PADRÕES DE PROJETO PARA DESENVOLVIMENTO DE APLICAÇÕES DE COMPUTAÇÃO VESTÍVEL

Álvaro Watanabe Filho **Lucas Eidi Takahashi** **Michel Jai Kil Oh**

Orientador: *Prof. Dr. Luciano Silva*

Resumo. Computação Vestível refere-se a uma possibilidade de interação humano-computador, onde os *gadgets* estão diretamente conectados com usuário e, em termos mais gerais, o usuário "veste" seu gadget. Os *gadgets* vestíveis têm a intenção de capturar e fornecer dados e informações de/aos usuários através de interfaces, normalmente conhecidas, como relógios, óculos, pulseiras, dentre outros. Embora a área de Computação Vestível já exista há algum tempo, ainda não existe um catálogo de soluções comuns para problemas comuns como, por exemplo, obtenção de dados de sensores, comunicação, processamento e feedback. Dentro deste contexto, este trabalho propõe um conjunto de padrões de projeto para problemas comuns levantados através de pesquisa bibliográfica nesta área. Além disto, foi desenvolvida uma aplicação na área de jogos digitais para mostrar um exemplo de utilização dos padrões. Através desta aplicação, também foi possível levantar alguns pontos para discussão das vantagens e desvantagens do uso do catálogo proposto.

Palavras-chave: *Computação Vestível, Padrões de Projeto, Interação Humano-Computador, Engenharia de Software.*

Abstract. Wearable computing refers to a possibility of human-computer interaction, where the gadgets are directly connected with users and, more generally, users "wear" their gadgets. Wearable gadgets are intended to capture and provide data and information to users through interfaces such as smartwatches, smartglasses, bracelets, among others. Although the area of wearable computing has been around for some time, there is not a catalog of common solutions to common problems such as data acquiring from sensors, communication, processing and feedback. Within this context, this work proposes a design pattern catalog for common problems raising in development of Wearable Computing applications. Furthermore, an application on digital games was developed in order to explain how to apply the patterns. Through this application, it was also possible to raise some points for discussion of the advantages and disadvantages of the proposed pattern.

Keywords: *Wearable Computing, Design Patterns, Human-Computer Interaction, Software Engineering.*

1. INTRODUÇÃO

A Computação Vestível (Wearable Computing) é uma área emergente em Interação Humano-Computador, onde os *gadgets* estão diretamente conectados com usuário e, em termos mais informais, diz-se que os usuários “vestem” os *gadgets*. Já existem vários dispositivos de Computação Vestível disponíveis no mercado como smartwatches, smartglasses, pulseiras, dentre outros, que tentam explorar maneiras mais naturais de obter e acessar dados e informações.

Embora o número de dispositivos apresente uma tendência a crescer, o desenvolvimento de aplicações para Computação Vestível ainda carece de bibliotecas e soluções para problemas comuns envolvendo obtenção de dados de sensores, comunicação, processamento e *feedback*. Assim, é de grande interesse computacional e industrial a proposta de bibliotecas e soluções para problemas comuns como, por exemplo, padrões de projeto, para suportar o desenvolvimento de aplicações em Computação Vestível.

Assim, dentro deste contexto, o objetivo deste artigo é propor o uso de padrões de projeto para os principais problemas encontrados no desenvolvimento de aplicações para Computação Vestível: aquisição de dados de sensores, comunicação e armazenamento de dados, processamento e feedback. Os padrões foram propostos após uma pesquisa bibliográfica para levantamento de soluções comuns de problemas compartilhados nos sistemas de Computação Vestível. Além dos padrões, foi adaptado um jogo do game engine Unity3D para receber os dispositivos vestíveis Oculus Rift e Leap Motion, para demonstrar um exemplo de aplicação dos padrões e explorar interfaces de entrada e saída em Computação Vestível. Como existem poucos trabalhos na área de desenvolvimento de aplicações para Computação Vestível, a proposta do uso de padrões que compreenda as principais atividades de uma aplicação pode servir de base para iniciantes em desenvolvimento na área, assim como guiar possíveis refatorações de aplicações existentes.

Este artigo está organizado da seguinte forma: a Seção 2 traz o referencial teórico de suporte a padrões de projeto e Computação Vestível; a Seção 3 evidencia a metodologia utilizada no desenvolvimento do trabalho; a Seção 4 apresenta os resultados (proposição de padrões); a Seção 5 apresenta a implementação da aplicação em Unity3D e, finalmente, a Seção 6 encerra com as conclusões e trabalhos futuros.

2. REFERENCIAL TEÓRICO

Nesta seção, serão tratadas a definição de Padrões de Projeto (ou Design Patterns), quais são os elementos que caracterizam padrão, a definição de uma Computação Vestível, as características de um dispositivo vestível, as problemáticas encontradas para cada característica e outros *patterns* propostos para cada problemática.

2.1. PADRÕES DE PROJETO

Padrões de projeto, ou *Design Patterns*, são padrões que descrevem um problema que ocorre com frequência em um determinado ambiente, e provê uma solução de objetos e interfaces que pode ser utilizada milhões de vezes sem a necessidade de criá-la mais de uma vez (GAMMA et al, 1994). Em geral, um padrão tem 4 elementos essenciais: o nome do padrão, usado para descrever o problema, solução e consequências em uma ou duas palavras; o problema, que descreve qual o problema visto para a construção do padrão; a solução, que descreve os elementos que fazem parte do desenvolvimento, seus relacionamentos, responsabilidades e colaborações; e as consequências, que são os resultados do padrão aplicado. Dentre todos os padrões de projeto, o catálogo GoF (*Gang of Four*) foi o pioneiro em propor este tipo de solução para organizar o desenvolvimento, juntando e padronizando diversas práticas que já existiam, conforme mostra a Figura 1.

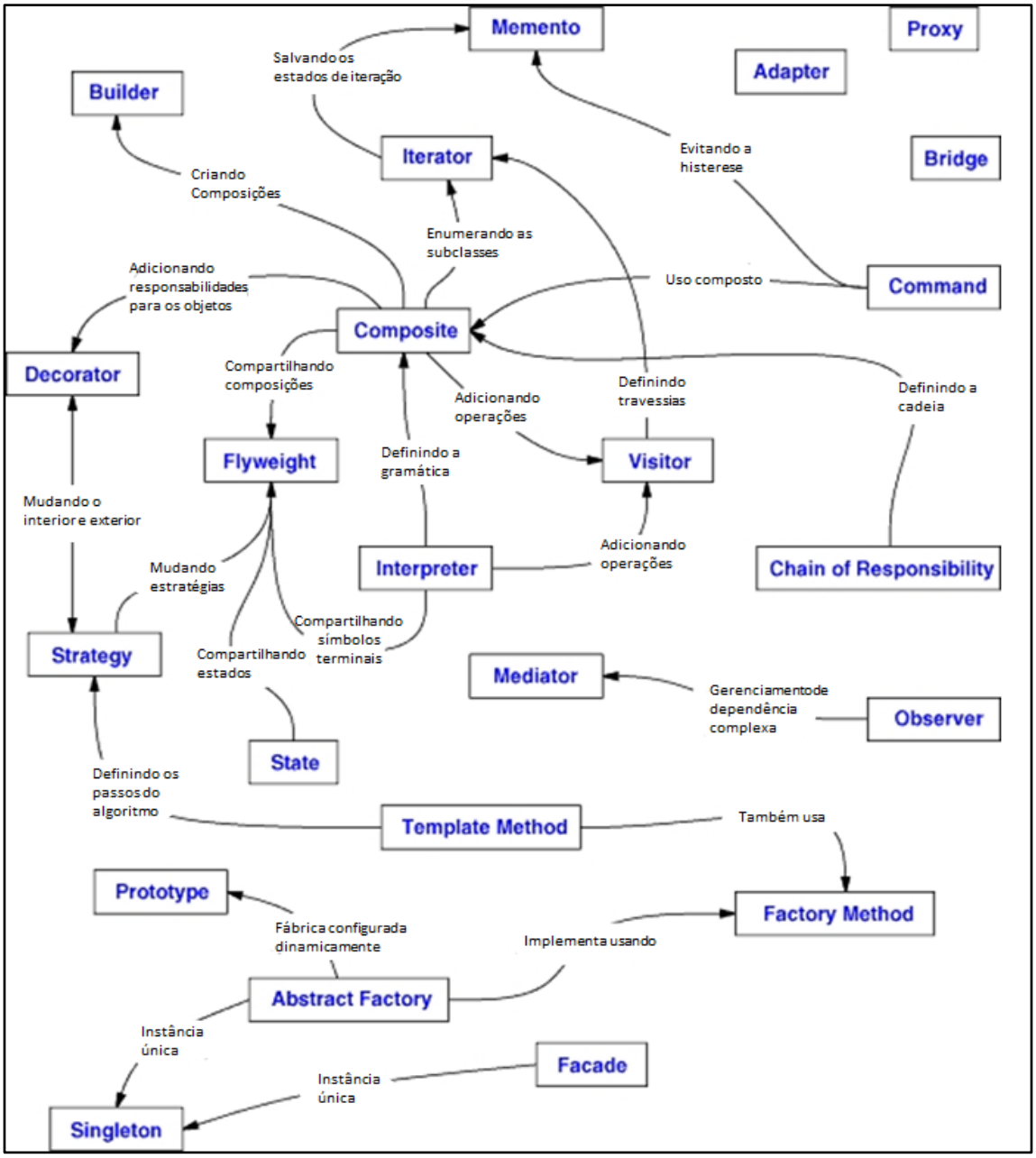

Figura 1: Mapa dos padrões do catálogo GoF e seus relacionamentos.

Os padrões do catálogo GoF são divididos em propostas de padrões criacionais, estruturais e comportamentais. Em todo o catálogo, observa-se as seguintes informações em cada padrão: nome, intenção, conhecido como, motivação, aplicabilidade, estrutura, participantes, colaborações, consequências, implementação, exemplo de código, usos conhecidos e padrões relacionados.

2.2. COMPUTAÇÃO VESTÍVEL

A Computação Vestível, ou *Wearable Computing*, é "...o estudo ou prática de inventar, moldar, construir ou usar os dispositivos sensoriais e computacionais em miniatura transmitidas pelo corpo." (MANN, 2014), ou seja, é todo o estudo da computação que envolve dispositivos sensoriais e seu uso no corpo. Ainda de acordo com o autor, a

Computação Vestível pode ser uma vestimenta ou utilizado sobre uma, conforme exemplos na Figura 2.

Figura 2: Exemplos de uma Computação Vestível (*Wearable Computing*).
Fonte: http://bits.blogs.nytimes.com/2012/04/17/wearable-computers-are-the-next-platform-wars-report-says/?_r=1

Smartwatches, marcado com o número 1 na figura, óculos de realidade aumentada, número 2, *Smartbands*, número 3, clipes marcadores de passos, número 4, óculos de realidade virtual, número 5, e até dispositivos de sensores de movimento, número 6, são exemplos de Computação Vestível.

Já um dispositivo vestível, de acordo com Sazonov e Neuman (2014), é um pequeno computador com sensores e capacidade de processamento, armazenamento, comunicação e, eventualmente, atuadores que provêm uma resposta para o usuário. Ou seja, um dispositivo vestível pode ser definido como um computador que se usa no corpo e contém os seguintes componentes: sensores (*Sensoring* ou *Tracking*), processamento (*Processing*), armazenamento (*Storage*), comunicação (*Communication*) e resposta (*Feedback*), sendo o último não obrigatório em todos os dispositivos.

Os sensores e as respostas, ainda na visão de Sazonov e Neuman (2014), são sistemas microeletromecânicos, e as vezes utilizando-se também de nanotecnologias, que buscam ler e enviar atividades sensoriais, seja para audição, visão, toque, entre outros. A evolução nesses sistemas aprimorou as interfaces sensoriais.

Mas, segundo Ratzka (2008), o rastreamento dos sensores e sua resposta tem um problema no desenvolvimento com a multimodalidade, ou seja, quando há a escolha de dois ou mais tipos de sensores que a aplicação deve trabalhar. O autor também oferece algumas sugestões para contornar este problema, observado na Figura 3.

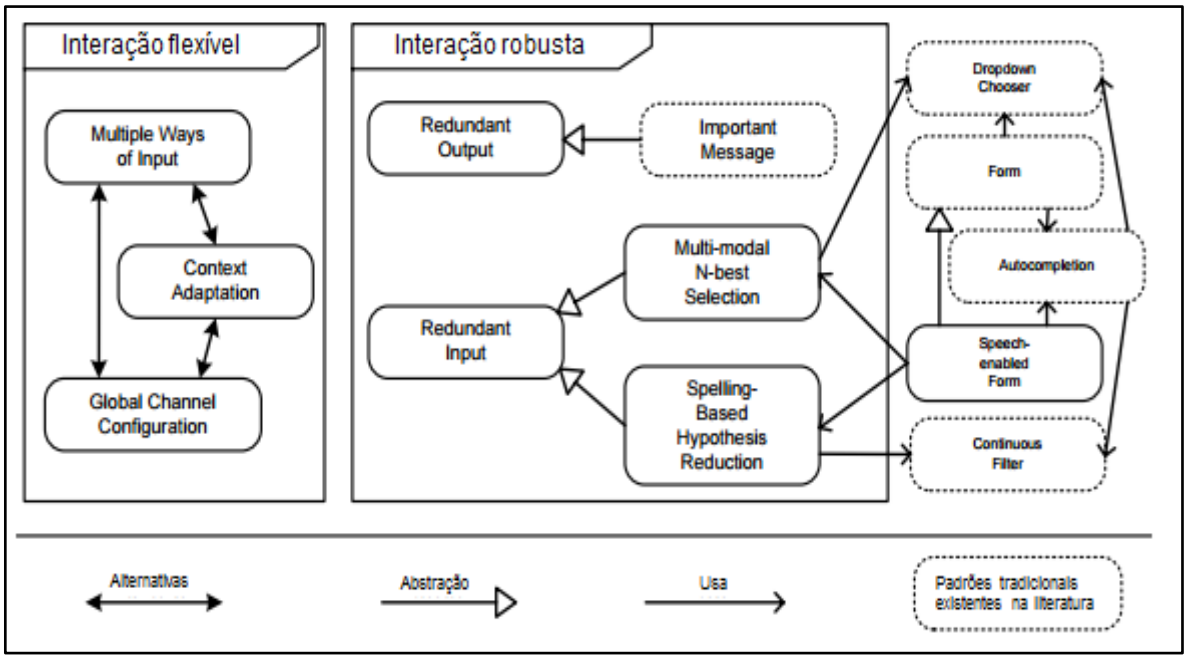

Figura 3: Mapa dos padrões propostos por Ratzka(2014).

Os padrões de Ratzka, que estão com os nomes em inglês na figura, são agrupados em dois grupos: a interação flexível, focada em acessibilidade e usabilidade entre contextos, e a interação robusta, focada em assegurar a comunicação entre usuário e sistema. Eles têm um padrão mais comportamental, têm apenas uma estrutura lógica para a solução. Entretanto é possível adaptá-los para criar uma estrutura de classes.

A comunicação é uma infraestrutura que conecta dispositivos para a troca de informação (PAHLAVAN; LEVESQUE, 2005). A comunicação referida por Sazonov e Neuman (2014) é a comunicação sem fio, ou seja, sem a necessidade de cabos para este fim. Sua rápida evolução nas técnicas permitiu uma entrega mais eficiente da informação.

No entanto, se o desenvolvimento na aplicação não for programado corretamente, é possível degradar a qualidade da comunicação. Völter, Kircher e Zdun (2004) relatam problemas de dificuldade de manutenção do código, a falta de escalabilidade e transparência para o desenvolvimento em arquiteturas distribuídas na comunicação, e sugerem alguns padrões para o melhor controle do envio e recebimento das informações, observado nas figuras 4 e 5.

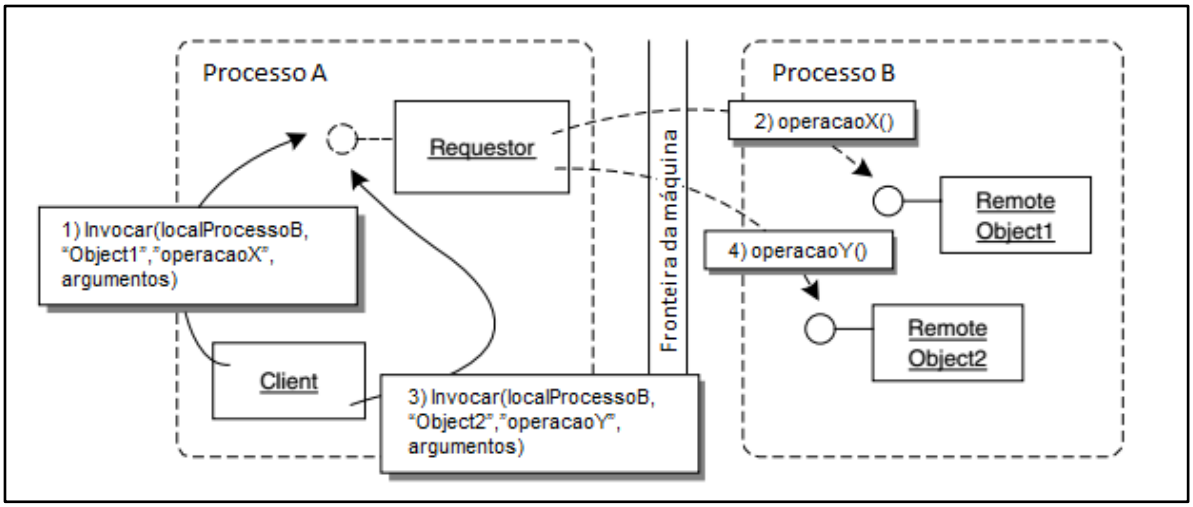

Figura 4: Exemplo de padrão, o *Requestor*, proposto por Völter, Kircher e Zdun (2004).

O *Requestor* é um padrão utilizado no processo do cliente para solicitar um objeto remoto através de uma operação. O cliente chama o Requestor através da função invocar, passando como parâmetro o local remoto, o objeto, a operação e os argumentos para a chamada do objeto remoto.

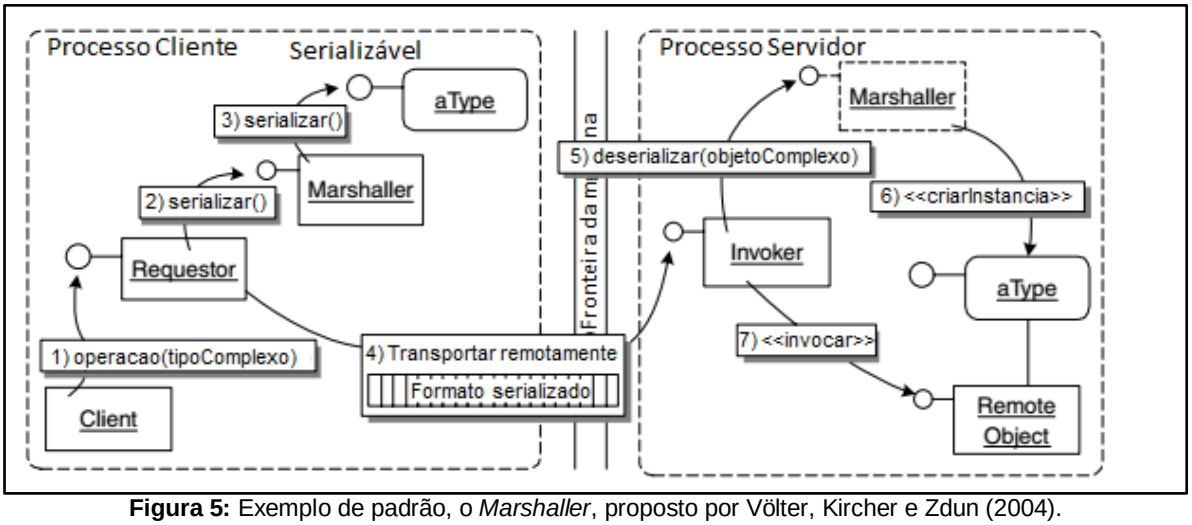

Figura 5: Exemplo de padrão, o *Marshaller*, proposto por Völter, Kircher e Zdun (2004).

O *Marshaller* é um padrão para facilitar as serializações dos dados que são enviados e recebidos. O *Requestor*, visto antes, pede para o *Marshaller* serializar os dados. Depois que o *Marshaller* serializa, o *Requestor* envia estes dados para o seu destino. Lá, o processo é recebido pelo *Invoker*, que chama o *Marshaller* interno para desserializar os dados enviar para o objeto remoto.

Poll Objects, conforme mostrado na Figura 6, são objetos que tornam possível o paralelismo. O *Requestor* envia os dados para o objeto remoto, que se encarrega de devolver os dados processados para um *Poll Object*. Cada novo resultado é armazenado em um *Poll Object* diferente. Eles têm uma função de estado para avisar ao cliente se os dados estão disponíveis ou não.

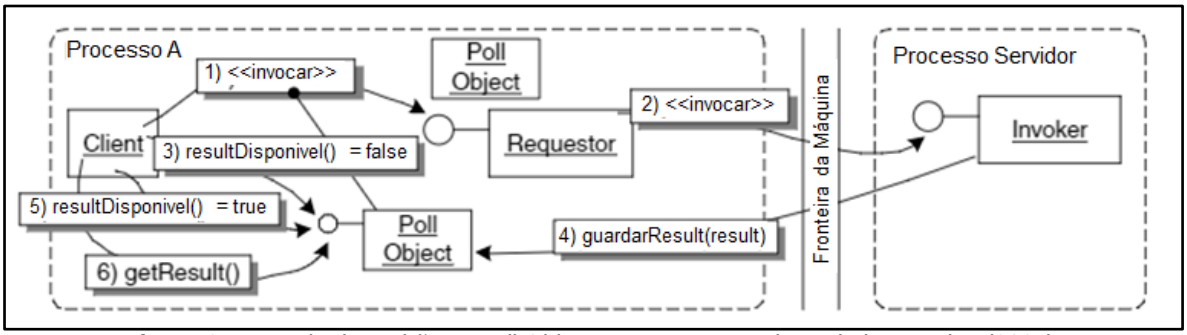

Figura 6: Exemplo de padrão, o *Poll Object*, proposto por Völter, Kircher e Zdun (2004).

O armazenamento é um repositório de informação para guardar os dados, podendo ser uma base de dados ou um arquivo (IEEE, 2000). Na visão de Sazonov e Neuman (2014), o armazenamento é importante para salvar os dados capturados dos sensores para o uso no processamento e seu avanço possibilita o aumento de dados armazenados no dispositivo vestível.

Mas o armazenamento pode ter diversos tratamentos de acesso aos dados (ORACLE, 2002), dependendo de fatores como o local do armazenamento, se está no dispositivo ou remotamente em servidor, e o seu tipo, se é uma base de dados ou arquivo. Para resolver esta problemática, é sugerido a implementação do padrão *Data Access Object*, ou DAO, Figura 7.

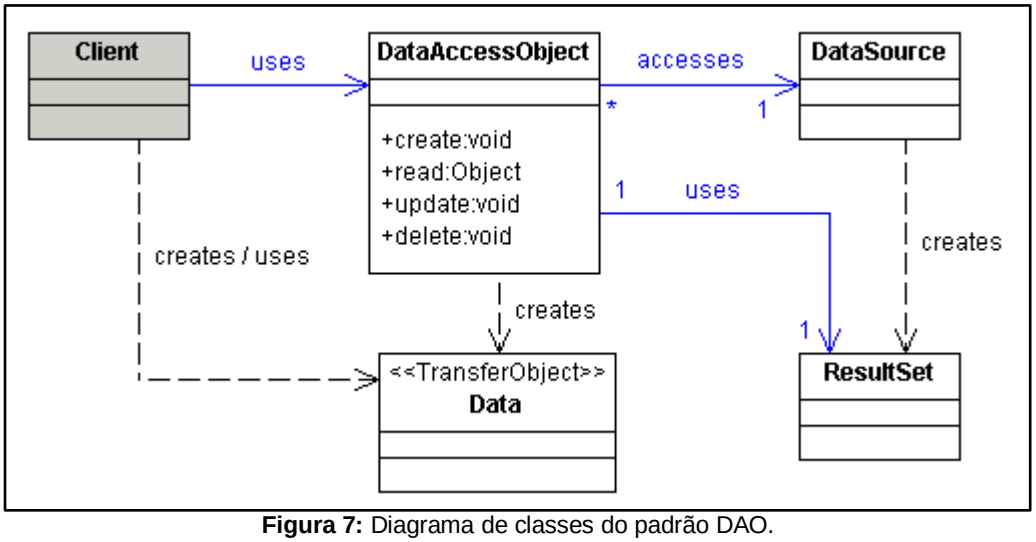

Figura 7: Diagrama de classes do padrão DAO.

Neste padrão, o acesso aos dados de uma base é feito através de um objeto. Este objeto é responsável por manipular os dados para envio e recebimento na base de dados. Desta forma, o cliente é capaz de processar os dados sem acessar a base de dados todo o tempo.

O processamento são técnicas que produzem uma saída para uma entrada, ou seja, processa uma informação através do recebimento de dados (RALSTON; REILLY; HEMMENDINGER, 2003). O aumento na capacidade de processamento, com a redução do tamanho dos processadores e a redução no consumo de energia, trouxe mais vantagens para um dispositivo vestível.

Porém, o acesso ao processamento da aplicação e o desenvolvimento de seus serviços podem ser desorganizados e concorridos, gerando graves problemas para a aplicação como um todo. O *Gang of Four* (GAMMA et al, 1994) falam de padronizações para reusos de códigos em programações orientadas a objetos, como exemplos abaixo.

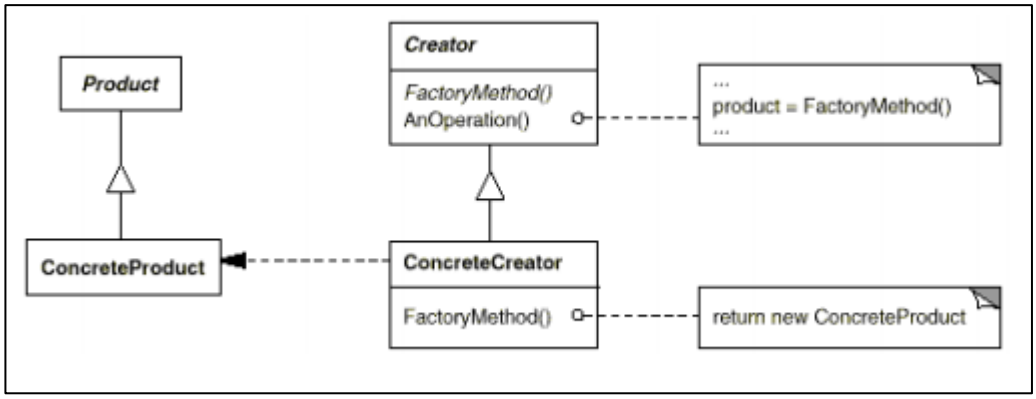

Figura 8: Diagrama de classe do padrão *Factory Method* do catálogo GoF.

O *Factory Method* tem como objetivo definir uma interface para criar objetos de forma a deixar subclasses decidirem qual classe instanciar. Neste padrão, a classe abstrata é a classe instanciada pelos demais objetos. As subclasses são as classes responsáveis por definir o produto. Todo o acesso aos produtos ocorrem pela classe abstrata.

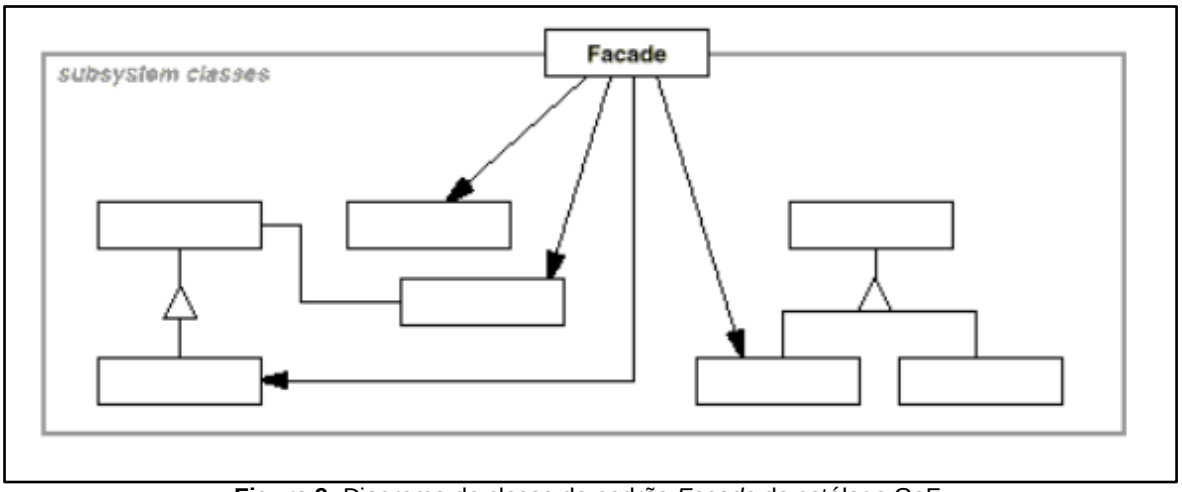

Figura 9: Diagrama de classe do padrão *Facade* do catálogo GoF.

O *Facade* é um padrão que cria uma interface de acesso aos subsistemas. Neste padrão, o cliente acessa apenas a classe *Facade* para chegar a qualquer outra classe que pertence ao seu conjunto de subsistemas. O seu uso ajuda na transparência dos acessos ao cliente.

Com os os padrões e problemas vistos acima, esta pesquisa visa o uso de padrões para o desenvolvimento de aplicações em Computação Vestível.

3. METODOLOGIA

Para o estudo do desenvolvimento em dispositivos vestíveis, foram feitos pesquisas aplicadas no assunto. Para isso, foram estudados diversos artigos para encontrar os principais problemas no desenvolvimento de uma aplicação e, consequentemente, necessite de um padrão para um melhor suporte.

Entretanto, foram encontradas poucas referências para esta área no levantamento bibliográfico. Para contornar este problema, foram elencados os cinco assuntos considerados importantes em um dispositivo vestível na visão de Sazonov e Neuman (2014): rastreamento dos sensores (*Tracking*), comunicação (*Communication*), processamento (*Processing*), armazenamento (*Storage*) e resposta (*Feedback*).

O rastreamento dos sensores e resposta tem um problema no desenvolvimento com a multimodalidade. Para resolver este problema, Ratzka (2008) sugere alguns padrões, mas todos de forma comportamental. Foi escolhido, então, o padrão *Context Adaptation* como base e buscou-se uma solução para o desenvolvimento no catálogo GoF que atendesse o padrão comportamental de Ratzka (2008). Foi encontrado o padrão *Strategy*, Figura 10. Além disso, devido a resposta não ser necessária em todos os dispositivos vestíveis, foram feitos padrões separados para cada um dos assuntos. Assim, quando não houver a necessidade da resposta, o padrão pode ser ignorado.

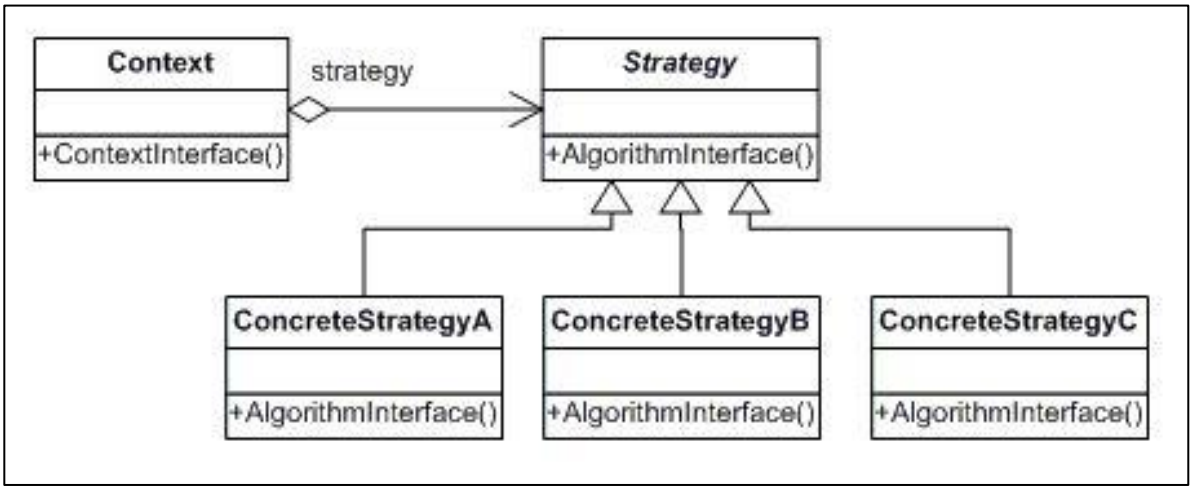

Figura 10: Diagrama de classes do padrão *Strategy*.

Na comunicação, foram encontrados problemas de dificuldade de manutenção do código, a falta de escalabilidade e transparência para o desenvolvimento em arquiteturas distribuídas neste assunto. Para atender a estas necessidades, Völter, Kircher e Zdun (2004) propuseram padrões como o *Marshaller*, *Requestor* e *Poll Object*. Entretanto, estes padrões são mais usados em uma aplicação cliente-servidor. Para adaptar os conceitos destes padrões para uso na tecnologia vestível, foi proposto o uso do padrão *Proxy*, Figura 11, do catálogo GoF com modificações nos seus elementos para utilizar conceitos importantes dos três padrões vistos. Assim, a comunicação pode ser utilizada tanto para os

componentes de comunicação conhecidos, como o *bluetooth* e *wi-fi*, quanto para a comunicação com os sensores.

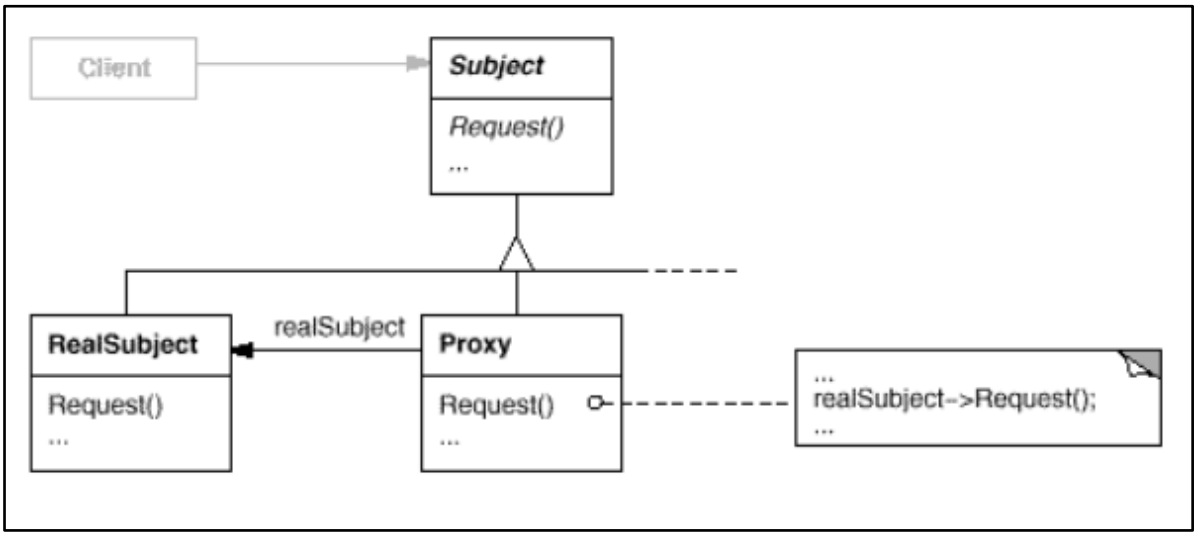

Figura 11: Diagrama de classes do padrão *Proxy*.

Para o armazenamento, foram encontrados problemas de diversidade de formas de acesso. A solução para este caso foi o uso do padrão DAO, conforme figura 7 (visto na seção 2), por ser um padrão conhecido e bem utilizado para problemas de armazenamento.

No processamento, foi notado que este assunto deve utilizar os padrões propostos anteriormente, além de ter que processar toda a aplicação em si. Assim, foi proposta uma pesquisa por padrões que possam manter a organização da aplicação para que o desenvolvedor possa entender facilmente seu contexto e, desta forma, foram escolhidos os padrões *Factory Method*, Figura 8 (visto na seção 2), e o *Facade*, Figura 9 (visto na seção 2), do catálogo GoF por manter uma boa indentação do código e a possibilidade de dividir o código através dos serviços com o qual a aplicação trabalha.

Ao final do desenvolvimento dos padrões, foi feita uma adaptação de uma aplicação existente para os padrões propostos. Assim, foi possível provar a implementação dos padrões estudados nesta pesquisa.

4. RESULTADOS E DISCUSSÃO

Buscando uma organização no desenvolvimento, foram propostos cinco padrões, que visto de forma conjunta na Figura 15 (próxima página), conseguem formar um único padrão conciso que abordam os principais assuntos para o desenvolvimento de uma aplicação para uma tecnologia vestível.

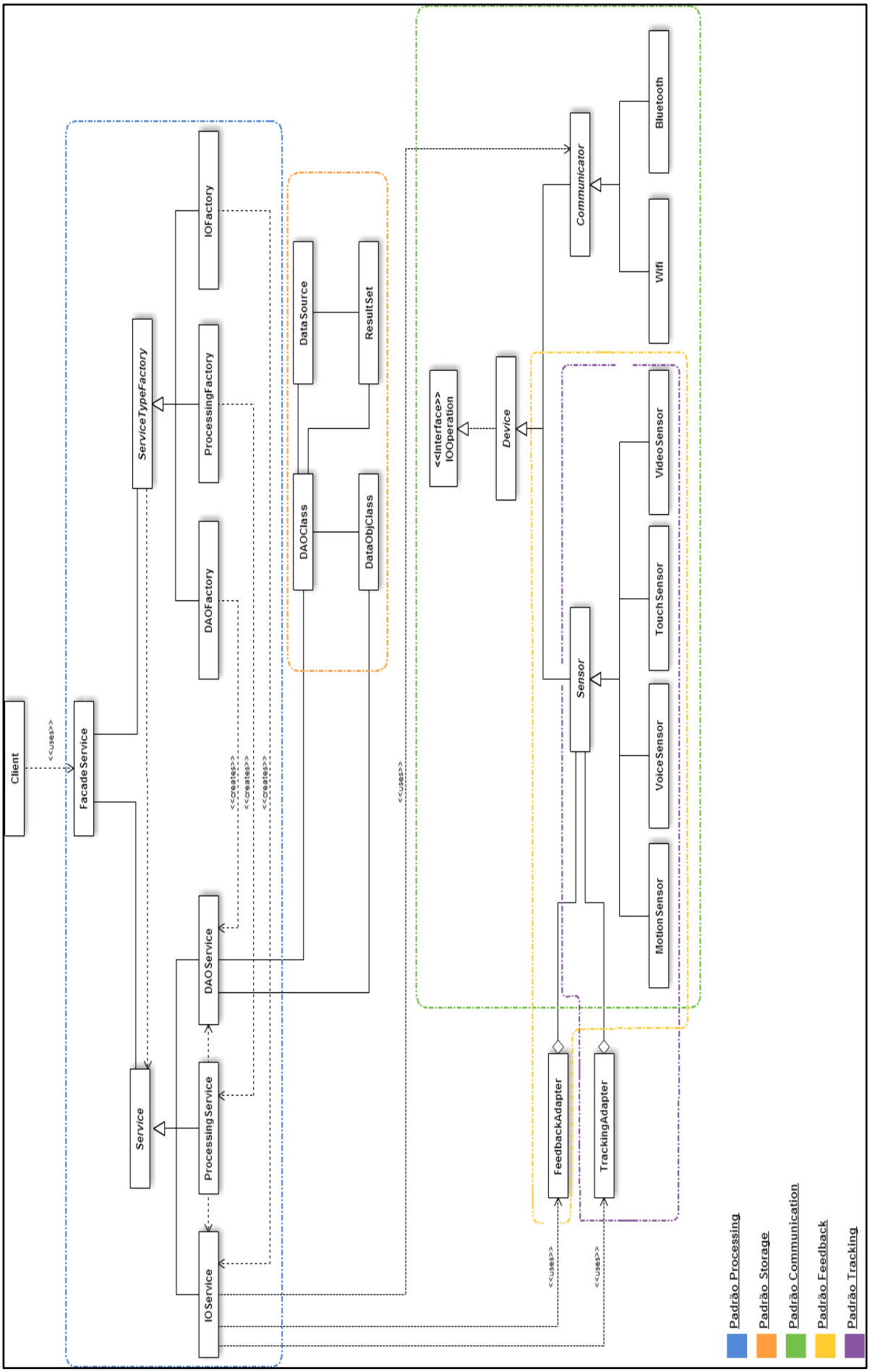

Figura 15: Visão dos padrões de projeto propostos

Em um fluxo de processo comum, a aplicação recebe os dados dos sensores através do padrão *Tracking*, detalhado na sessão 4.1 Tracking, ou através de outra comunicação provida pelo *Communication*, sessão 4.2 Communication, processa toda a aplicação no *Processing*, sessão 4.3 Processing, armazena os dados no *Storage*, sessão 4.4 Storage, e retorna os dados no *Feedback*, sessão 4.5 Feedback, ou novamente pelo *Communication*.

4.1. TRACKING

No rastreamento de sensores, foi notado que cada dispositivo físico vestível trabalha com um tipo de sensor, e cada tipo de sensor tem um tratamento diferente na programação. Para manter um controle e organização no recebimento dos dados do dispositivo físico, de forma transparente para a aplicação, foi criado um padrão, chamado de *Tracking*, que tem como funcionalidade o recebimento dos dados através de uma única classe, que usará as classes de cada tipo de sensor para transcrever a mensagem que deve ser enviada para processamento, como pode ser visto na Figura 16 (próxima página).

Este padrão é uma adaptação do padrão *Strategy* do catálogo GoF, onde o IOService é a classe cliente que utiliza a classe adaptadora TrackingAdapter, que por sua vez utiliza a classe adaptável abstrata Sensor para ler os dados das classes adaptáveis concretas MotionSensor, VoiceSensor, TouchSensor e VideoSensor. Cada classe adaptável tem a responsabilidade de ler os dados de cada tipo de sensor: movimento, voz, toque e vídeo. A classe TrackingAdapter deve ter um método read() para ler os dados providos da classe Sensor, que também deve ser implementado na classe devido a sua agregação. A classe Sensor deve ter um método abstrato read(), pois sua implementação deve ser feito pelas subclasses para que leiam os dados providos pelos dispositivos.

De modo geral, a classe que controla os dados de entrada e saída, IOService, faz a requisição à classe TrackingAdapter, que tem a responsabilidade de controlar o recebimento dos dados dos sensores e transcrever os dados recebidos para que seja utilizado no processamento da aplicação, acessada pela classe ProcessingService. Para o recebimento, é utilizado a classe abstrata Sensor, que generaliza as classes de cada tipo de sensor, que contém a leitura customizada para cada sensor.

O recebimento dos dados do dispositivo físico por uma classe adaptiva torna transparente a forma com que os dados são recebidos pela aplicação. Como consequência, há uma melhora na coordenação para a comunicação entre a parte física do dispositivo vestível e a aplicação, além da implementação dos diferentes tipos de sensores na forma de componentes e a possibilidade de extensão deste padrão para outros projetos de desenvolvimento no assunto.

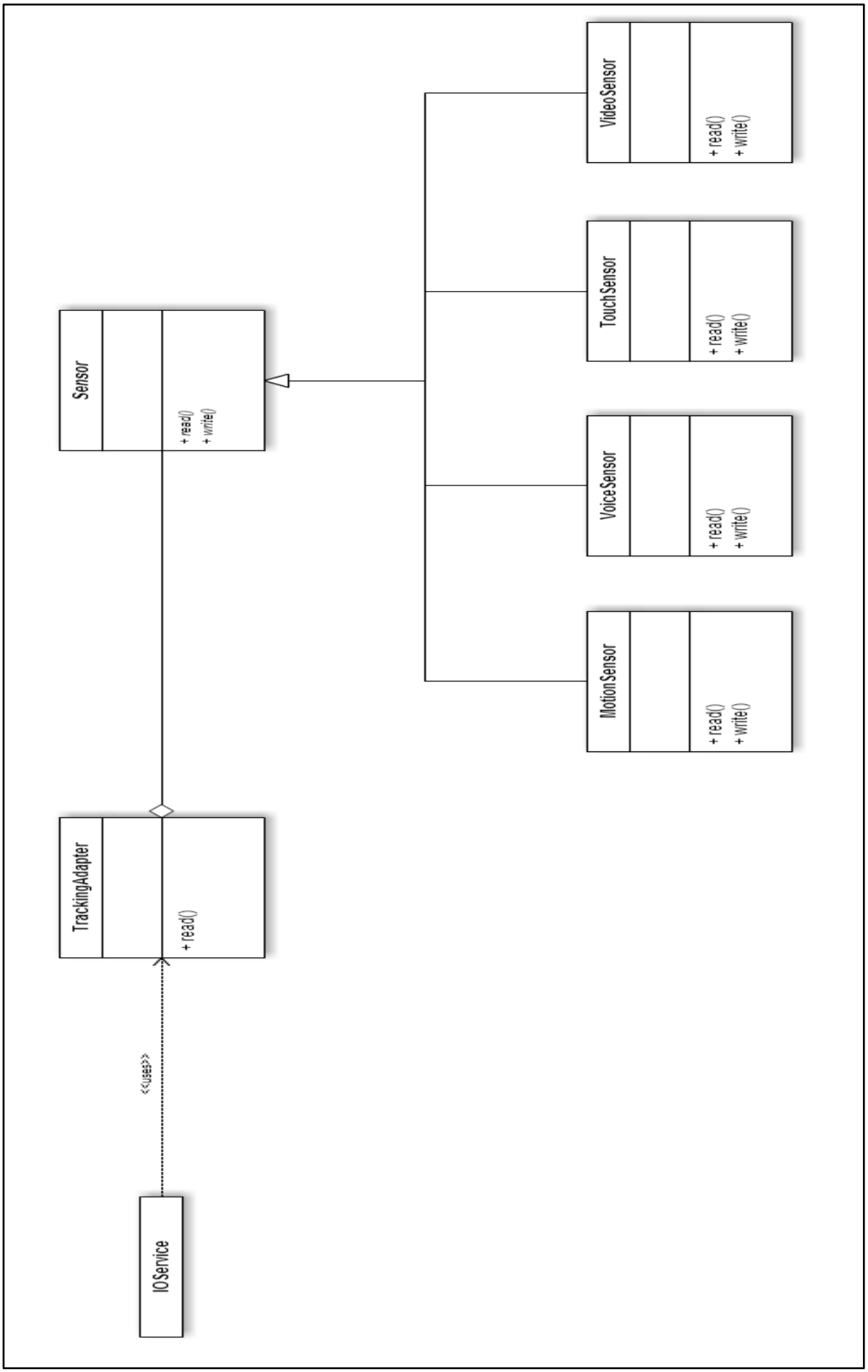

Figura 16: Diagrama de classes do padrão *Tracking*.

4.2. COMMUNICATION

O uso da comunicação é frequente, principalmente entre a aplicação e a parte física do dispositivo. Entretanto, há diversas formas de comunicação, que variam conforme o que o dispositivo pode oferecer. Além disso, o modo de comunicação pode ser síncrono, assíncrono ou ambos, dependendo da necessidade do aplicativo e do tipo de comunicação que será utilizado. Para cada comunicação que estabelecer, também deve ser pensado em como os dados estão entrando e como deverá ser sua saída.

Desta forma, este padrão, de nome *Communication*, visa tornar os processos de comunicação, seja síncrona ou assíncrona, de forma transparente, generalizando os mecanismos de comunicação em uma classe e abstraindo suas operações de leitura e escrita, conforme observado na Figura 17 (próxima página).

A construção deste padrão é baseada no *Proxy* do catálogo GoF, *Marshaller* e *Requestor* proposto por Völter, Kircher e Zdun. As classes abstratas Sensor e Communicator funcionam como a classe Proxy para os diversos tipos de sensores e comunicações sem fio, e as subclasses Wifi, Bluetooth, MotionSensor, VoiceSensor, TouchSensor e VideoSensor fazem o *marshalling* e a requisição para os dispositivos. Para isso, elas devem implementar os métodos read() e write(), onde a interface IOOperation obriga essa implementação, e implementar as serializações e requisições nelas.

Neste padrão, a classe abstrata Device generaliza as classes para cada tipo de comunicação, além de utilizar a interface IOOperation que contém os métodos que obrigatoriamente cada classe que deriva da Device deve utilizar. Desta forma, a classe que controla o serviço de entrada e saída IOService obtém o acesso a qualquer tipo de comunicação através da superclasse Communication. A classe Sensor é acessada de forma diferenciada pois as subclasses têm outros tratamentos que a classe Communication não tem necessidade.

O uso deste padrão ajuda na coordenação da comunicação entre os dispositivos de comunicação e a aplicação, simplificando as comunicações síncronas e assíncronas e ajudando na possibilidade de estender o código em outros projetos sobre o assunto.

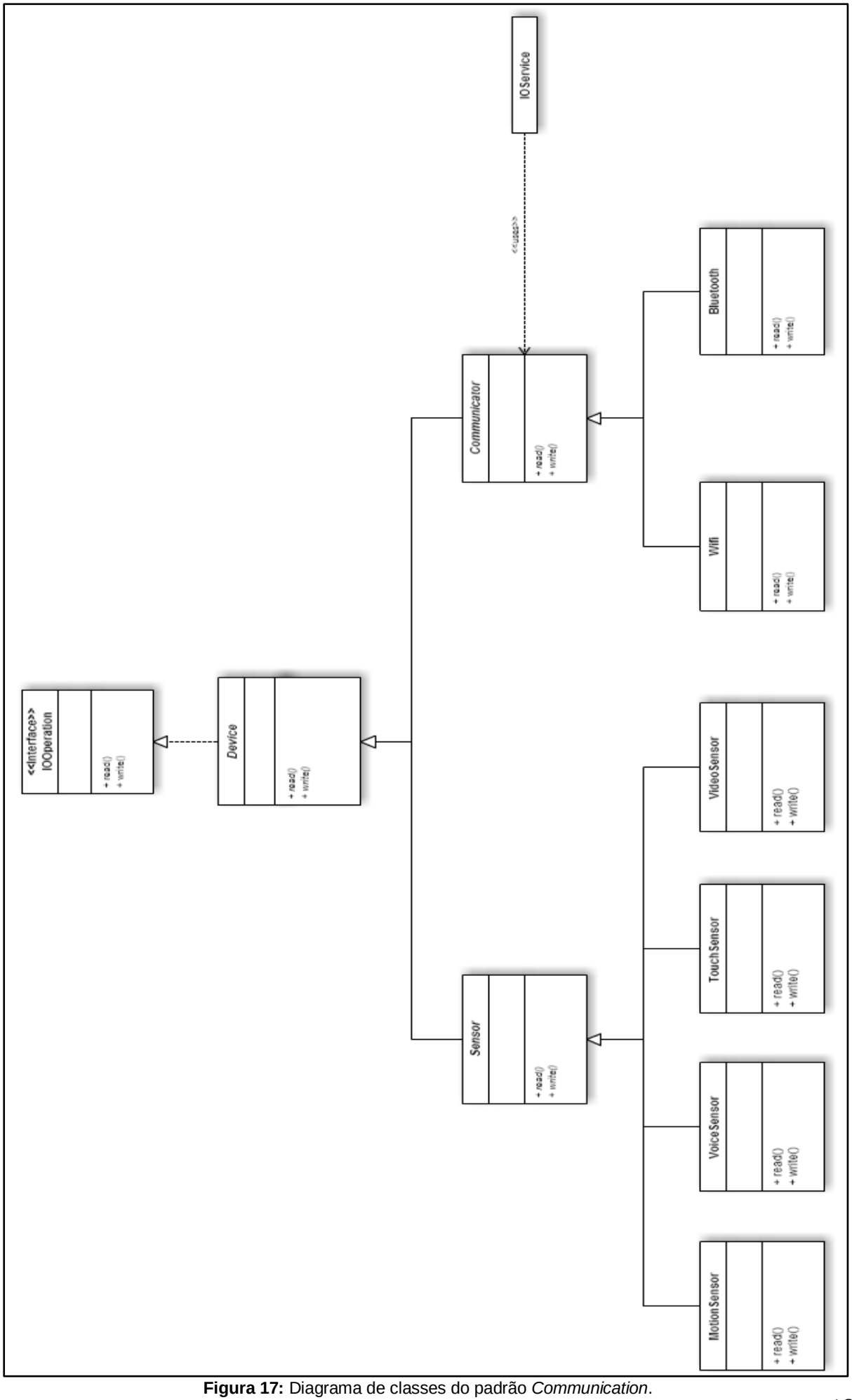

Figura 17: Diagrama de classes do padrão *Communication*.

4.3. PROCESSING

A solução para o processamento foi organizar os serviços que a aplicação necessita para criar uma modularização de seus processos através de um acesso único para os mesmos, ajudando a organizar o desenvolvimento e possibilitando a extensão do código, conforme pode ser observado na Figura 18 o padrão *Processing*.

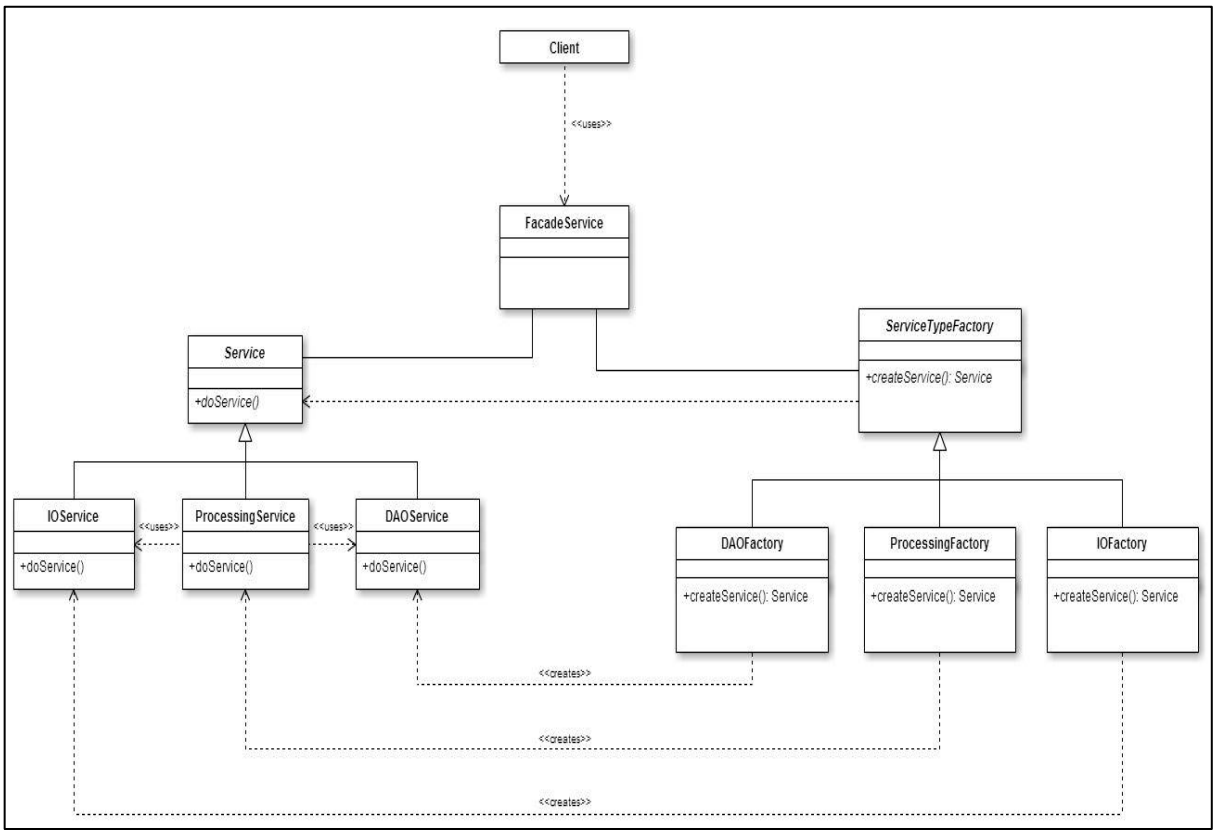

Figura 18: Diagrama de classes do padrão *Processing*.

O padrão se baseia nos padrões *Façade* e *Factory Method* do catálogo GoF. A classe FacadeService assume o papel da classe Facade, sendo o único ponto de acesso para controlar todos os serviços da aplicação. A ServiceTypeFactory e a Service são as classes abstratas para a criação das classes de fábricas de produto e produtos, respectivamente, e a fábrica de serviços ServiceTypeFactory depende da classe Service para a criação de um serviço, através do método createService(). Desta forma, as subclasses DAOFactory, ProcessingFactory e IOFactory conseguem criar as subclasses da classe Service: DAOService, ProcessingService e IOService. Para que o FacadeService consiga acessar as ações de cada serviço, é necessário que a classe Service tenha o método doService() e extenda para as subclasses. Além disso, há uma dependência da ProcessingService com a IOService e DAOService para que o processamento consiga armazenar os dados e enviar e receber os dados providos pelas interfaces de entrada e saída.

No processo, as requisições a serem feitas à aplicação devem ser feitas pela FacadeService, que controla as chamadas para as subclasses de serviços necessárias para

atender a solicitação. Caso não haja um serviço, a FacadeService também controla a fábrica de serviços para que se crie o serviço com os processos necessários.

Este padrão propõe prover uma organização e controle nos processos comuns aos aplicativos vestíveis. Assim, os serviços necessários para a aplicação são tratados como componentes no desenvolvimento e seu processamento se torna transparente nas chamadas do cliente.

4.4. STORAGE

Para o armazenamento, a solução foi criar uma camada para controlar as operações básicas: inserção, leitura, atualização e remoção, tanto para os armazenamentos no dispositivo quanto para os armazenamentos externos. Para que a aplicação possa manipular corretamente os dados, sem que haja conflitos no armazenamento e não dependa do tipo do armazenamento, foi proposto o padrão *Storage*, com a sua estrutura conforme a Figura 19.

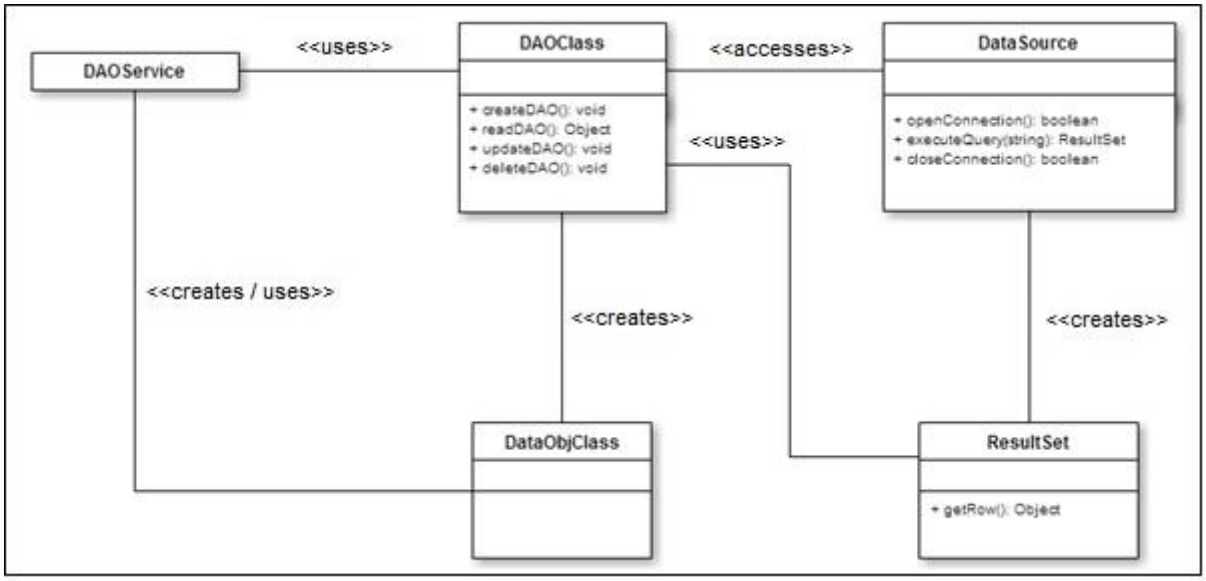

Figura 19: Diagrama de classes do padrão *Storage*.

O padrão se baseia no padrão *Data Access Object* (DAO). A classe DAOClass é responsável pela interface de controle dos dados entre o serviço e a conexão do local do armazenamento. Nela há os métodos básicos de operação: createDAO() para inserir, readDAO() para ler, updateDAO() para atualizar e deleteDAO() para excluir os dados do armazenamento. A classe DataSource faz a conexão com o armazenamento, através dos métodos openConnection() e closeConnection(), e realiza a operação da DAOClass com o método executeQuery(). No caso de uma leitura, o retorno dos dados são temporariamente armazenados na classe ResultSet até que se faça uma nova consulta. A classe DataObjClass receberá os dados providos pela ResultSet para que a DAOService utilize os dados recebidos.

Além de organizar e controlar o desenvolvimento para o armazenamento, é possível também expandir seu uso, caso o dispositivo vestível tenha maior capacidade de armazenamento interno.

4.5. FEEDBACK

Assim como o padrão visto na seção 4.1. Tracking, a resposta ao sensor também busca controlar a comunicação entre os diversos tipos de sensores de forma transparente. Porém, a resposta busca enviar os dados ao invés de receber. Por isso, para controlar o fluxo de saída dos dados para o sensor e segregar o fluxo para melhor organização, foi proposto o padrão *Feedback*, conforme a Figura 20 mostra sua estrutura.

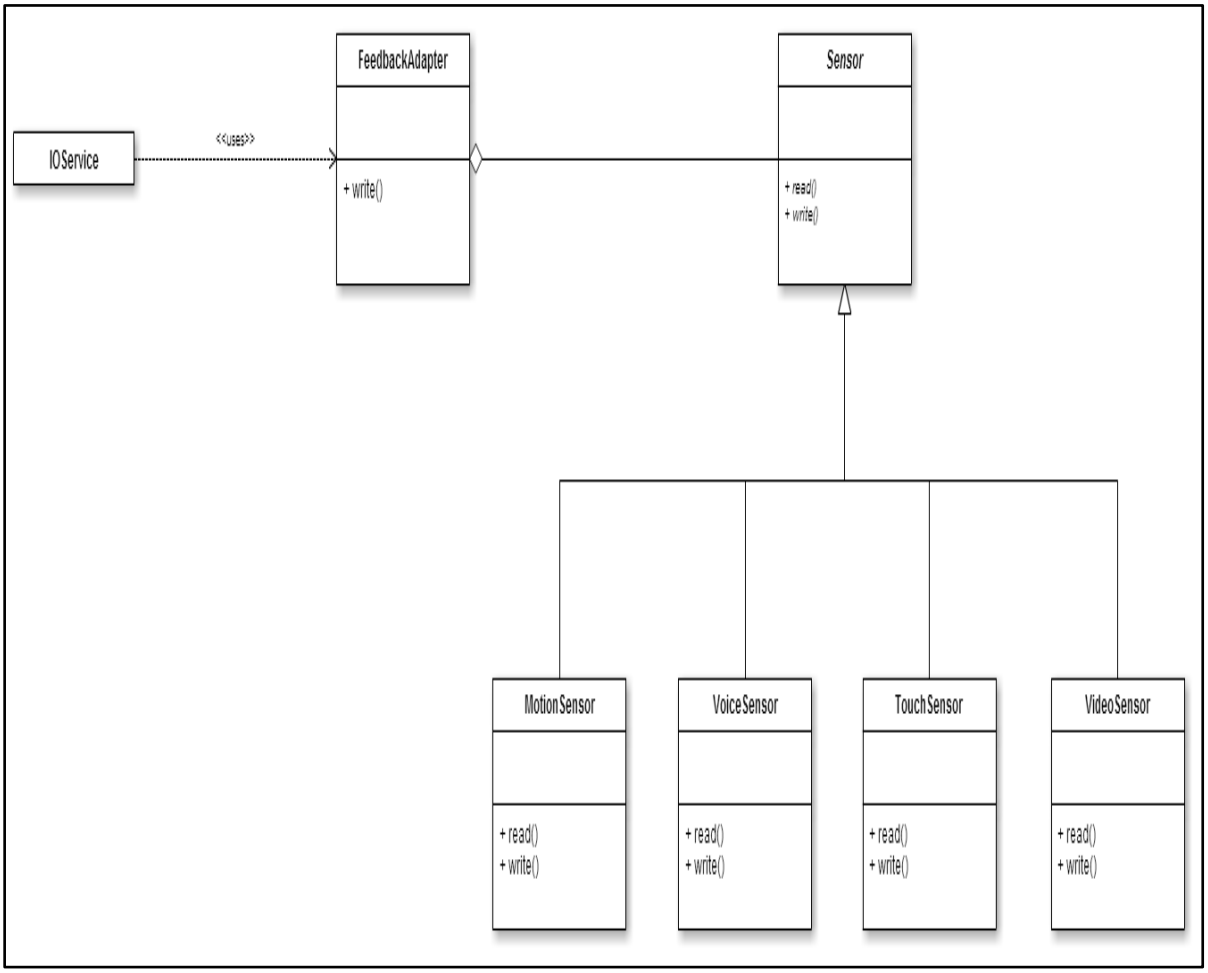

Figura 20: Diagrama de classes do padrão *Feedback*.

Este padrão também é uma adaptação do padrão *Strategy* do catálogo GoF, onde o IOService é a classe cliente que utiliza a classe adaptadora FeedbackAdapter, que por sua vez utiliza a classe adaptável abstrata Sensor para enviar os dados das classes adaptáveis concretas MotionSensor, VoiceSensor, TouchSensor e VideoSensor. Cada classe adaptável tem a responsabilidade de serializar os dados para envio em cada tipo de sensor: movimento, voz, toque e vídeo. A classe FeedbackAdapter deve ter um método write() para enviar os dados providos da classe IOService. A classe Sensor deve ter um método abstrato

write(), pois sua implementação deve ser feito pelas subclasses para que serializem e enviem os dados para os dispositivos.

De modo geral, a classe que controla os dados de entrada e saída, IOService, faz a requisição à classe FeedbackAdapter, que tem a responsabilidade de controlar o envio dos dados dos sensores. Para o envio, é utilizado a classe abstrata Sensor, que generaliza as classes de cada tipo de sensor, que faz a serialização dos dados para o envio e a leitura correta em cada sensor.

O envio dos dados para o dispositivo físico por uma classe adaptiva torna transparente a forma com que os dados são enviados pela aplicação. Como consequência, há uma melhora na coordenação para a comunicação entre a parte física do dispositivo vestível e a aplicação, além da implementação dos diferentes tipos de sensores na forma de componentes e a possibilidade de extensão deste padrão para outros projetos de desenvolvimento no assunto.

5. IMPLEMENTAÇÃO DOS PATTERNS

Para testar os padrões propostos, foi adaptado o jogo de demonstração *Angry Bots*, da ferramenta de desenvolvimento de jogos Unity. Foram usados como dispositivos vestíveis o Oculus Rift, equipamento de sistema visual em realidade virtual, e o Leap Motion, equipamento para captação dos movimentos das mãos.

Neste jogo de tiro foi analisado o funcionamento do personagem principal, a adaptação para o uso dos padrões em seu código e as restrições da ferramenta para esta implementação. A principal restrição encontrada foi o impedimento do uso de heranças múltiplas, que impossibilitou o uso das heranças previstas nos padrões juntamente com a herança da classe que possibilita o acesso aos objetos do jogo, a classe MonoBehaviour.

A solução encontrada para implementar os padrões propostos foi a separação de lógicas na programação que compõem o jogo em classes que herdam atributos da MonoBehaviour e a invocação destas classes para que sejam instanciadas dentro das classes dos padrões propostos, observado no diagrama da Figura 21 (próxima página).

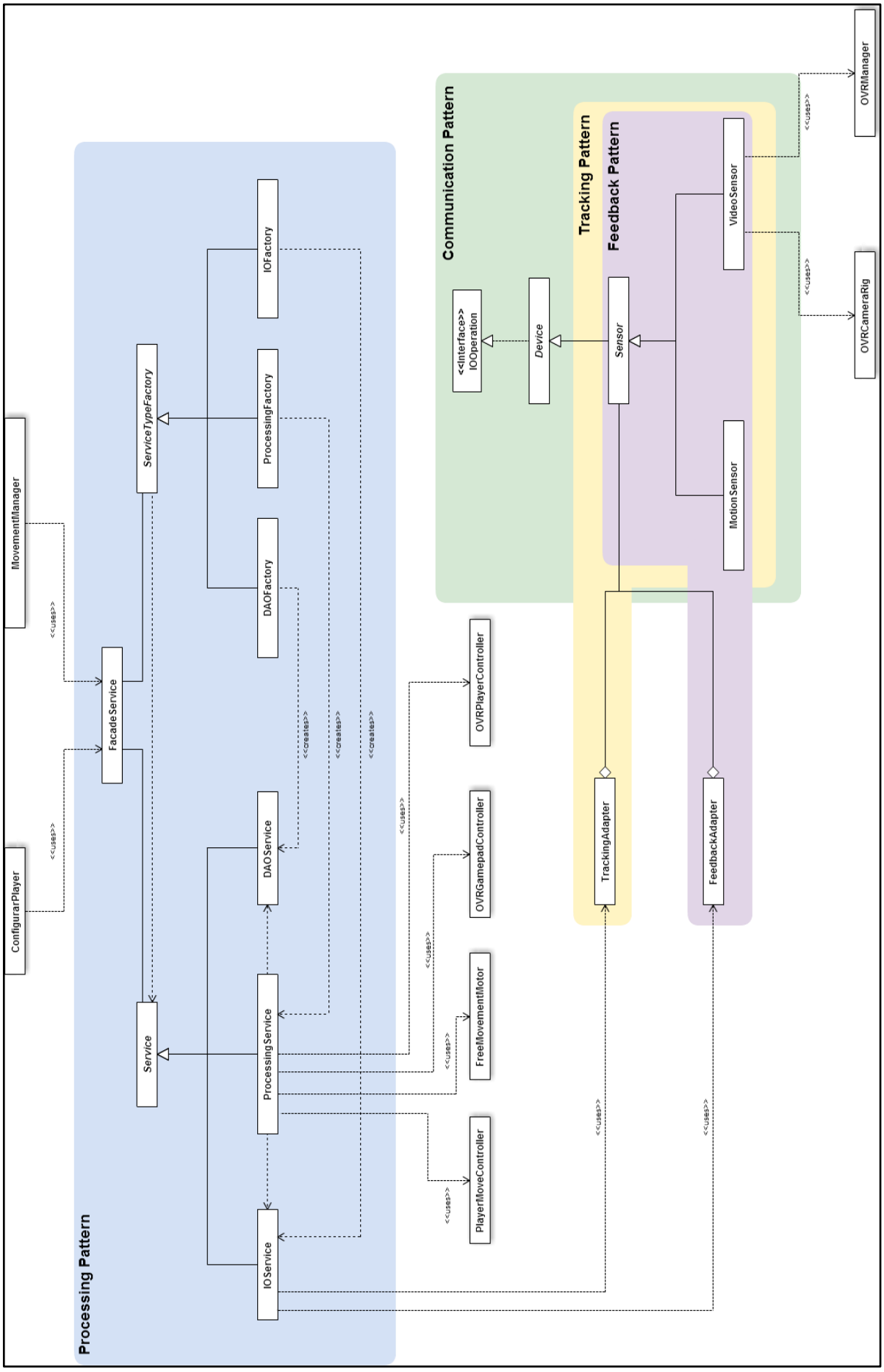

Figura 21: Diagrama de classes da implementação dos padrões no jogo Angry Bots.

No diagrama, as classes que não pertencem a um padrão são as classes que foram criadas para a implementação da lógica do jogador nas restrições do Unity. As funções Awake(), Start() e Update() são exemplos de métodos que devem ser implementados para que o personagem tenha suas ações feitas pela ferramenta. É o caso das classes ConfigurarPlayer e MovementManager, nos Códigos 1 e 2 respectivamente.

```
1 //--------------------------------------------
2 // Classe para instanciar o pattern no Unity
3 //--------------------------------------------
4 using System;
5 using UnityEngine;
6
7 public class ConfigurarPlayer : MonoBehaviour
8 {
9     void Awake() {
10         FacadeService fs = new FacadeService();
11         fs.camera();
12         fs.movimentoPlayer();
13     }
14
15 }
```

Código 1: Trecho do código da classe ConfigurarPlayer.

A classe ConfigurarPlayer foi instanciada para o objeto do personagem do jogo. Quando o jogo está carregando, antes de iniciar, o objeto executa a função Awake(), instanciando a classe FacadeService e seus métodos camera() e movimentoPlayer() para enviar o jogo para a saída do vídeo para o Oculus Rift e controlar a movimentação do jogador.

```
1 //--------------------------------------------
2 // Classe para instanciar o pattern no Unity
3 //--------------------------------------------
4 using UnityEngine;
5 using System.Collections;
6
7 public class MovementManager : MonoBehaviour {
8   public GameObject leapMotionOVRController = null;
9   public HandController handController = null;
10
11    public FacadeService fs;
12    // Use this for initialization
13    void Start () {
14        fs = new FacadeService();
15    }
16
17    // Update is called once per frame
18    void Update () {
19        fs.controlHands ();
20
21    }
22 }
```

Código 2: Trecho do código da classe MovementManager.

A classe MovementManager foi instanciada para o objeto do personagem do jogo. Quando o jogo inicia, o objeto instancia a classe FacadeService, e nas atualizações quadro a quadro do personagem é invocado o método controlHands() para buscar o movimento da mão no Leap Motion e enviá-lo para o processamento da movimentação do jogador.

Para as classes previstas nos padrões, o padrão Storage e as classes Wifi e Bluetooth não puderam ser testados, pois o jogo não tinha a necessidade de uso de um armazenamento e uma comunicação em Wifi e Bluetooth. Para os demais, o fluxo se seguiu da forma como esperado, como visto nos códigos abaixo.

Como previsto no padrão Processing, visto no Código 4, a FacadeService instancia as classes IOFactory, ProcessingFactory e DAOFactory através de um atributo da classe ServiceTypeFactory, assim como a IOService, ProcessingService e DAOService são instanciados na Service.

```
1 //------------------------------------------------------------------------------
2 // Classe: FacadeService
3 // Pattern: Processing
4 //------------------------------------------------------------------------------
5 using System;
6 using UnityEngine;
7
8 public class FacadeService
9 {
10     //variaveis da abstract factory que a Facade acessa
11     private ServiceTypeFactory ioFactory;
12     private ServiceTypeFactory processingFactory;
13     private ServiceTypeFactory daoFactory;
14
15     private Service ioService;
16     private Service processingService;
17     private Service daoService;
18
19     public FacadeService() //Abstract Factory é criado junto com a Facade
20     {
21         this.ioFactory = new IOFactory();
22         this.processingFactory = new ProcessingFactory();
23         this.daoFactory = new DAOFactory();
24
25         this.ioService = ioFactory.createService();
26         this.processingService = processingFactory.createService();
27         this.daoService = daoFactory.createService();
28     }
29
30     public void controlHands() //Método para controle e processamento das mãos
31     {
32         processingService.doService((IOService) ioService, 1);
33     }
34
35     public void movimentoPlayer() //Método para processamento do movimento do player
36     {
37         processingService.doService(1);
38     }
39
40     public void camera() //Método para enviar dados para o Oculus Rift
41     {
42         processingService.doService((IOService) ioService, 2);
43     }
44 }
```

Código 3: Código fonte da classe FacadeService.

No Código 4, é notado que a classe ProcessingFactory é uma subclasse da superclasse ServiceTypeFactory, e sobrescreve o método createService(), criando a classe ProcessingService. As demais fábricas também seguem o mesmo exemplo.

```
1 //-----------------------------------------------------
2 // Classe: ProcessingFactory
3 // Pattern: Processing
4 //-----------------------------------------------------
5 using System;
6
7 public class ProcessingFactory : ServiceTypeFactory
8 {
9     public override Service createService() {
10         return new ProcessingService();
11     }
12 }
```

Código 4: Trecho do código da classe ProcessingFactory.

. A ProcessingService, como observado no Código 5, é uma subclasse da Service, e sobrescreve a classe doService(), tendo um parâmetro de opção para escolher o serviço a ser executado. A mesma lógica se aplica às classes DAOService e IOService.

```
1 //-----------------------------------------------------------------------------
2 // Classe: ProcessingService
3 // Pattern: Processing
4 //-----------------------------------------------------------------------------
5 using System;
6 using System.Collections;
7 using UnityEngine;
8
9 public class ProcessingService : Service
10 {
11     //Variaveis
12     private static GameObject playerObj = GameObject.Find("Player");
13     private static HandController handController = playerObj.GetComponentInChildren<HandController>();
14
15     public override void doService(int opcao) {
16         switch (opcao) {
17         case 1: //Inclui os processamentos do movimento do Player e o movimento do Oculus Rift
18
19             FreeMovementMotor fmm = playerObj.AddComponent<FreeMovementMotor>();
20             PlayerMoveController pmc = playerObj.AddComponent<PlayerMoveController>();
21
22             //OVRMainMenu omm = playerObj.AddComponent<OVRMainMenu>();
23             OVRGamepadController ogc = playerObj.AddComponent<OVRGamepadController>();
24             OVRPlayerController opc = playerObj.AddComponent<OVRPlayerController>();
25             break;
26         default:
27             Debug.LogWarning("Invalid option in ProcessingService.");
28             break;
29         }
30     }
```

Código 5: Trecho do código da classe ProcessingService.

A classe TrackingAdapter, Código 6, segue o modelo do padrão, tendo o método read() para ler os dados dos sensores e enviar o retorno para a IOService. No método read, há também o uso da classe Sensor para acessar suas subclasses, neste caso é acessado a MotionSensor para ler o movimento das mãos no Leap Motion.

```
1 //------------------------------------------------------------------------
2 // Classe: TrackingAdapter
3 // Pattern: Tracking
4 //------------------------------------------------------------------------
5 using System;
6 using UnityEngine;
7
8 public class TrackingAdapter
9 {
10     public object read(int opt)
11     {
12         switch (opt) {
13         case 1: //ler as maos
14             Sensor motionSensor = new MotionSensor(); //Instancia da MotionSensor a partir da classe abstrata Sensor
15             return motionSensor.readHands();
16             //break;
17         default:
18             Debug.LogWarning("Invalid option in TrackingAdapter.");
19             break;
20         }
21         return null;
22     }
23 }
```

Código 6: Trecho do código da classe TrackingAdapter.

A classe FeedbackAdapter, Código 7, segue o modelo do padrão, tendo o método write() para enviar os dados para os sensores. Neste método também há uma Sensor instanciada com uma subclasse, neste exemplo visto é acessado a VideoSensor para enviar as imagens para o Oculus Rift.

```
1 //------------------------------------------------------------------------
2 // Classe: FeedbackAdapter
3 // Pattern: Feedback
4 //------------------------------------------------------------------------
5 using System;
6 using UnityEngine;
7
8 public class FeedbackAdapter
9 {
10     public void write() {
11         Sensor videoSensor = new VideoSensor (); //Instancia do VideoSensor a partir da classe abstrata Sensor
12         videoSensor.write();
13     }
14 }
```

Código 7: Trecho do código da classe FeedbackAdapter.

Como visto na Código 8, a classe MotionSensor é uma subclasse da Sensor e implementa os métodos read() e write() em sua classe. Neste exemplo, o read() é implementado na readHands() para retornar as mãos lidas no Leap Motion para a TrackingAdapter. Todas as demais subclasses da Sensor seguem o mesmo modelo da MotionSensor.

```
1 //------------------------------------------------------------------------------
2 // Classe: MotionSensor
3 // Pattern: Communication
4 //------------------------------------------------------------------------------
5 using System;
6 using UnityEngine;
7
8 public class MotionSensor : Sensor
9 {
10     private static HandController handC;
11
12     public override object read() {
13         return null;
14     }
15
16     public override HandModel[] readHands() {
17         handC = GameObject.Find("Player").GetComponentInChildren<HandController>();
18         return handC.GetAllGraphicsHands ();
19     }
20
21     public override void write() {
22
23     }
24
25 }
```

Código 8: Código fonte da classe MotionSensor.

5.1. COMENTÁRIOS FINAIS DA APLICAÇÃO E IMPLEMENTAÇÃO DOS PADRÕES

Com a implementação completa, conforme visto na Figura 22, notou-se que as alterações feitas para a adaptação do jogo no Oculus Rift e Leap Motion foram simplificadas com o uso dos padrões e foi mantida uma maior organização no código. O maior benefício na sua implementação foi separar o código em assuntos e competências. Comparado ao jogo original, que mantinha todas as ações do jogador e movimentação das câmeras em quatro classes, os padrões trouxeram um entendimento claro de onde estava localizado cada lógica implementada para o personagem do jogo.

Figura 22: Testes da adaptação do jogo *Angry Bots* com os padrões propostos.

Para facilitar mais as consultas, as classes foram separadas nas pastas de seus respectivos padrões, conforme observado na Figura 23. Entretanto, houve obstáculos para uma implementação total dos padrões devido as restrições de desenvolvimento na ferramenta escolhida.

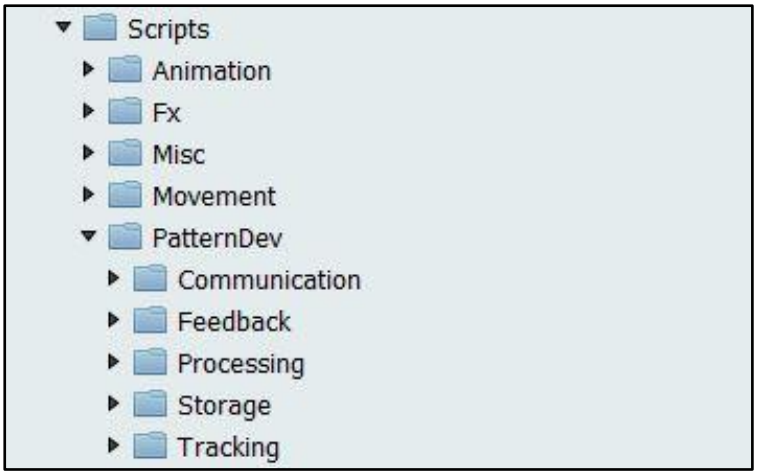

Figura 23: Pastas criadas nos Assets do Unity.

6. CONCLUSÃO E TRABALHOS FUTUROS

O estudo visou a mostrar a importância da aplicação de um padrão de projeto nos desenvolvimentos em tecnologias vestíveis, estudando a arquitetura comum presente nos dispositivos vestíveis, propondo um padrão de projeto a partir desta estrutura e implementando-os numa aplicação de um jogo com código aberto.

Os padrões de projeto propostos foram, em sua maioria, uma posição satisfatória na sua aplicabilidade. Foi possível obter os benefícios de sua proposição na implementação, garantindo uma organização nas funcionalidades da aplicação. Os testes nos dispositivos não tiveram dificuldades em transmitir e receber as informações da aplicação e foi obtido uma organização conforme previsto nos padrões.

Contudo, o jogo escolhido para adaptar o desenvolvimento não previa o uso de todos os padrões propostos. Além disso, a ferramenta escolhida para o desenvolvimento, o Unity, tem uma forma própria para o funcionamento da linguagem de programação, limitando o uso do padrão da forma esperada e obrigando a ajustar algumas funções.

O desenvolvimento de uma aplicação que utilize os padrões de projeto que não foram possíveis de testar ou o uso de todos os padrões propostos, e que seja desenvolvido preferencialmente em outra ferramenta de desenvolvimento, pode ser feito futuramente para estudar seus benefícios nos demais dispositivos vestíveis.

Também pode ser feito, em um trabalho futuro, o teste dos padrões deste trabalho em outras aplicações, ou até novas proposições de padrões de projeto para a Computação Vestível, que busque uma abrangência maior de dispositivos vestíveis e suas peculiaridades no desenvolvimento da aplicação.

REFERÊNCIAS BIBLIOGRÁFICAS

GAMMA, Erich et al. *Design Patterns*: Elements of Reusable Object-Oriented Software. EUA: Addison-Wesley. 1994.

IEEE. *IEEE 100*: The Authoritative Dictionary of IEEE Standards Terms. 7. ed. EUA: IEEE Press, 2000.

MANN, Steve. Wearable Computing. In: SOEGAARD, Mads; DAM, Rikke Friis. *The Encyclopedia of Human-Computer Interaction*, 2. ed. Dinamarca: The Interaction Design Foundation, 2014. Disponível em: <https://www.interaction-design.org/encyclopedia/wearable_computing.html>. Acesso em: 25 abr 2015.

ORACLE, Inc. *Core J2EE Patterns* - Data Access Object. EUA. 2002. Disponível em: <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html>. Acesso em: 7 dez 2014.

PAHVALAN, Kaveh; LEVESQUE, Allen H. *Wireless Information Networks*. 2. ed. EUA: Wiley, 2005.

RALSTON, Anthony; REILLY, Edwin D.; HEMMENDINGER, David. *Encyclopedia of Computer Science.* 4. ed. EUA: Wiley, 2003.

RATZKA, Andreas. *Patterns for Robust and Flexible Multimodal Interactions*. Alemanha. 2008. Disponível em: <http://ceur-ws.org/Vol-610/paper14.pdf?origin=publication_detail>. Acesso em: 6 dez 2014.

SAZONOV, Edward; NEUMAN, Michael R. *Wearable Sensors*: Fundamentals, Implementation and Applications. EUA: Elsevier. 2014.

VÖLTER, Markus; KIRCHER, Michael; ZDUN, Uwe. *Remoting Patterns*: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware. Inglaterra: John Wiley & Sons, Ltd. 2004.