (1995, p. 12) cita uma definição dada por Christopher Alexander (1977): "Cada padrão descreve um problema que ocorre com frequência em nosso ambiente, e então descreve a raiz da solução daquele problema, de modo que você possa usar essa solução um milhão de vezes, sem nunca fazê-la da mesma forma duas vezes. "

Sendo assim, um padrão de projeto define uma solução padrão para um problema específico. Desta forma, ao implementar um padrão de projeto, sabemos que esta é uma solução concisa para o problema, universalmente utilizada e conhecida. Utilizar padrões de projeto é uma ótima prática de projeto, uma vez que os padrões facilitam o entendimento do projeto por outros, tendo formato e jargões bem definidos.

Padrões de projeto se aplicam a diversas áreas, não só a desenvolvimento de *software.* A própria definição supracitada fala sobre arquitetura e urbanismo, mas se encaixa perfeitamente no escopo da computação.

2.6. Considerações Finais

Este capítulo apresentou os principais temas abordados neste projeto. Um sistema distribuído entre um dispositivo *Android* e um protótipo de sistema embarcado, desenvolvido com *Arduino*. A utilização de padrões de projeto é um meio de manter a arquitetura com alta qualidade de implementação, baixo acoplamento, alta coesão, capacidade de reuso, fácil entendimento e edição do código.

No capítulo seguinte estaremos abordando as decisões de implementação. Isto envolve a criação de pacotes focados em solucionar problemas específicos, e a função de cada classe e interface dentro destes pacotes. A maioria dos problemas foram solucionados utilizando padrões de projeto, de forma a manter soluções concisas presentes na literatura.

CAPÍTULO 3: DESENVOLVIMENTO DO TRABALHO

Neste capítulo será abordado a arquitetura desenvolvida como os pacotes criados e suas classes, a importância de cada classe e cada interface no projeto e os principais algoritmos.

3.1. Considerações Iniciais

Nesta seção pontos chave do projeto serão apresentados, de forma que nas seções seguintes tenha-se ciência da responsabilidade de cada componente da aplicação.

3.1.1. Desenvolvimento com o Arduino

Para o desenvolvimento com o *Arduino* utilizou-se a *Arduino IDE*, um ambiente de desenvolvimento próprio para *Arduino*, assim como a linguagem utilizada foi C++, a linguagem suportada pela plataforma.

3.1.2. Desenvolvimento com o Android

Para o desenvolvimento com o *Android* utilizou-se o *Android Studio*, um ambiente de desenvolvimento próprio para o *Android*, disponibilizado pela Google. A linguagem de programação utilizada com é o Java, por requisitos do *Android SDK*.

3.1.3. Mensagem

A comunicação entre os dispositivos utilizando mensagens JSON (JavaScript Object Notation). Com isto é possível utilizar interpretadores disponíveis, além de se manter próximo aos modelos de comunicação utilizados em IoT, normalmente em JSON também.

Tanto na aplicação *Android* quanto no *sketch Arduino*, o manuseio das mensagens é delegado para uma classe, a qual está inserida em um pacote específico para lidar com a escrita e leitura das mensagens do *SmartPendant*.

3.1.4. Conexão

A conexão é feita utilizando *bluetooth.* Tanto no *Arduino* quanto no *Android* desenvolveu-se uma classe específica para lidar com *Bluetooth*, tornando mais transparente a utilização do mesmo.

3.1.5. Contexto

A aplicação *Android* possui diferentes comportamentos conforme o contexto em que se encontra. Um pacote foi criado a fim modelar o tratamento de contexto neste projeto. Uma classe fachada faz o serviço receber requisições dependentes de contexto e despachá-las ao contexto correto.

3.2. Projeto

Nesta seção serão apresentados os diagramas UML de cada pacote mencionado na seção anterior, assim como explicação sobre os relacionamentos dentro do pacote.

3.2.1. Mensagem

Para o trato das mensagens JSON temos o pacote **spmessage**. Este pacote lida com o processamento de mensagens recebidas e criação de mensagens para envio. Na figura 2 temos diagrama UML deste pacote. A seguir temos alguns pontos que se vale levantar sobre o diagrama:

3.2.1.1. SpMessage

A classe **SpMessage** é uma classe fachada para facilitar o uso deste pacote. Com ela é possível registrar observadores de eventos (pelo método *addEventListener*), entregar uma mensagem para processamento e recuperar uma instância do construtor de mensagens.

3.2.1.2. EventListener

A interface **EventListener** é necessária para implementar o padrão de projeto observador. Outros objetos podem se registrar como observador através do método *SpMessage.addEventListener* e, quando uma mensagem de evento for recebida, estes observadores serão notificados, recebendo um objeto da classe **SpEvent** com a representação do evento ocorrido.

3.2.1.3. SpEvent

Esta classe é utilizada para representar um evento ocorrido no pingente. Os atributos *source* e *type* modelam o padrão de botão utilizado e qual o tipo de ação que ocorreu sobre eles. Só é possível alcançar um objeto desta classe se registrando como um observador através de um objeto da classe **SpMessage**.

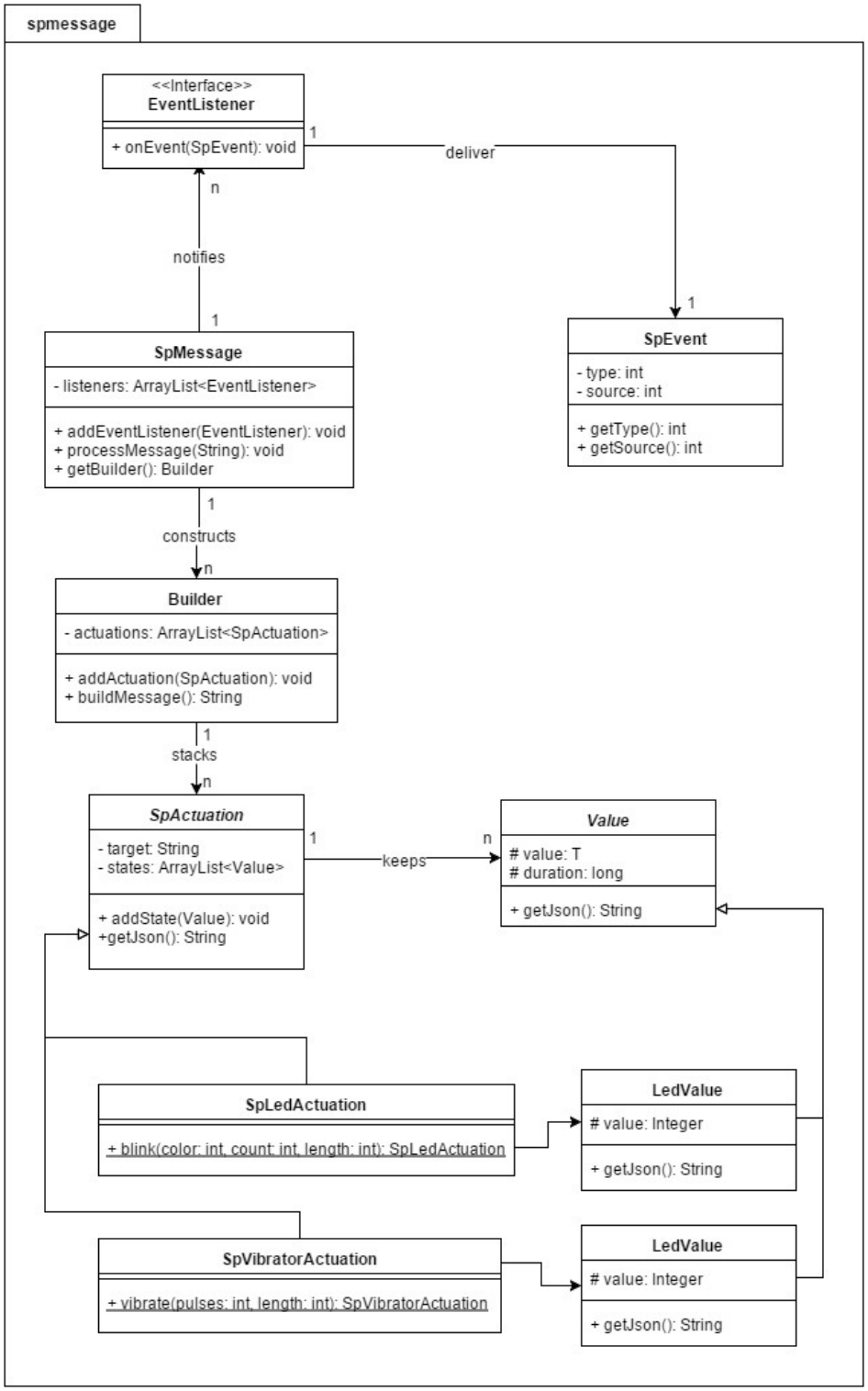
3.2.1.4. Builder

Esta classe é pertencente ao padrão de projeto *builder*. Ele é utilizado para acumular diversos comandos de atuação (**SpActuation**) e então criar uma única mensagem contendo todos os comandos requisitados.

3.2.1.5. SpActuation

Esta é uma classe abstrata, ou seja, não podem existir objetos desta classe. Ela é apenas utilizada para definir a estrutura base de um comando de atuação. Seus atributos consistem de um alvo para a atuação (*target*) e de uma lista de estados que este alvo deve assumir (*states*). Estes estados são objetos derivados da classe **Value**. Os métodos *addState* e *getJson* possuem uma implementação padrão e na maioria dos casos não necessitam de sobrescrita.

Figura 2 -- Diagrama UML do pacote spmessage.



Fonte: elaborado pelo autor.

3.2.1.6. Value

Esta também é uma classe abstrata. Ela visa representar o valor que um atuador deve assumir e por quanto tempo deve se manter nele. Este valor é de um tipo genérico *T*, ou seja, cada implementação pode utilizá-lo com o tipo de dados que quiser. O método *getJson* deve ser sobrescrito sempre, para implementar a correta representação deste valor na mensagem.

3.2.1.7. SpLedActuation

Esta classe estende **SpActuation**. Nela é utilizado o padrão de projeto método fábrica. Com isto um método cria o objeto de acordo com a atuação requisitada, facilitando o uso da classe. Outros métodos fábrica podem ser adicionados visando implementar diferentes comportamentos do LED.

O método fábrica desta implementação cria um comando para piscar o LED com uma determinada cor e um determinado tempo, tanto para aceso quanto para apagado. Outros métodos fábrica podem ser feitos caracterizando troca de cores ou variações na intensidade da luz.

3.2.1.8. LedValue

Estende a classe **Value**, a fim de implementar a representação de cores. O *Android* já possui uma classe para representar cores através de números inteiros. A classe **Color** do *Android* possui métodos para separar os canais de cores da representação inteira, assim como construir uma representação em número inteiro baseado nos canais vermelho, verde e azul. O método *getJson* constrói uma representação do valor passando separadamente cada um dos canais.

3.2.1.9. SpVibratorActuation

Estende **SpActuation** para, assim como a **SpLedActuation**, implementar um método fábrica que lida com a construção do objeto. Este método simplesmente cria uma

sequência de pulsos do motor vibratório com uma contagem de pulsos definida, e mesmo tempo para estado ligado e estado desligado.

É possível a criação de outros métodos fábrica para trabalhar com animações de *vibracall*, como fazer ritmos. Com isto seria possível atrelar diferentes ritmos a diferentes eventos do celular.

3.2.1.10. VibratorValue

Estende **Value** a fim de criar uma representação para o valor do motor vibratório. Nesta implementação, utilizamos um valor inteiro com 0 (zero) para desligado e 1 (um) para ligado. O método *getJson* cria a representação do valor corretamente na mensagem.

3.2.2. Conexão

O pacote **spbluetooth** trata o controle de conexão da aplicação *Android* com o *sketch Arduino*. Das quatro classes pertencentes ao pacote, três trabalham com processamento paralelo, a fim de que as tarefas relativas a conexão, não bloqueiem o ramo principal de execução. Na figura 3 está o diagrama UML deste pacote. A seguir temos alguns pontos que se vale levantar sobre o diagrama:

3.2.2.1. BluetoothSerial

A principal classe deste pacote. Gerencia a execução dos códigos paralelos, o estado de conexão do *bluetooth*, a conexão *bluetooth* e o registro de observador. Embora gerencie todo o serviço, não possui nenhuma característica peculiar. A implementação é simples e não requer ferramentas sofisticadas.

3.2.2.2. AsyncFindBondedDevice

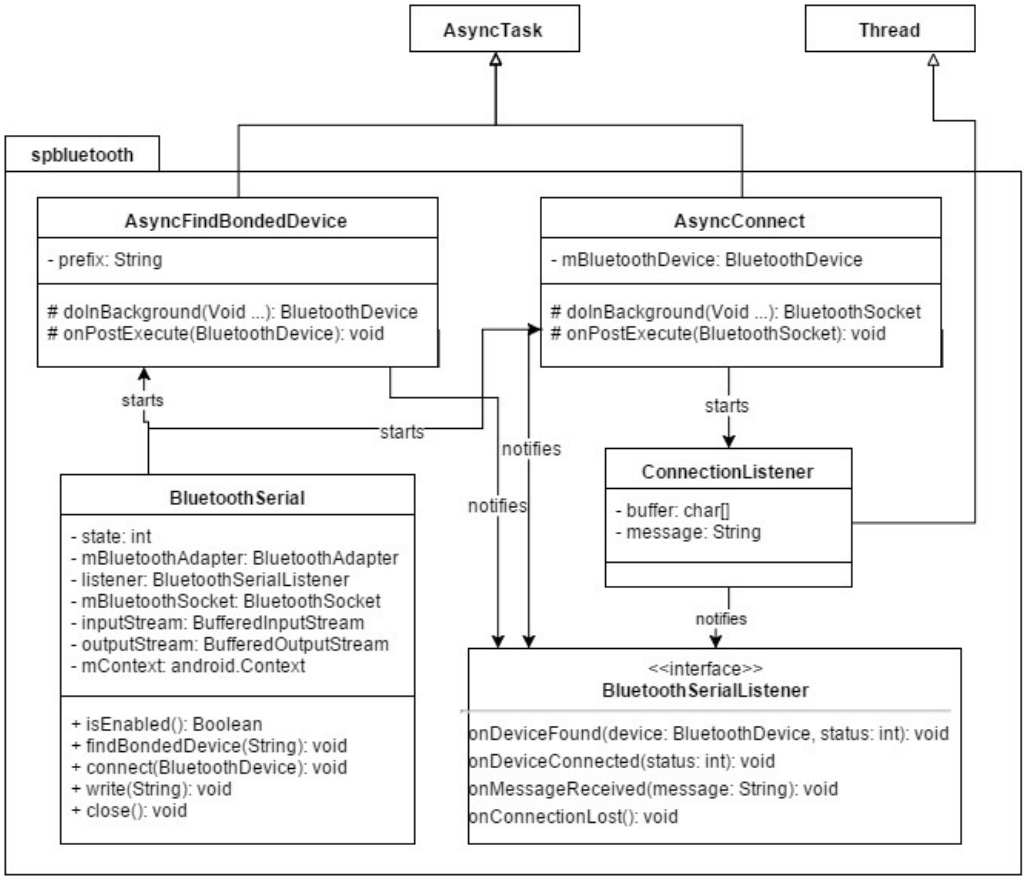
O primeiro código paralelo a ser executado. Lida com a busca do acessório dentre os dispositivos pareados. Esta classe estende **AsyncTask**, uma ferramenta de processamento paralelo do *Android*. *AsyncTasks* são focadas executar um trecho de código em paralelo e retornar um resultado, como se fosse simplesmente um método. Exceto pela

diferença que ocorre assincronamente. Caso o pingente seja encontrado, o observador registrado será notificado.

3.2.2.3. AsyncConnect

Lida com a conexão do *Android* com o pingente. Da mesma forma que o **AsyncFindBondedDevice**, esta classe também estende **AsyncTask** de forma a manter o processamento paralelo. Caso a conexão seja realizada com sucesso, o observador registrado é notificado

Figura 3 -- Diagrama UML do pacote spbluetooth.



Fonte: elaborado pelo autor.

3.2.2.4. ConnectionListener

Diferentemente das duas classes anteriores, o **ConnectionListener** não estende **AsyncTask**, ele estende **Thread**. Isto porque ele ouve a conexão enquanto a conexão está ativa. Ele permanece em execução por todo o tempo, diferente de um **AsyncTask**, que tem por propósito executar uma ação e encerrar. Quando uma mensagem é recebida, ela é convertida para **String** e enviada ao observador registrado.

3.2.2.5. BluetoothSerialListener

Esta é a interface de observação do pacote **spbluetooth**. Implementando esta interface e se registrando como observador (através do construtor da classe **BluetoothSerial**) é possível receber notificações sobre os estados de conexão do *bluetooth* e ser notificado quando uma mensagem do pingente for recebida.

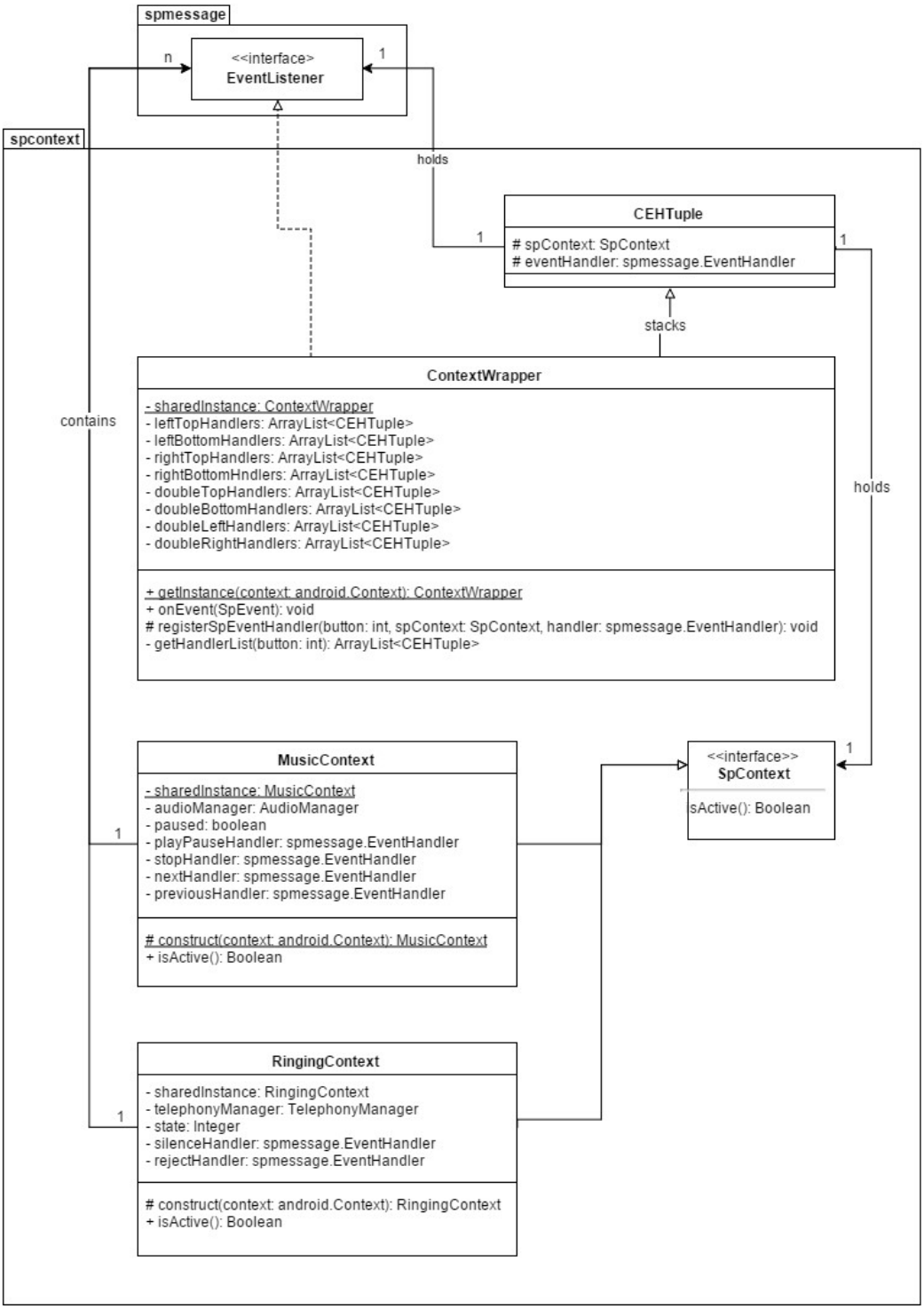
3.2.3. Contexto

Como o foco do *SmartPendant* é ter diferentes reações com base no contexto do *smartphone*, então o tratamento de eventos oriundos do pingente deve levar em conta o contexto ativo. Desta forma, o pacote **spcontext** lida com o tratamento de eventos com base no contexto. Na figura 4 temos o diagrama UML deste pacote. A seguir temos alguns pontos que se vale levantar sobre o diagrama:

3.2.3.1. ContextWrapper

Esta classe empacota todo o gerenciamento de contexto. Ela implementa **spmessage.EventHandler** e deve ser registrada em **spmessage.SpMessage** como o tratador de eventos oriundos do pingente. Trabalhando sobre o padrão de design cadeia de responsabilidade, mediante a ocorrência de um evento, esta classe é encarregada de estimar o contexto atual. Com base no contexto atual e no evento ocorrido, deve então despachar o processamento do evento para o tratador correto.

Figura 4 -- Diagrama UML do pacote spcontext.



Fonte: elaborado pelo autor.

Como este sistema deve conter somente uma instância de **ContextWrapper**, esta classe implementa o padrão de projeto *singleton*. Como esta é a única classe visível fora do pacote **spcontext**, então todos os contextos são instanciados dentro desta classe. Sendo assim, todos os contextos devem também ser *singletons*.

3.2.3.2. SpContext

Esta interface define a funcionalidade de um contexto. Como podemos ver no diagrama, somente um método está presente nesta interface. A única rotina exigida na implementação de um contexto é a *isActive*, que deve averiguar se o contexto em questão é o contexto ativo no momento.

Uma característica para um contexto, mas que o Java não provê através de interfaces, é a rotina estática *construct*. Uma rotina estática é um método que não pertence a uma instância da classe, mas pertence à classe em si. Logo é possível executá-lo antes que qualquer objeto seja instanciado. O Java não permite que interfaces possuam métodos estáticos.

O método *construct* é a implementação do padrão de projeto método fábrica. Este método não só instância o contexto como também registra todos os tratadores de eventos deste contexto.

3.2.3.3. MusicContext

Esta é uma implementação de **SpContext** para lidar com o contexto "tocando música". Neste contexto deve ser possível controlar o tocador de música utilizando o pingente. Note que há quatro atributos, nesta classe, do tipo **spmessage.EventHandler**, um para cada ação que possa ser tomada através do pingente: *play/pause*, *stop*, *next*, *previous*.

Sendo **spmessage.EventHandler** uma interface, não pode ser instanciada. Desta forma, novas classes devem implementá-la e executar as ações disponíveis neste contexto. Como cada uma destas classes são simples (um único método) e vão possuir uma única instância, decidiu-se por utilizar classes anônimas.

Em Java é possível ter classes anônimas. Para isto deve-se instanciar uma interface como se fosse uma classe, mas logo abaixo se transcreve a implementação desejada. Desta forma poupa-se criar diversos arquivos com pequenas classes cada.

Por trás destas classes anônimas, permeia o padrão de projeto estratégia. Nesta abordagem temos um problema a ser resolvido: tratar o evento de botões acionados. Mas, para um contexto, temos diversas estratégias de tratar este evento. Dependendo de quais os botões utilizados, devemos trabalhar com uma estratégia diferente.

3.2.3.4. RingingContext

Esta é uma implementação de **SpContext** para lidar com contexto "recebendo chamada". Neste contexto deve ser possível silenciar uma chamada recebida, ou rejeitá-la. Desta forma ela possui dois atributos do tipo **spmessage.EventHandler**: *silenceHandler* e *rejectHandler*, implementados como classes anônimas e também sendo uma implementação do padrão de projeto estratégia.

3.3. Descrição das Atividades Realizadas

Nesta seção, os algoritmos mais relevantes do projeto serão explicados em detalhe. Sendo eles: conexão *bluetooth*, recepção de mensagem, tratamento de evento e envio de mensagem. Nos diagramas de sequência apresentados introduzimos uma nova classe: **AccessoryDaemon**.

Esta classe representa a o serviço de execução em segundo plano da aplicação *Android*. Em outras palavras, é um programa que fica executando sem ter interface gráfica e gerencia a criação e configuração das classes apresentadas na seção anterior. O **AccessoryDaemon** também é o observador do **BluetoothSerial**, sendo informado dos estados de conexão e de mensagens recebidas.

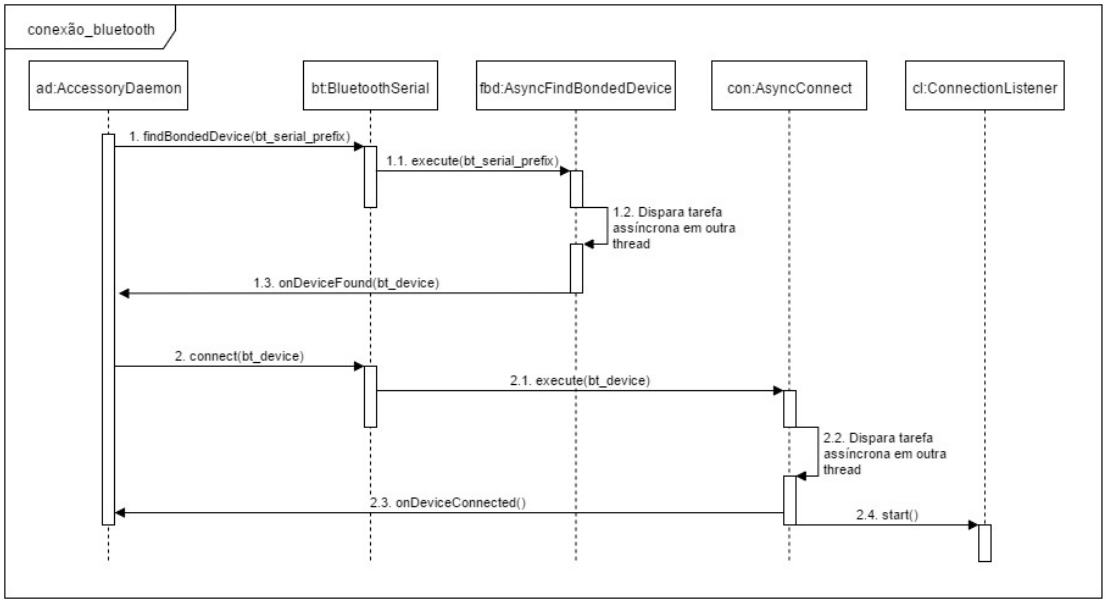
Uma vez que o foco deste trabalho é explorar a arquitetura utilizada no *SmartPendant*, então todas as partes relacionadas com recuperar informações do *Android* serão ignoradas. Simplesmente assume-se que temos acesso as informações apresentadas. É possível encontrar como se recupera informações específicas do *Android* através da

página do desenvolvedor *Android* (https://developer.android.com), assim como em diversos fóruns.

3.3.1. Conexão Bluetooth

Na figura 5 temos o diagrama de sequência da conexão *bluetooth*.

Figura 5 -- Diagrama de sequência da conexão bluetooth.



Fonte: elaborado pelo autor.

Durante uma conexão *bluetooth*, primeiro se busca o dispositivo desejado dentre os dispositivos pareados. Para isto, **BluetoothSerial** dispara a **AsyncFindBondedDevice**. Em um *thread* diferente, a tarefa assíncrona recupera a lista de dispositivos pareados e busca, entre eles, o dispositivo requisitado. Caso encontre, então notifica o observador registrado através de *BluetoothSerialListener.onDeviceFound*, enviando o dispositivo encontrado como parâmetro.

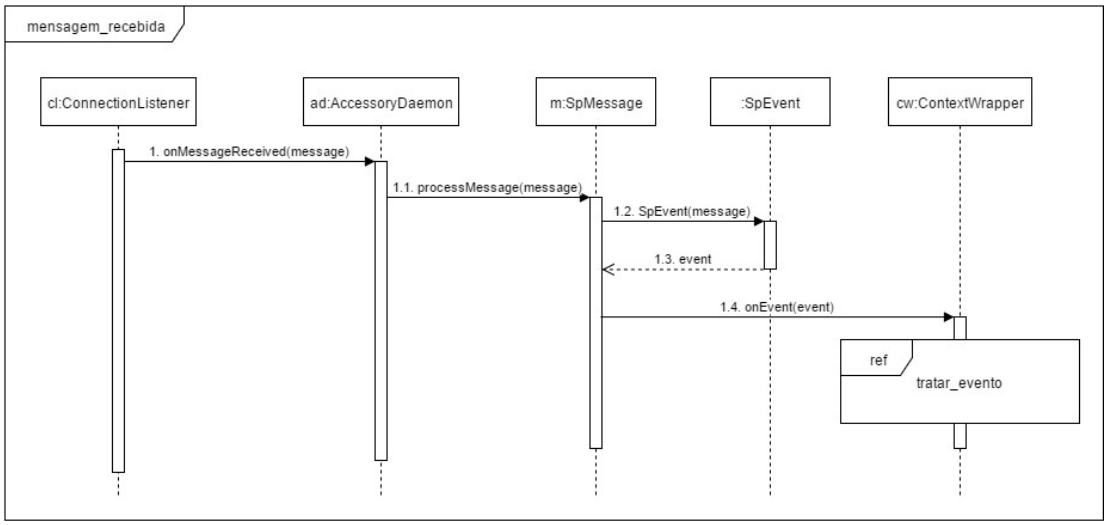
Tendo o objeto do dispositivo *bluetooth*, é solicitada a conexão com o dispositivo em questão. Para isto, **BluetoothSerial** dispara a **AsyncConnect**. Em um *thread* diferente, a aplicação *Android* solicita ao dispositivo a conexão. Em sucesso, o observador é notificado através de *BluetoothSerialListener.onDeviceConnected*. Ainda, a *thread*

ConnectionListener é então disparada e fica ouvindo a conexão, esperando por mensagens recebidas.

3.3.2. Recepção de Mensagem

Na figura 6 temos o diagrama de sequência da recepção de mensagem.

Figura 6 -- Diagrama de sequência da recepção de mensagem.



Fonte: elaborado pelo autor.

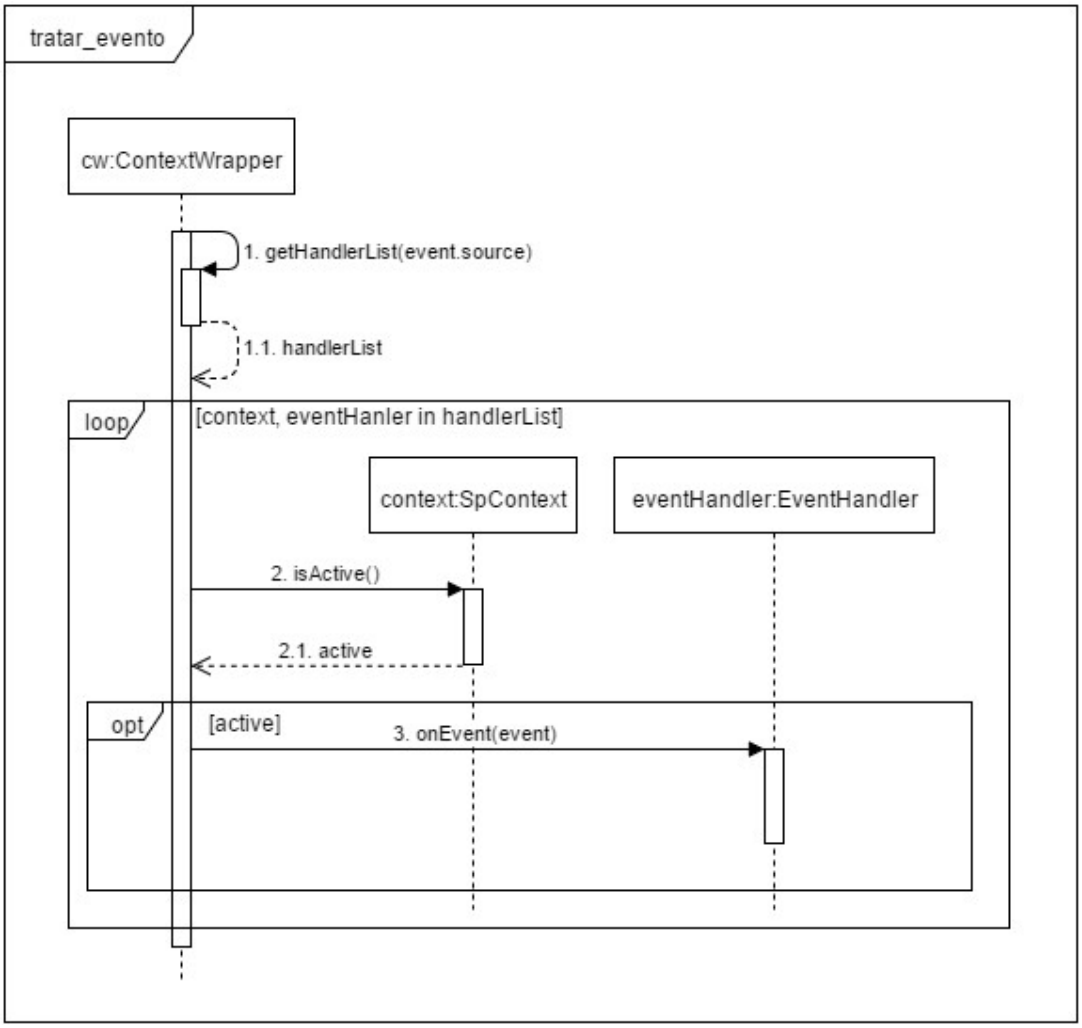
Ao receber uma mensagem, **ConnectionListener** notifica o observador registrado e entrega a mensagem utilizando *BluetoothSerialListener.onMessageReceived*. O **AccessoryDaemon** envia a mensagem recebida para **SpMessage**, a fim de ser processada. Em **SpMessage**, um objeto **SpEvent** é construído utilizando a própria mensagem. Então **SpMessage** notifica o observador registrado do evento ocorrido.

Para o processamento da mensagem recebida foi utilizado o módulo de JSON próprio do *Android*. Esse é um dos grandes benefícios de se utilizar mensagens JSON, a maioria das tecnologias já possuem um interpretador.

3.3.3. Tratar Evento

Na figura 7 temos o diagrama de sequência do tratamento de eventos.

Figura 7 -- Diagrama de sequência do tratamento de eventos.



Fonte: elaborado pelo autor.

A classe **ContextWrapper** possui oito listas de tratadores de eventos. Uma lista para cada um dos possíveis padrões de botão do *SmartPendant*. Cada uma destas listas possui objetos **CEHTuple**. Esta classe é utilizada para unir um contexto com um de seus tratadores de evento. Desta forma é possível trabalhar com diversos contextos simultâneos. Por exemplo: se o dispositivo está com a tela travada (este contexto não foi implementado) e está tocando música, é possível evitar sobreposição de padrões de botão entre estes contextos. Desta forma pode-se tomar ações do contexto “tela travada” enquanto se ouve música.

Como **ContextWrapper** possui diversas listas de tratadores de eventos, então primeiro é necessário recuperar a lista correta com base no padrão utilizado. Isto é feito no passo 1 e em seus subpassos.

Tendo a lista em mão, então é necessário verificar se algum de seus possíveis contextos está ativo. Esta é a responsabilidade de **ContextWrapper** dentro do padrão de projeto cadeia de responsabilidades. Caso um contexto ativo seja encontrado, então o tratamento de eventos é despachado para seu respectivo tratador.

O tratador de eventos, específico para o padrão de botões utilizado, e para o contexto atualmente ativo, toma então as ações esperadas sobre o *Android*. Esta é a segunda e última responsabilidade a ser cumprida no padrão de projeto cadeia de responsabilidades.

3.3.4. Envio de Mensagem

Na figura 8 temos o diagrama de sequência do envio de mensagem.

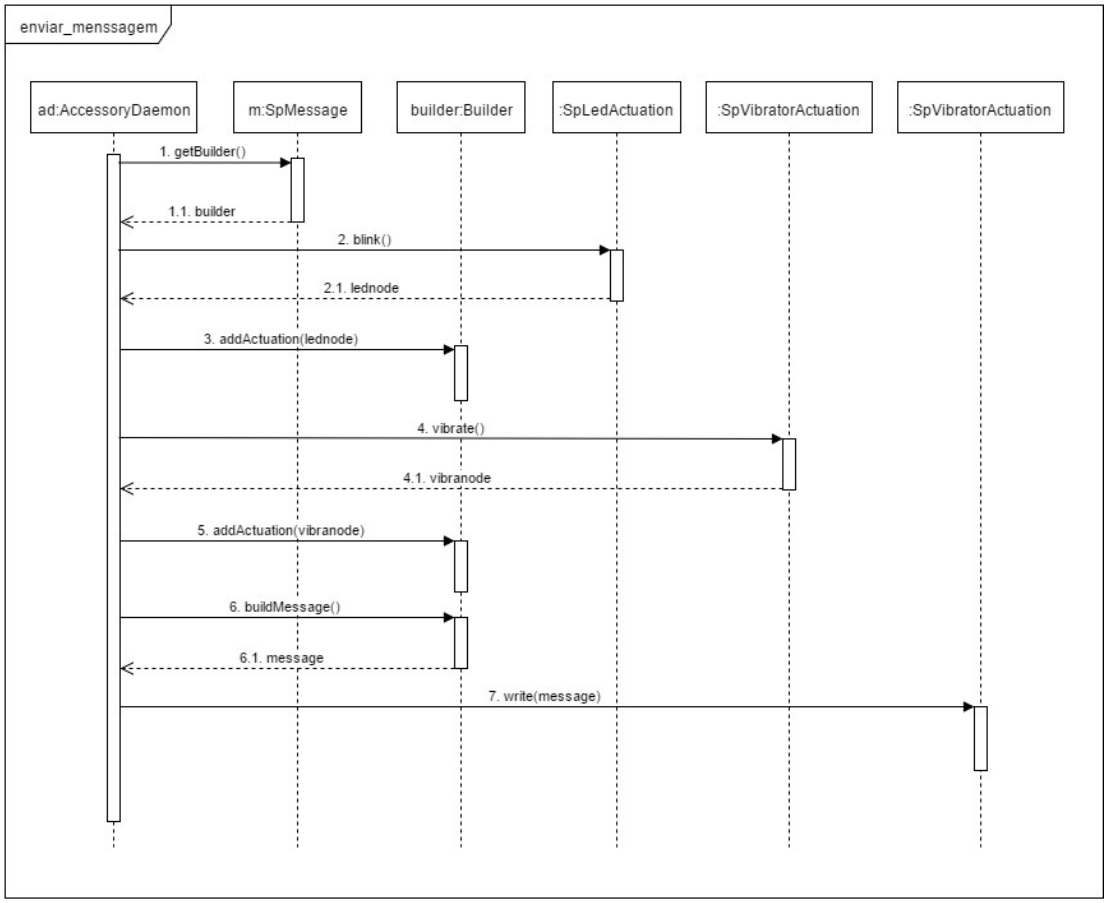
Inicialmente recupera-se o **Builder** de **SpMessage**. A utilização do padrão de projeto *builder* consiste em ir alimentando-o com partes de um produto, e então recuperar o produto final. Neste caso iremos preenchendo-o com pedaços de mensagem, para então montar uma mensagem completa.

Tendo um **Builder** criamos a atuação de LED e de vibração através de seus respectivos métodos fábrica. Adicionamos essas atuações a mensagem que estamos produzindo. Construímos a mensagem completa e a enviamos por *bluetooth* ao pingente.

3.4. Resultados Obtidos

O sistema se comportou bem tanto no *Android* respondendo ao pingente quanto no pingente respondendo ao *Android*. O requisito de propor uma arquitetura escalável para o gerenciamento do periférico foi atingido e o produto funciona.

Figura 8 -- Diagrama de sequência do envio de mensagem.



Fonte: elaborado pelo autor.

3.5. Dificuldades e Limitações

Dentre as dificuldades encontradas, a principal foi relativa ao aparelho *Android* utilizado. Possuindo o sistema operacional na versão 2.3.6, enquanto que a versão 6.0 já foi anunciada. Algumas das funcionalidades esperadas ainda não estavam disponíveis nesta versão. Outras funcionalidades tinham que ser tratadas de maneira diferente da apresentada na documentação. Outra dificuldade foi a falta de experiência do aluno com a linguagem de programação C++, utilizada no *Arduino*.

Outras dificuldades encontradas durante o desenvolvimento foi quando a conexão *bluetooth* devia ser realizada. A escolha foi por quando a antena *bluetooth* do celular está ligada. Parece simples falando desta maneira, mas na programação há todo mecanismo de

comunicação interprocesso, a fim de se registrar como um observador da antena *bluetooth*, de forma a ser notificado quando o estado da antena mude.

Algo que deu um pouco de trabalho também foi a leitura das mensagens. Uma característica que ocorre tanto no *Android*, quanto no *Arduino* ou em qualquer outro sistema que lide com comunicação serial. A mensagem não chega inteira no *socket* de comunicação. Eram necessários em torno de duas a quatro leituras para se recuperar uma mensagem do *socket*. Desta forma é necessário manter um caractere de fim de mensagem que, neste caso, foi utilizado o caractere de quebra de linha "\n".

Um problema mais fora do alcance do aluno são os aplicativos de terceiros. Como durante o controle do tocador de música. O *Android* confirmava que havia música sendo tocada, mas os comandos sobre o tocador não funcionavam, pois o aplicativo *PlayerPro* não implementava o mecanismo de comunicação interprocesso padrão para controle de música. Mas o tocador de música padrão do *Android* reconheceu os comandos sem problema nenhum.

Quanto as limitações do projeto estão na aplicabilidade. Quando se fala de internet das coisas vemos dispositivos de coleta de dados, que ficam o tempo todo submetendo leituras de sensores para a nuvem. Esta não é uma boa aplicação desta arquitetura. Ela é mais voltada para a interação *smartphone* e periférico, trocando somente mensagens de eventos e atuação.

3.6. Considerações Finais

Neste projeto os requisitos foram atingidos com sucesso, propondo um modelo escalável para diferentes aplicações, com alta coesão e baixo acoplamento. Uma implementação bem modularizada permite qualquer um fazer uso da arquitetura base e adaptar o sistema conforme suas necessidades.

CAPÍTULO 4: CONCLUSÃO

Neste capítulo abordaremos as contribuições que este projeto gerou, as contribuições deste projeto para o aluno, as contribuições do curso para o desenvolvimento do projeto e a experiência do aluno com o curso. Por fim, sugestões de continuidades do projeto são apresentadas.

4.1. Contribuições

A principal contribuição deste trabalho foi uma arquitetura base para o relacionamento *smartphone*/periférico. Uma arquitetura de baixo acoplamento e alta coesão, cobrindo conexão *bluetooth*, recepção de mensagem, tratamento de eventos e criação de mensagens.

O trabalho proporcionou ao aluno novas experiências com C++, prototipação de circuitos eletrônicos, uso do JSON fora do JavaScript e comunicação serial sobre uma conexão *bluetooth*.

4.2. Relacionamento entre o Curso e o Projeto

A principal base para o desenvolvimento deste projeto a disciplina de Programação Orientada à Objeto, uma vez que conceitos relacionados como interfaces e herança foram essenciais para o desenvolvimento.

A disciplina Sistemas Operacionais foi útil pelos conceitos de processamento paralelo, comunicação interprocessos e até mesmo pelo desenvolvimento *Android*, tema abordado em Sistemas Operacionais II.

A disciplina Sistemas Distribuídos foi útil no desenvolvimento das mensagens e no tratar da característica assíncrona dos dispositivos operando em conjunto.

A disciplina Estágio Supervisionado I foi essencial, onde o aluno estagiou no SIDI (Samsung Instituto de Desenvolvimento para a Informática) e adquiriu grande experiência no desenvolvimento para *Android*.

A disciplina Estágio Supervisionado II foi essencial, onde o aluno estagiou na Circuitar e adquiriu relevante experiência no desenvolvimento de bibliotecas em C++ para *Arduino*.

A iniciação científica foi de grande utilidade pela experiência em produção de relatório, monografia e desenvolvimento com a linguagem de programação Java.

4.3. Considerações sobre o Curso de Graduação

O curso traz uma base sólida em desenvolvimento de sistemas computacionais, assim. Embora a base eletrônica, de controle e de processamento de sinais sejam muito mais teóricas do que práticas, ainda assim são suficientes para se manter em uma conversa dentro de uma empresa.

Isso é inclusive algo que falta no curso. Vemos muita teoria sobre circuitos, controle e processamento de sinais, mas não fazemos algo que funcione sobre esta teoria. Quando nos deparamos com uma situação de aplicação, temos noção da teoria matemática, mas não sabemos por onde começar a implementação, e nem como ela deve ser.

Falta no curso uma disciplina sólida de projeto. Lidar com metodologias ágeis, sistemas de versionamento e testes unitários. São temas essenciais no mercado de trabalho que nem todos têm contato durante o estágio.

Um ponto forte: talvez muito mais pela instituição do que pelo curso, é o reconhecimento. Em toda empresa que você entra, as pessoas te olham de maneira diferente, te tratam de maneira diferente, mas da mesma forma esperam de você resultados diferentes.

4.4. Trabalhos Futuros

A continuidade deste trabalho pode ser feita adicionando novas funcionalidades. Implementando novos contextos e adicionando novos eventos ao pingente, como por exemplo reconhecimento de movimento, utilizando uma IMU (acelerômetro, giroscópio e magnetômetro) ou reconhecimento de posição utilizando um GPS.

Outra vertente que pode surgir deste trabalho, é utilizar o pingente como periférico de outro sistema, como um sistema de automação residencial, por exemplo. Isso aprofundaria ainda mais o projeto em conceitos como sistemas embarcados e computação ubíqua.

REFERÊNCIAS

1. ALEXANDER, C et al. **A Pattern Language**. Nova York: Oxford University Press, 1977.

2. BANZI, M.; SHILOH, M. **Primeiros Passos com o Arduino.** Tradução Aldir José Coelho Corrêa da Silva. 2.ed. São Paulo: Novatec Editora Ltda., 2015.

3. COULOURIS, G. et al. **Sistemas Distribuídos**, Conceitos e Projetos. 4.ed. Porto Alegre: Artmed Editora S.A., 2007.

4. GAMMA, E. et al. **Design Patterns**, elements of reusable object-oriented software. Boston: Addison-Wesley Longman Publishing Co., Inc., 1995

5. JACKSON, W. **Android Apps for Absolute Begginers.** Berkeley: Apress, 2011

6. KIRK, S. The Wearables Revolution: Is Standardiztion a Help or a Hindrance? **Consumer Eletronics Magazine, IEEE**, v.3, n.4, p.45-50, out. 2014.

7. LAPINSKI, M. et al. **Wearable Wireless Sensing for Sports and Ubiquitous Interactivity.** MIT Media Lab, Cambridge, 2011

8. LYYTINEN, K.; YOUNGJIN, Y. Issues and Challenges in Ubiquitous Computing. **Communications of the ACM**, v.45, n.12, dec. 2002.

9. MORRIS, D. et al. Wearable Sensors for Monitoring Sports Performance and Training. In: International Summer School and Symposium, 5, 2008, Hong Kong, Medical Devices and Biosensors, IEEE, 2008, p.121-124. Disponível em: < http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4575033>. Acesso em: 05 nov. 2015.

10. WHITE, E. **Making Embedded Systems**, Design Patterns for Great Software. [s.l.]: O'Reilly Media, 2011.