Bruno Vitorelli Tacca de Oliveira

INTRACS - Sistema em código aberto para coleta e processamento de dados inerciais com protótipo multisensor sem fio de baixo custo e customização em métodos de processamento

Presidente Prudente

2022

Bruno Vitorelli Tacca de Oliveira

INTRACS - Sistema em código aberto para coleta e processamento de dados inerciais com protótipo multisensor sem fio de baixo custo e customização em métodos de processamento

Dissertação apresentada como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação – Área de concentração em Computação Aplicada, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de Presidente Prudente.

Universidade Estadual Paulista "Júlio de Mesquita Filho" – UNESP

Instituto de Biociências, Letras e Ciências Exatas

Programa de Pós-Graduação em Ciência da Computação

Orientador: Prof. Dr. Danilo Medeiros Eler

Presidente Prudente

2022

O48i

Oliveira, Bruno Vitorelli Tacca
INTRACS - Sistema em código aberto para coleta e processamento de dados inerciais com protótipo multisensor sem fio de baixo custo e customização em métodos de processamento / Bruno Vitorelli Tacca Oliveira. -- Presidente Prudente, 2022
99 p. : il., tabs.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Faculdade de Ciências e Tecnologia, Presidente Prudente
Orientador: Danilo Medeiros Eler

1. Arquitetura limpa. 2. Padrões de software. 3. Internet das coisas. 4. Sensores inerciais. 5. Software gratuito. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Faculdade de Ciências e Tecnologia, Presidente Prudente. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

UNIVERSIDADE ESTADUAL PAULISTA

Câmpus de Presidente Prudente

CERTIFICADO DE APROVAÇÃO

TÍTULO DA DISSERTAÇÃO: INTRACS - Sistema em código aberto para coleta e processamento de dados inerciais com protótipo multisensor sem fio de baixo custo e customização em métodos de processamento

AUTOR: BRUNO VITORELLI TACCA DE OLIVEIRA
ORIENTADOR: DANILO MEDEIROS ELER

Aprovado como parte das exigências para obtenção do Título de Mestre em Ciência da Computação, área: Computação Aplicada pela Comissão Examinadora:

Prof. Dr. DANILO MEDEIROS ELER (Participaçao Virtual)
Departamento de Matemática e Computação / Faculdade de Ciências e Tecnologia de Presidente Prudente

Prof. Dr. VANDER LUÍS DE SOUZA FREITAS (Participaçao Virtual)
Departamento de Computação / Universidade Federal de Ouro Preto

Prof. Dr. ALMIR OLIVETTE ARTERO (Participaçao Virtual)
Departamento de Matemática e Computação / Faculdade de Ciências e Tecnologia de Presidente Prudente

Presidente Prudente, 07 de março de 2022

À minha família e a todos que me apoiaram durante essa jornada.

Agradecimentos

Agradeço primeiramente à minha família, em especial à minha esposa Renata que sempre esteve ao meu lado quando precisei, que me fez não querer desistir quando fraquejei, e que constantemente cuidou de mim, de nós, de nossas filhas e de nosso lar. Vocês são minha base, o chão que me permite levantar, meu presente e meu futuro. Muito obrigado.

Agradeço também a meus amigos por sempre me lembrar que: "*Já começou, agora só terminar.*"; "*Já está acabando, só mais alguns meses.*" e; "*Importante, e o churrasco?*". Muito obrigado pelo apoio, pelas risadas, e por me ajudar a clarear minha visão, que muitas vezes ficou embaçada.

Agradeço ao professor Danilo Medeiros Eler, pela oportunidade, orientação e agilidade durante todo o desenvolvimento dessa pesquisa. Agradeço também à UNESP, por estar presente em toda minha formação acadêmica, muito da minha trajetória profissional e, ainda assim, por me proporcionar a oportunidade de trilhar essa jornada acadêmica.

Por fim, agradeço a todos que, de alguma forma, estiveram comigo e contribuíram durante essa etapa de minha vida.

"The search for truth is more precious than its possession."

— Albert Einstein

Resumo

O avanço tecnológico permite pesquisa e inovação em diversas áreas. Com a redução do tamanho dos componentes os dispositivos estão cada vez menores, favorecendo a portabilidade de protótipos e produtos. Sensores e comunicação sem fio aparecem predominantemente nas inovações relacionadas à internet das coisas, em particular, sensores inerciais permitem a análise de variáveis inerciais relativas ao seu local de apoio, o que torna possível coletar dados relacionados à cinemática do corpo humano. A existência de produtos acessíveis e a possibilidade de analisar parâmetros cinemáticos no cotidiano pode fornecer dados valiosos para uma reabilitação fisioterapêutica. Por exemplo, desde que a coleta e processamento sejam suficientemente precisos, informações como a melhora ou piora dos quadros clínicos podem ser obtidas diariamente e até mesmo disponibilizadas instantaneamente para o responsável. Para auxiliar neste quesito, esta pesquisa apresenta um aplicativo multiplataforma para coleta e processamento de dados inerciais, também fornecendo um protótipo de baixo custo, de fácil aquisição e montagem, que é compatível com múltiplas unidades de medição inercial (IMU) e comunicação sem fio. O sistema possui foco em sua estrutura interna, utilizando diversos padrões e princípios aliados a arquitetura limpa para promover maior longevidade e manutenibilidade ao projeto, inclusive facilitando a contribuição de métodos de processamento customizados. Todos os resultados desta pesquisa são fornecidos em código aberto e gratuito com intuito de incentivar o engajamento da comunidade externa.

Palavras-chaves: Arquitetura limpa, Fusão de sensores, Internet das coisas, Padrões de projeto, Processamento customizado de dados, Protótipo de baixo custo, Repositório público com código aberto, Unidade de medição inercial.

Abstract

Technological advances allow research and innovation in several areas. As components reduce in size devices are getting smaller, favoring the portability of prototypes and products. Sensors and wireless communication appear predominantly in innovations related to the internet of things, in particular, inertial sensors allow the analysis of inertial variables related to their place of fixation, which makes it possible to collect data related to kinematics of the human body. The existence of accessible products and the possibility of analyzing kinematic parameters in everyday life can provide valuable data for physiotherapeutic rehabilitation. For example, as long as the collection and processing are sufficiently accurate, information such as the improvement or worsening of clinical conditions can be obtained daily and even made instantly available to the person in charge. To assist in this regard, this research presents a multiplatform application for collecting and processing inertial data, also providing a low-cost prototype, which is easy to acquire and has a simple installation process, furthermore being compatible with multiple Inertial Measurement Unit (IMU) and wireless communication. The software focuses on its internal structure, using several patterns and principles allied to clean architecture to promote greater longevity and maintainability to the project, including facilitating the contribution of customized computing methods. All results of this research are provided in a free and open-source format to encourage the engagement of external contributors.

Keywords: Clean architecture, Custom data processing, Design patterns, Inertial measurement unit, Internet of things, Low-cost prototype, Open source with public repository, Sensor fusion.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

O avanço tecnológico de dispositivos embarcados, juntamente com a diminuição do tamanho físico do *hardware*, possibilita soluções para problemas em diversas áreas, inclusive na área da saúde, como a medicina preventiva e reabilitação fisioterápica. Com a criação de MEMS (Sistemas Microeletromecânicos - tecnologia feita usando métodos de microfabricação), teve início as discussões sobre dispositivos vestíveis (HERSHBERGER, 1991) e em 1997, Paul Saffo escreveu sobre os prospectos futuros dos sensores e sua importância nas inovações. Logo após, em 1999, Kevin Ashton cunhou o termo Internet das Coisas (*Internet of Things*) (SURESH et al., 2014). A nova era de dados, a Internet das Coisas, está cada vez mais presente no cotidiano, e os componentes mais encontrados nestas tecnologias são os sensores.

Sensores inerciais tiveram suas primeiras aplicações em sistemas de navegação e podem ser datados desde 1817 pelo giroscópio apresentado por Bohnenberger (WAGNER; TRIERENBERG, 2010). Na área de saúde, um dos primeiros usos de sensores vestíveis se deu com aparelhos auditivos criados por meio do uso de microfones (ASHTON, 1951). Algumas décadas depois, o esforço concentra-se no desenvolvimento de tecnologias não invasivas para monitoramento e coleta de dados relacionados à saúde do indivíduo no dia a dia (KIZAKEVICH; JOCHEM; JONES, 1988).

Consequente à popularização do Bluetooth em 2002, a tecnologia de pedômetro (medidor de passos por sistema pendular) foi repensada e, com a ajuda de um acelerômetro e a intenção de fornecer mais informações sobre a saúde do indivíduo, nasceram os famosos *Fitness Trackers* para monitorar as atividades físicas praticadas no dia a dia, por empresas como Nike e Fitbit em meados de 2007 (MACMANUS, 2015). Alguns estudos sobre detecção de movimentos com o uso de acelerômetros tiveram início em meados de 1990 (OLSEN; BEKEY, 1986; WILLEMSEN; BLOEMHOF; BOOM, 1990).

1.1 Contextualização

Atualmente, percebe-se uma forte tendência nos estudos de sensores vestíveis, cada vez menos intrusivos e disponibilizados a baixo custo (SHANDILYA; MEENA; KUMAR, 2016; XU et al., 2019). Existem diferentes tecnologias disponíveis para mapear ou monitorar o corpo em movimento, como visão computacional, placas de força e sensores inerciais, estes últimos de menor custo. Entretanto, como mencionado anteriormente, vale destacar que a maioria das soluções baseadas em sensores inerciais vestíveis são tecnologias proprietárias sendo disponibilizadas a um preço elevado (GOULERMAS et al., 2008; AGOSTINI et al., 2015).

A diversidade das unidades inerciais variam de soluções proprietárias de alto custo como Xsens, Shimmer e NXP (GOULERMAS et al., 2008; LIU et al., 2012; KORK et al., 2017; DEHZANGI; TAHERISADR; CHANGALVALA, 2017; ALCARAZ et al., 2018), a protótipos de baixo custo desenvolvidos especificamente para o estudo com controladoras personalizadas (WANG et al., 2012; SHANDILYA; MEENA; KUMAR, 2016; XU et al., 2019). Como concluíram Zebin, Scully e Ozanyan (2015), os resultados de sistemas baseados em sensores inerciais possuem uma precisão aceitável quando usados para monitorar movimento e rotação articulares, indicando possíveis aplicações para diagnosticar e avaliar resultados de reabilitações em diversas doenças neuromusculares, tendo em vista que é possível rastrear e registrar medidas de velocidade e rotação angular.

Mesmo com o avanços na literatura sobre alternativas tecnológicas com comunicação sem fio, como protótipos de baixo custo utilizando uma palmilha com sensores de força (SHANDILYA; MEENA; KUMAR, 2016; XU et al., 2019), a maioria das análises clínicas de fisioterapia ainda são realizadas *in loco* carregando um alto custo atrelado aos equipamentos usados. Em contrapartida ao baixo custo dos sensores inerciais, a precisão é vista como um desafio.

A proposta desta pesquisa vai de encontro à usabilidade dos sensores inerciais e a dificuldade de encontrar soluções de baixo custo que atendam necessidades mais abrangentes de pesquisa, como coleta e processamento de dados em tempo real. A solução aqui apresentada é aberta, gratuita e estruturada. Iniciativas similares são encontradas em alguns trabalhos com foco na área da saúde e biomecânica, porém a maioria delas tem o foco apenas no seu escopo funcional do estudo e não no desenvolvimento do sistema (DELP et al., 2007; LÉTOURNEAU et al., 2021).

Existem trabalhos com propostas similares para análise em tempo real mas ou não foi encontrado um repositório público (VAJS et al., 2020), ou os objetivos e licença diferem em alguns aspectos (LAVIKAINEN et al., 2021), enquanto a presente pesquisa fornece um repositório público[1] com código aberto e gratuito. Além disto, não foram encontrados projetos que permitem a contribuição de métodos de processamento customizados. A solução arquitetônica implementada por este projeto facilita esta contribuição, permitindo a seleção do método de processamento na tela do aplicativo.

A criação de um sistema estruturado com uma arquitetura bem definida provê muitas vantagens ao projeto como um todo, principalmente quando se busca a contribuição da comunidade externa. Um dos benefícios de orientar o desenvolvimento com uma arquitetura, padrões e princípios é o controle orgânico atrelado à estrutura como um todo, incrementando a longevidade e manutenibilidade do projeto. O objetivo da arquitetura do sistema é minimizar os recursos humanos necessários para construir e manter um sistema, visto que com o passar do tempo, após diversos incrementos funcionais, um sistema mal

[1] <https://github.com/brunotacca/INTRACS>

estruturado irá inevitavelmente ter um aumento de custo de manutenção (MARTIN, 2017).

1.2 Objetivos

O objetivo geral deste trabalho é desenvolver um aplicativo multiplataforma para auxiliar na coleta e processamento de dados inerciais por meio de tecnologia de comunicação sem fio nomeado de INTRACS (*Inertial Tracker Computing System*), fornecendo a possibilidade da coletar e processar dados de múltiplos sensores, além de permitir a seleção de um método de processamento dentre os possíveis já implementados. Estas implementações podem ser feitas por meio da contribuição de métodos de processamento que deve ser estruturada de uma maneira simples, diminuindo a barreira de entrada para novos contribuidores no projeto.

Em particular ao desenvolvimento do aplicativo, é objetivo deste trabalho o realizar em uma arquitetura de sistema para favorecer a longevidade e a manutenibilidade do projeto, orientando padrões e princípios para futuras contribuições. Assim, reduzindo a possibilidade de débitos técnicos e *code smells* naturalmente por conta da arquitetura aplicada. Todo o projeto deve ser disponibilizado em código aberto e gratuito em um repositório público contendo uma licença abrangente e diversos documentos de apoio para orientar e auxiliar a comunidade externa.

Especificamente para a coleta dos dados de sensores inerciais, deve ser utilizado um *hardware* de fácil acesso, aquisição e relativamente baixo custo, sendo compatível com múltiplos sensores e transmissão sem fio. Como também, possuir tamanho reduzido e facilidade no posicionamento dos componentes favorecendo a portabilidade e a futura criação de vestimentas que auxiliem na fixação dos sensores em diferentes pontos do corpo.

1.3 Contribuições

As produções aqui apresentadas podem servir como base para diversos trabalhos e pesquisas futuras, facilitando a coleta e processamento de dados inerciais, por meio da redução de custo aquisitivo e da facilidade na configuração de novos métodos de processamento. Seria possível por exemplo montar uma vestimenta inteligente com 32 sensores inerciais em pontos específicos espalhados pelo corpo humano, em que um método de processamento integrasse estes diferentes dados em um valores que determinem o estado atual do corpo, como sentado, deitado, correndo, andando, entre outros. Resumidamente, as principais contribuições do INTRACS são:

- Uso de *hardware* de fácil acesso para construção de um protótipo inercial multissensor capaz de transmitir dados por tecnologia sem fio.

- A criação de um sistema que é executado no protótipo para transmitir os dados dos sensores inerciais para um *smartphone*.
- A criação de um aplicativo multiplataforma que é utilizado para receber os dados de uma unidade inercial por *bluetooth*.
- Estruturação do desenvolvimento em um padrão arquitetônico que favoreça a longevidade e manutenibilidade do sistema.
- Disponibilização de todo projeto em código aberto gratuito em um repositório público[2] com uma licença abrangente e diversos documentos de apoio.
- Implementação de um método de processamento customizado demonstrando o uso da coleta e processamento dos dados inerciais.

1.4 Organização do texto

Os demais capítulos deste trabalho estão organizados da seguinte maneira: O Capítulo 2 apresenta brevemente os conceitos principais que envolvem a construção deste trabalho, e finaliza na Seção 2.6 com uma discussão sobre trabalhos relacionados; O Capítulo 3 detalha a estrutura interna do aplicativo multiplataforma, incluindo seu propósito, tecnologias e arquitetura; As produções advindas da pesquisa são descritas no Capítulo 4, por meio da exibição das telas do aplicativo, os métodos de processamento customizados, o protótipo inercial multissensor e os detalhes do repositório público em código aberto; Por fim, no Capítulo 5 apresenta-se uma discussão sobre os resultados e trabalhos futuros.

[2] <https://github.com/brunotacca/INTRACS>

2 Fundamentação e Trabalhos Relacionados

Existem muitos assuntos que fazem parte da criação de um sistema. As etapas existentes desde a elucidação do problema até a solução entregue são diversas e variam de metodologia para metodologia. Tendo como base que o intuito deste projeto é disponibilizar um sistema aberto à comunidade com um código e uma arquitetura de alta manutenibilidade, além de facilitar a contribuição de métodos de processamento customizados favorecendo um projeto com crescimento contínuo.

Os conceitos mais importantes neste projeto estão relacionados a programação orientada a objetos apresentada na Seção 2.1, seguindo pela Seção 2.2 que explica os padrões de arquitetura que são usados no sistema como um todo. Após isso, na Seção 2.3 é descrito o funcionamento dos sensores inerciais que fornecem os dados para o sistema juntamente com uma breve descrição de alguns trabalhos que fazem o seu uso. Por fim um resumo sobre a fusão de sensores na Seção 2.4, que podem ser implementadas como métodos de processamento customizados neste projeto.

2.1 Programação orientada a objetos

A programação orientada a objetos é um paradigma que define conceitos de classes e objetos com o intuito de modelar um sistema subdividido em partes reutilizáveis de código, o que facilita sua manutenção (GAMMA et al., 1996; PRADHAN; DWIVEDI; RATH, 2015; FOWLER, 2018). Essas características são obtidas por meio da modelagem do problema ao utilizar os conceitos de herança, polimorfismo e encapsulamento. A estrutura de classes que compõem a solução frequentemente contextualiza o domínio do problema por meio de seus componentes e comportamentos (SREEKUMAR; SIVABALAN, 2015; PRADHAN; DWIVEDI; RATH, 2015).

Com a popularidade crescente do paradigma desde seu nascimento, diversas sugestões oferecendo a padronização de soluções para problemas rotineiros começaram a aparecer, intituladas de princípios e padrões. Como o significado destes termos as vezes difere de autor para autor, neste trabalho é considerado que princípios são conceitos abstraídos que guiam o leitor a resolver um problema sem forçar regras concretas de modelagem e estrutura interna. Em paralelo, os padrões fornecem um modelo concreto para um problema em questão, bastando adaptar o proposto para o contexto em que se enquadra (GAMMA et al., 1996; MARTIN, 2000; FOWLER et al., 2002; LARMAN, 2004).

As próximas seções apresentam os princípios GRASP e SOLID como também

os padrões de projeto GoF. Estas diretrizes são constantemente adotadas durante toda a elaboração da arquitetura deste projeto, como também devem servir de orientação no desenvolvimento de novas funcionalidades. É importante ressaltar que os conceitos apresentados por estes autores devem ser vistos como guias para um bom desenvolvimento e não como regras de obrigatória adoção, pois em suas concepções originais todos princípios e padrões possuem suas vantagens e desvantagens (GAMMA et al., 1996; MARTIN, 2000; LARMAN, 2004).

2.1.1 Princípios GRASP

Os princípios GRASP (*General Responsibility Assignment Software Patterns*) definem diretrizes para atribuições de responsabilidades às classes e objetos durante a construção de sistemas orientados a objetos. Os modelos orientam o desenvolvedor a pensar sistematicamente no sistema em termos de responsabilidade, papéis atribuídos aos objetos e colaboração entre objetos. A responsabilidade pode ser atribuída a um objeto solitário ou a um grupo de objetos que trabalham juntos para cumprir esta responsabilidade, e os princípios ajudam a identificar qual será a responsabilidade atribuída e como será a organização dos objetos (LARMAN, 2004).

As responsabilidades são dividades em dois tipos, a de fazer e de saber. A responsabilidade de fazer define que um objeto deve fazer algo como a criação ou execução de um cálculo, iniciar uma ação em outros objetos ou controlar e coordenar atividades em outros objetos. A responsabilidade de saber define que um objeto deve conhecer seus próprios dados privados encapsulados, conhecer sobre objetos relacionados (referências e associações) e conhecer elementos em que o objeto pode se derivar ou calcular. Durante a criação do projeto, as responsabilidades são atribuídas às classes dos objetos. Por exemplo, é possível dizer que “um ComputingMethod é responsável por criar ComputedData”, refletindo a responsabilidade de fazer, ou “um ComputingMethod é responsável por saber seu dados de entrada e saída” refletindo a responsabilidade de saber (LARMAN, 2004).

Os diferentes princípios apresentados em GRASP são: Criador; Especialista de informação; Baixo acoplamento; Controlador; Alta coesão; Polimorfismo; Fabricação pura; Indireção e; Variações protegidas. Estes modelos são respostas a certos problemas recorrentes no desenvolvimento de software. Eles não foram projetados apenas para criar uma nova maneira de desenvolver sistemas, mas para documentar e padronizar melhor o existente usando princípios de programação comprovados em um projeto orientado a objetos (LARMAN, 2004). A arquitetura desenvolvida para este projeto faz uso de todos estes princípios GRASP em suas diferentes camadas e componentes.

O princípio Criador (*Creator*) define quando uma classe B possui a responsabilidade da criação de instâncias de uma classe A. Na arquitetura proposta este princípio está presente na maioria dos componentes, pois cada componente é responsável por responder

apropriadamente, ou seja, com o objeto apropriado definido (LARMAN, 2004). Neste projeto, frequentemente um componente precisa criar e converter um objeto para outro formato para direcionar o fluxo para outra camada. A injeção de dependências possui uma responsabilidade de criação característica, pois faz a ligação de instâncias concretas em todos os componentes que utilizam abstrações na inicialização do sistema.

O princípio Especialista de Informação (*Information Expert*) define a atribuição de responsabilidade de saber e fazer sobre uma informação para as classes que são definidas como especialistas desta informação (LARMAN, 2004). Na arquitetura deste projeto, as camada possuem responsabilidades sobre as informações de entrada e saída e, como cada camada possui também diferentes componentes, cada componente deve respeitar essas responsabilidades, sendo especialistas de informação em seu contexto (Subseção 2.2.1.2).

O Baixo Acoplamento (*Low Coupling*) recomenda que os elementos tenham o mínimo de dependência possível entre eles, pois se houver alterações, os elementos dependentes (acoplados) podem ser afetados. Por exemplo, uma subclasse é considerada altamente acoplada com sua superclasse (LARMAN, 2004). A separação de camadas da arquitetura deste projeto favorece o desacoplamento ao custo de uma maior quantidade código e classes, a regra de dependência apresentada na Subseção 2.2.1.1 explica o acoplamento existente entre as camadas.

O Controlador (*Controller*) é o elemento responsável por delegar ações (como o clique de um botão) para o responsável pela execução das mesmas, sendo o elemento que recebe a entrada e, com o conhecimento necessário, passa adiante a execução para o seu responsável correto (LARMAN, 2004). A Figura 1 ilustra um controlador deste projeto, um componente que existe na camada de adaptadores da arquitetura limpa (Subseção 3.3.4), sendo o responsável por conhecer e permitir a execução dos casos de uso existentes na camada de aplicação.

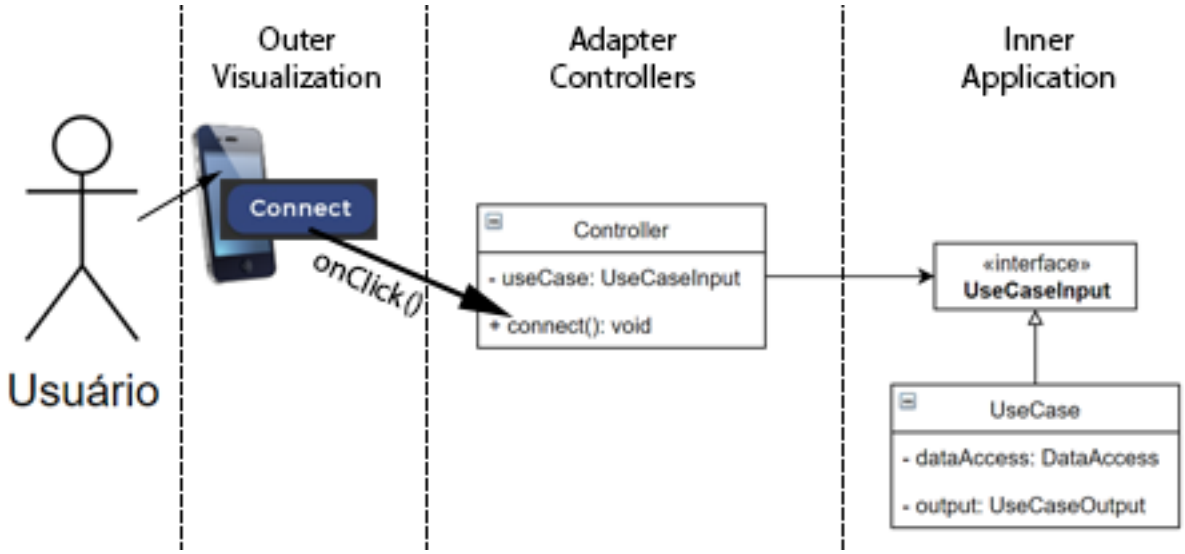


Figura 1 – Controlador responsável por realizar a conexão com um dispositivo Bluetooth.

A coesão é uma medida qualitativa informal medindo a relação funcional das operações de um componente. Um objeto com Alta Coesão (*High Cohesion*) possui operações focadas e gerenciáveis em acordo com sua responsabilidade atribuída, e muito próximo deste conceito está o princípio de responsabilidade única do SOLID (Subseção 2.1.3) (LARMAN, 2004). Em contrapartida, um objeto com baixa coesão possui relações com diversos componentes diferentes,é um objeto que "faz" muito por si próprio e, normalmente foge de sua própria responsabilidade, favorecendo o alto acoplamento (LARMAN, 2004). A alta coesão existe naturalmente na arquitetura deste projeto durante a criação de seus componentes para uma nova funcionalidade e casos de uso. A separação das camadas favorece a divisão de responsabilidades fazendo com que cada componente não precise "fazer" muito, pois entende-se que nas camadas vizinhas existem outros componentes que possuem suas responsabilidades de execução e, no final, todo o fluxo necessário é coberto.

O princípio do Polimorfismo (*Polymorphism*) aparece para resolver o problema de execuções que são orientadas pelo tipo, pois no lugar de escolher encadear condicionais (*if*, *else* ou *switch*) opta-se pelo uso de subtipos e métodos polimórficos (LARMAN, 2004). A regra de dependência na arquitetura limpa permite a comunicação entre camadas internas com camadas externas por meio do polimorfismo com *interfaces*. Além disto, neste projeto as saídas (ou resultados de uma execução) são representadas por uma classe que faz uso do polimorfismo para estruturar o resultado esperado (sucesso de um tipo genérico) ou uma falha (carregando uma exceção).

A Fabricação Pura (*Pure Fabrication*) sugere a criação de uma classe "conveniente" quando a atribuição de responsabilidade fica confusa entre as classes especialistas. Esse componente não representa um conceito no domínio do problema, mas deve existir para manter a alta coesão e baixo acoplamento dos outros componentes (LARMAN, 2004). Na arquitetura limpa, este princípio de conveniência é mais frequente na camada de Adaptadores (Subseção 2.2.1.2), os componentes desta camada são criados fora do domínio do problema e com suas responsabilidades bem definidas, como apresentar a informação (*Presenter*) ou encarregar-se da persistência de dados (*Repository*).

A Indireção (*Indirection*) é um princípio utilizado para atribuir responsabilidade e ao mesmo tempo evitar o acoplamento direto entre dois ou mais objetos, favorecendo o desacoplamento e permitindo uma possível reutilização (LARMAN, 2004). A arquitetura limpa faz uso da Indireção por meio do modelo de Inversão de Dependência, ou seja, as fronteiras entre uma camada interna e uma camada externa são definidas por interfaces na camada interna, que por sua vez devem ser implementadas na camada externa.

E por fim, o princípio de Variações Protegidas (*Protected Variants*) aparece como solução parar projetar elementos do sistema de uma forma que fiquem protegidos de variações (alterações) eventuais em outros elementos, sugerindo a criação de uma interface estável para a atribuição de responsabilidade nos pontos de possíveis instabilidades

(LARMAN, 2004). A separação de responsabilidades por camadas na arquitetura limpa segue a regra de dependência, onde componentes de camadas internas não devem conhecer componentes de camadas externas, criando a comunicação por meio de interfaces (abstrações).

Notavelmente, os princípios GRASP estão constantemente presentes por toda arquitetura limpa e durante a construção, manutenção e incremento deste projeto. A separação de responsabilidades entre as camadas e componentes fornece um desacoplamento natural durante o desenvolvimento de novas funcionalidades, além de favorecer uma redução significativa na complexidade dos algoritmos presentes em cada componente e na comunicação entre os mesmos. Consequentemente, a desvantagem deste desacoplamento e segregação de responsabilidades é um aumento significativo na reescrita de código, além de que a estrutura do sistema como um todo possui uma complexidade maior.

2.1.2 Padrões de projeto GoF

O acrônimo GoF—*Gang of Four* (Gangue dos Quatro)—foi cunhado para os quatro autores do livro de padrões de projeto que descreve elementos reutilizáveis em sistemas orientados à objetos (GAMMA et al., 1996). São 23 padrões de projeto divididos em três categorias distintas: Padrões de criação (*Creational patterns*) para a criação de objetos; Padrões estruturais (*Structural patterns*) para a composição e relacionamento entre objetos e; Padrões comportamentais (*Behavorial patterns*) para as interações (GAMMA et al., 1996).

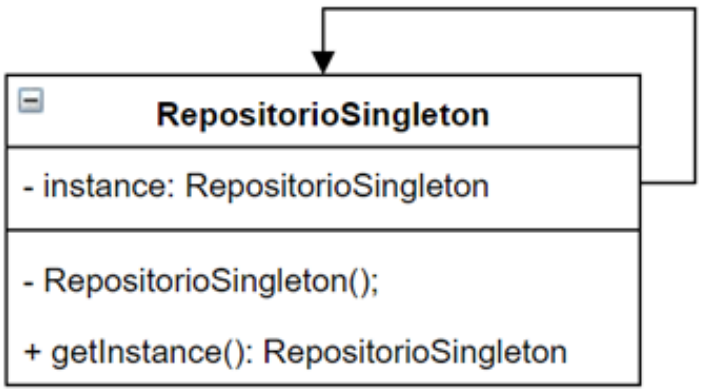


Figura 2 – Exemplo de um repositório no padrão *Singleton*.

Os cinco padrões de criação que auxiliam na decisão de criação novos objetos são: *Abstract Factory*; *Builder*; *Factory Method*; *Prototype* e; *Singleton*; Estes padrões descrevem estruturas diferentes com uso de classes abstratas (ou interfaces) para delegar ou mascarar a criação das instâncias de um objeto. O padrão de criação mais usado na arquitetura limpa em praticamente todos seus componentes é o *Singleton*, que garante que uma classe tenha apenas uma instância (referência na memória), fornecendo um ponto de

acesso único como ilustrado na Figura 2 (GAMMA et al., 1996). Além disso, as instâncias concretas dos componentes das camadas internas da arquitetura devem ser instanciados uma única vez por meio da injeção de dependências (Subseção 2.2.1.4). Esse componente pode ser elaborado por meio do padrão *Abstract Factory* que possui o objeto de criar e retornar as instâncias concretas de uma certa classe ou abstração (MARTIN, 2017).

Os sete padrões estruturais que auxiliam na composição estrutural de objetos são: *Adapter*; *Bridge*; *Composite*; *Decorator*; *Facade*; *Flyweight* e; *Proxy*. Estes padrões descrevem modos de juntar diferentes objetos para adicionar novas funcionalidades. O intuito é melhorar a composição de múltiplas classes e objetos em uma estrutura maior (hierarquia) utilizando polimorfismo e heranças (GAMMA et al., 1996). O padrão estrutural *Facade* ilustrado na Figura 3, que aparece frequentemente na camada de adaptadores da arquitetura limpa, sugere a criação de uma classe para simplificar (e esconder) toda complexidade de estrutura e execução de um subsistema. As camadas externas delegam a execução para camadas internas (sub-sistemas) por meio de componentes (Subseção 3.3.2). O Controlador por exemplo pode ser considerado um padrão *Facade*.

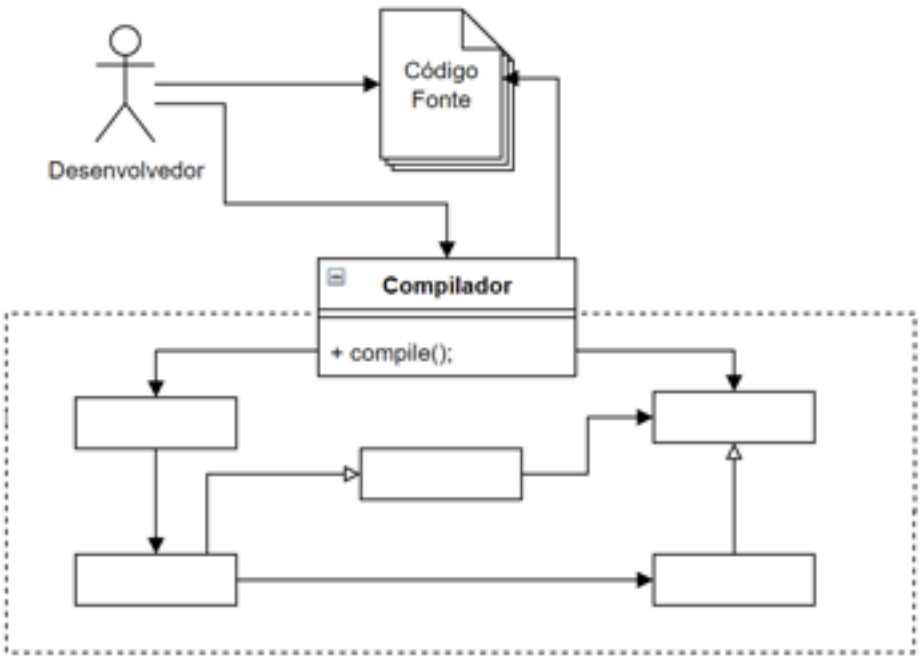


Figura 3 – Exemplo do padrão *Facade* escondendo a complexidade de um compilador (Adaptado de Gamma et al. (1996)).

Os 11 padrões comportamentais que auxiliam na composição estrutural de objetos são: *Chain of Responsibility*; *Command*; *Interpreter*; *Iterator*; *Mediator*; *Memento*; *Observer*; *State*; *Strategy*; *Template Method* e; *Visitor*. Estes padrões são focados na execução e atribuição de responsabilidades entre objetos ou classes com o uso de herança ou composição, com o objetivo de facilitar a visualização de um fluxo de controle originalmente complexo (GAMMA et al., 1996). Um padrão importante na arquitetura deste projeto é o

padrão *Strategy*, pois os métodos customizados de processamento são criados seguindo este padrão. Ele permite a definição de uma família de algoritmos desacoplada em que cada membro da família pode ser selecionado em tempo execução como ilustrado na Figura 4, em outras palavras, permitindo que o mesmo objeto trabalhe com diferentes algoritmos da mesma família em tempo de execução (GAMMA et al., 1996).

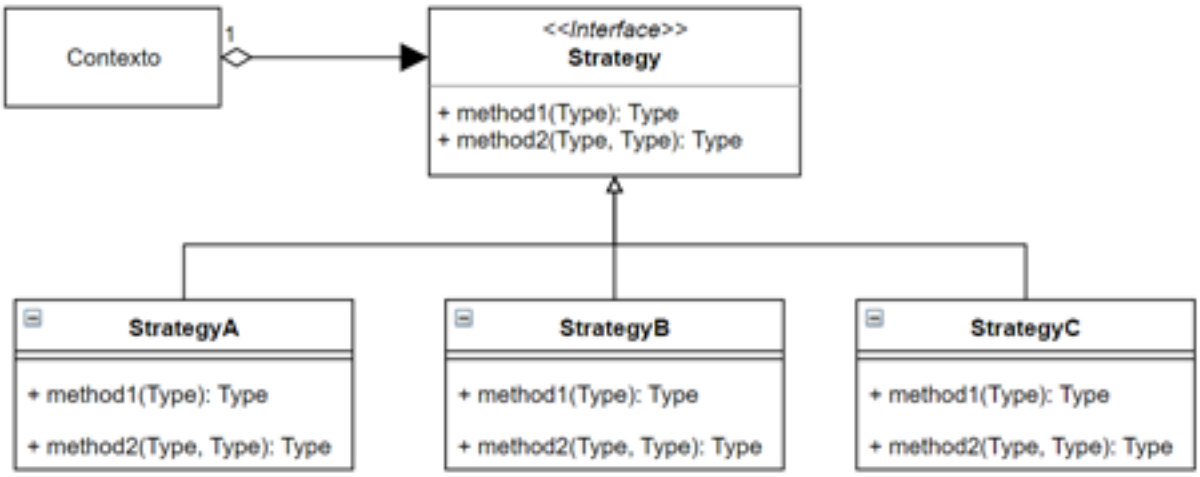


Figura 4 – Modelo do padrão *Strategy* (Adaptado de Gamma et al. (1996)).

Os padrões de projeto GoF auxiliam na elaboração de novas estruturas durante o desenvolvimento de novas funcionalidades para o projeto, especialmente quando a solução não é trivial e não está coberta pela arquitetura limpa. A solução para métodos de processamento customizados é um ótimo exemplo que não está presente na proposta original da arquitetura limpa. Nesse sentido, o padrão *Strategy* adotado facilita a adição de novos algoritmos em contribuições pela comunidade.

2.1.3 Princípios SOLID

A orientação a objetos oferece mecanismos para que objetos consigam representar conceitos da realidade. Naturalmente o que é real está em constante mudança e a representação no sistema deve possuir um certo nível de tolerância a alterações, dependendo de sua estrutura. Esta tolerância é um dos fatores chave em desenvolvimento de sistemas, pois a manutenção demanda muitas vezes esforços crescentes ao longo do tempo de vida do sistema (MARTIN, 2000). O acrônimo SOLID condensa cinco princípios fundamentais para a concepção de objetos que respondem aos problemas que afetam a escalabilidade e longevidade de um sistema: *Single-Responsibility Principle*; *The Open/Closed Principle*;

The Liskov Substitution Principle; *Interface Segregation Principle* e; *Dependency-Inversion Principle* (MARTIN, 2003; MARTIN, 2017).

O princípio da responsabilidade única (*Single-Responsibility Principle*) define que uma classe deve possuir apenas um motivo para alterações, conceito que está muito próximo ao princípio de alta coesão do GRASP (Subseção 2.1.1), porém com o significado de coesão alterado para os motivos que causam uma classe a ser alterada. Resumidamente, considera-se o fato de que quanto mais responsabilidades uma classe possui, menos coesão existe fazendo com que a classe esteja mais propensa a mudanças (MARTIN, 2003; LARMAN, 2004; MARTIN, 2017). A arquitetura limpa favorece a responsabilidade única em diferentes componentes, principalmente nas camadas internas, o maior exemplo de responsabilidade única são os casos de uso (Subseção 3.3.3.2), pois a arquitetura força a criação deles com um único propósito, que dificilmente é alterado e só está susceptível a alterações quando o próprio domínio muda (Subseção 3.3.3).

Semelhantemente, o princípio aberto-fechado (*Open/Closed Principle*) define que as entidades de um sistema devem estar fechadas para modificações mas abertas para extensões. Isto significa que é preferível criar sub-classes ou incrementar a composição de membros de uma entidade (respeitando o princípio de responsabilidade única), do que modificar o estado atual desta classe, modificando os membros existentes para se adaptar ao novo comportamento (MARTIN, 2003; MARTIN, 2017). A camada mais interna da arquitetura representa a solução para o problema por meio de entidades (Subseção 3.3.3), pois esta camada contém os componentes que mais afetam a arquitetura em caso de modificações. Este e muitos outros princípios devem estar presentes durante a elaboração desta camada para qualquer solução nesta arquitetura.

O princípio da substituição de Liskov (*The Liskov Substitution Principle*) foi escrito por Barbara Liskov em 1987 definindo a seguinte propriedade de substituição: se para cada objeto o1 do tipo S há um objeto o2 do tipo T de forma que, para todos os programas P definidos em termos de T, o comportamento de P é inalterado quando o1 é substituído por o2 então S é um subtipo de T (LISKOV, 1987; MARTIN, 2017). Em outras palavras este princípio define que, se as classes B e C são implementações (filhos hierárquicos) de A, logo os objetos de B e C podem ser trocados entre si sem que o sistema seja afetado. A substituição de Liskov está presente em todos as implementações das abstrações de fronteiras na arquitetura limpa. Dentro de uma camada não se conhece o objeto concreto da camada vizinha, a dependência (o uso) acontece pelo uso da abstração existente dentro da camada (Subseção 2.2.1.1). Uma das maiores vantagens dessa arquitetura é poder trocar uma implementação de Bluetooth por *Wi-Fi*, ou até mesmo uma implementação em PostgreSQL para MongoDB sem afetar o sistema internamente.

O uso extensivo de interfaces em um sistema de longa vida útil induz o aparecimento de interfaces volumosas. Por serem muito grandes elas perdem a coesão e acabam forçando

dependências que fogem do escopo de responsabilidade de suas heranças. O princípio da segregação da interface (*Interface Segregation Principle*) indica que uma classe não deve ser forçada a implementar interfaces e métodos que não irá utilizar, conhecendo apenas suas abstrações que possuem alta coesão. Isto quer dizer que quando temos um problema que irá reduzir a coesão de uma interface, a solução é segregar esta interface (divida-lá, separa-lá) em partes coesas, mantendo assim a alta coesão e não forçando dependências em toda cadeia hierárquica (MARTIN, 2003; MARTIN, 2017). Como a arquitetura limpa é muito desacoplada em sua concepção original, os componentes e interfaces tendem a ser simples e concisos, mantendo a alta coesão durante o ciclo de vida do sistema.

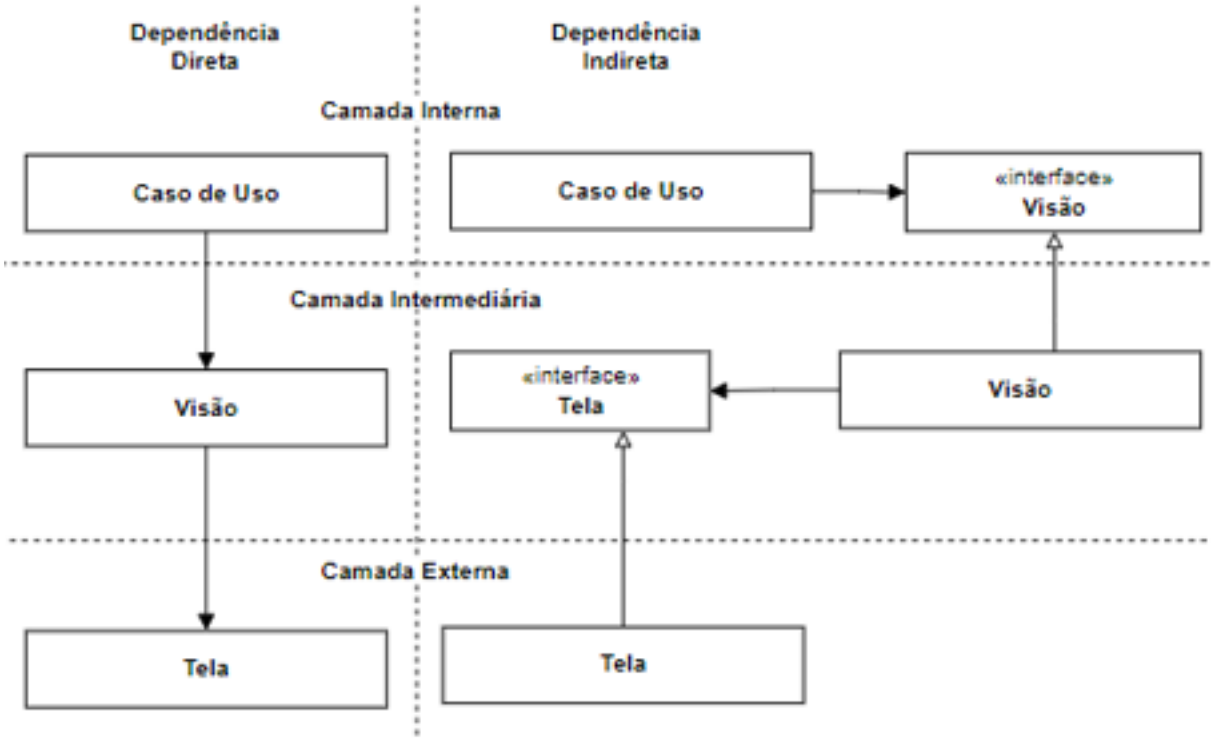


Figura 5 – Diferença da dependência direta e do padrão *Dependency Inversion*.

Por fim, a Figura 5 ilustra o princípio de inversão da dependência (*Dependency-Inversion Principle*) que define duas regras: Módulos de alto nível não devem depender de módulos de baixo nível, mas ambos devem depender de abstrações e; Abstrações não devem depender de detalhes, detalhes devem depender de abstrações (MARTIN, 2003; MARTIN, 2017). Logo, as dependências de uma classe não devem ser concretas mas abstratas, sem que precise conhecer a implementação que será usada. A arquitetura limpa aplica a inversão de dependência entre todas suas camadas, de fato a camada de aplicação e de entidades são as camadas de mais alto nível e não dependem das outras camadas, apenas de componentes internos (Subseção 3.3.3).

2.2 Padrões arquitetônicos de projeto

Com o passar do tempo, um sistema que passa por alterações periódicas está propenso ao incremento de dívidas técnicas por falhas de modelagem e desenvolvimento. Um padrão arquitetônico de projeto (arquitetura de sistema) é um conjunto de princípios, com padrões e conceitos genéricos que juntos formam um modelo abstrato para um ou mais sistemas se encaixarem (GARLAN; SHAW, 1994; FOWLER et al., 2002; MARTINI; BOSCH, 2015). Garlan e Shaw (1994) estabelecem um padrão arquitetônico como a definição do vocabulário de componentes e conectores que podem ser usados em instâncias do padrão, juntos com um conjunto de restrições no modo em que podem ser combinados.

A definição exata de o que é realmente um padrão arquitetural de projeto difere entre autores, mas existem dois elementos comuns: a quebra em alto nível de um sistema em partes e decisões que são difíceis de serem alteradas (MARTIN, 2017). Também é perceptível que não há apenas um modo correto de definir a arquitetura para um sistema pois ela pode variar de acordo com o contexto na sua criação e mudar durante o ciclo de vida do projeto (FOWLER et al., 2002). Martin (2017) define arquitetura de sistema como um modelo de especificações contendo bastante detalhes de baixo nível que dão suporte a decisões de alto nível, sendo ambas decisões parte de um todo, a arquitetura do sistema.

As opções mais estudadas para este projeto foram: A Arquitetura *Ports and Adapters* de Cockburn (2005); a Arquitetura *Onion* de Palermo (2008) e; a Arquitetura Limpa (*Clean Architecture*) de Martin (2017). Embora todas estas arquiteturas compartilhem similaridades entre elas, neste projeto optou-se pela arquitetura limpa com algumas adequações para que os métodos customizados de processamento de dados inerciais sejam fáceis de ser implementados e adicionados ao sistema, sem que a arquitetura e camadas internas sofra algum impacto inesperado na adição de um método.

2.2.1 Arquitetura limpa

A arquitetura limpa foi proposta por Martin (2017) com o intuito de juntar os conceitos de diferentes arquiteturas em uma ideia mais concreta e detalhada. Os pontos em comum avaliados pelo autor são: Independência de *frameworks*; Facilidade em testes unitários; Independência de interface do usuário; Independência de banco de dados; e Independência de agentes externos (MARTIN, 2017). De fato, a proposta visa um total desacoplamento das regras de negócio e casos de uso de quaisquer fatores externos, isso permite que o sistema seja funcional com apenas suas camadas internas, possibilitando facilmente na troca de componentes externos, como *frameworks* de interface, banco de dados, etc.

Na Figura 6 é ilustrada a estrutura da arquitetura limpa. As camadas concêntricas

representam a ordem do fluxo de controle e as dependências dos componentes internos de cada camada. Quanto mais afastado do centro mais dependente a camada será, e quanto mais perto do centro menos dependente a camada será. Isso significa que a camada de entidades (*Entities*) e a camada de casos de uso (*Use Cases*) possuem maior independência em relação às outras camadas. Por outro lado a camada mais externa de *Frameworks e Drivers* é a mais dependente de todo o sistema (MARTIN, 2017).

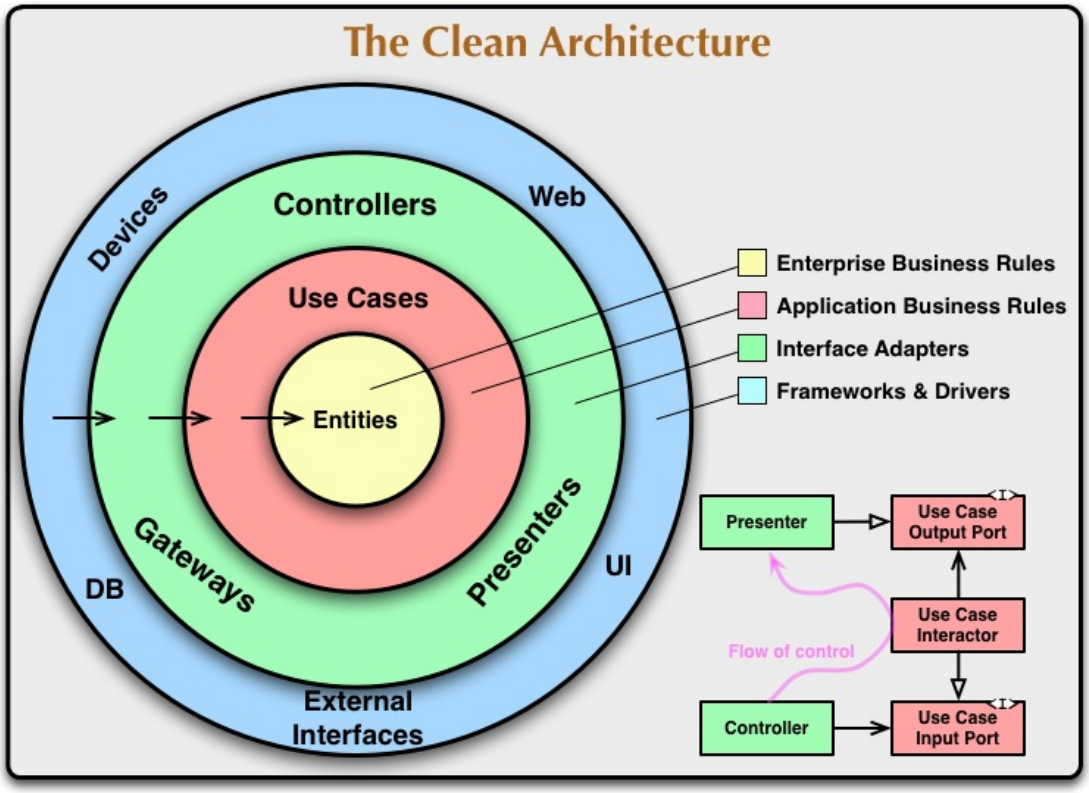


Figura 6 – *The clean architecture* (MARTIN, 2017, p.192 fig.22.1).

As setas que partem da camada mais externa para a camada mais interna na Figura 6 representam, de maneira resumida, a regra de dependência que é abordada na Subseção 2.2.1.1. Além disso, também existe a distinção entre quatro camadas concêntricas em que cada uma possui sua própria responsabilidade com seus próprios componentes, que são detalhadas na Subseção 2.2.1.2. E por fim, pode-se notar uma pequena ilustração exemplificando o fluxo de controle na arquitetura, apresentado na Subseção 2.2.1.3 (MARTIN, 2017). O que não está presente nesta imagem é a injeção de dependências, um componente essencial que deve existir para qualquer sistema que adota essa arquitetura, este é abordado na Subseção 2.2.1.4.

2.2.1.1 Regra de dependência

Os diferentes círculos concêntricos representam diferentes setores do sistema, em que cada área possui sua própria responsabilidade. Resumidamente, os círculos externos

possuem mecanismos enquanto os círculos internos possuem regras de negócio (MARTIN, 2017). Como ilustrado na Figura 6 a regra de dependência possui um sentido, de fora para dentro, dos círculos externos para os círculos internos, isso significa que os componentes existentes em cada círculo possuem dependência com componentes do círculo interno vizinho, porém, são independentes de um possível vizinho externo (MARTIN, 2017).

A Figura 7 ilustra a regra de dependência e seus componentes em cada camada. Nota-se que quanto mais perto um componente está no centro, maior o seu nível de abstração e menor a sua probabilidade de ser alterado, de outro modo, os componentes mais externos são mais concretos e sofrem alterações frequentes. A Figura 7 exemplifica por exemplo que componentes da interface do usuário são dependentes dos *Presenters*, como também os componentes de implementação do banco de dados são dependentes dos *Gateways*. Porém, o inverso não é verdadeiro, *Gateways* e *Presenters* não são dependentes de implementações de banco de dados ou de interface do usuário (MARTIN, 2017).

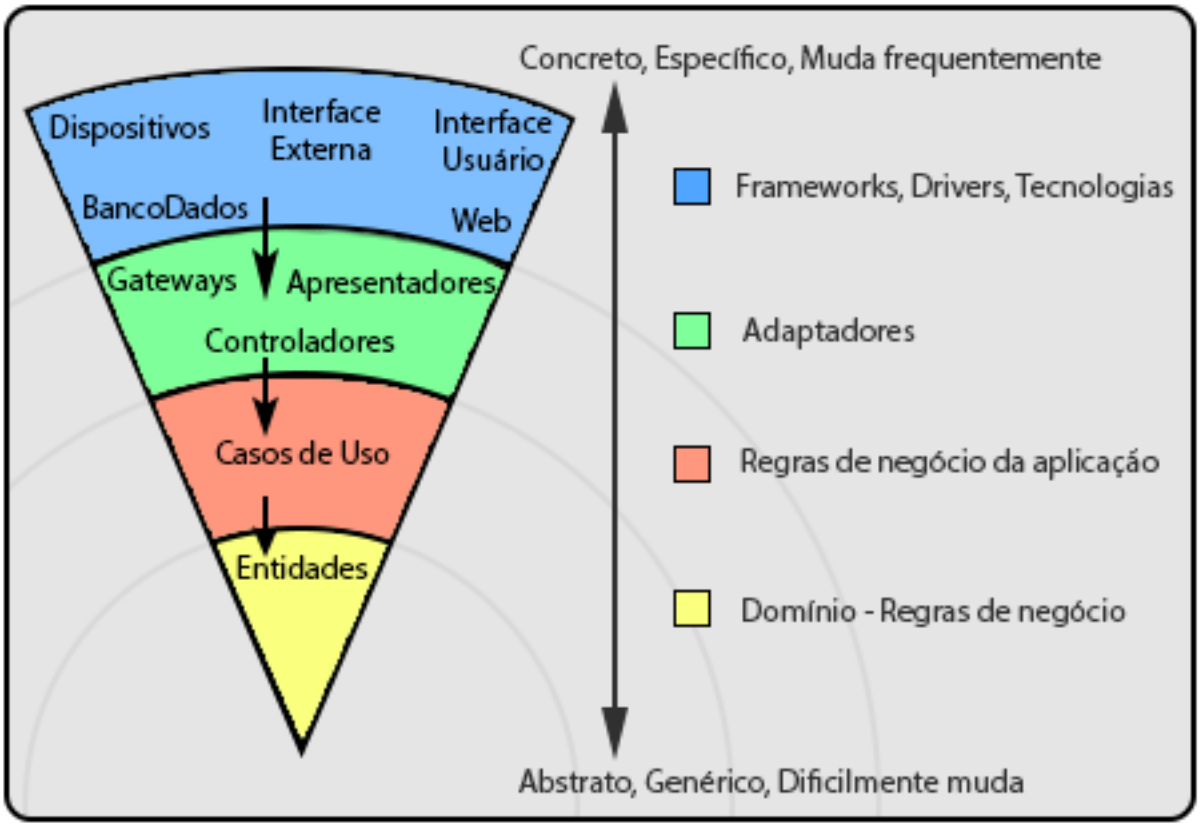


Figura 7 – Nível de abstração com componentes das camadas da arquitetura limpa.

A regra de dependência é a base de sustentação da arquitetura limpa e não deve ser violada, pois ao quebrá-la cria-se o acoplamento de uma camada interna com uma camada externa, o que implica que quando uma camada externa sofrer uma alteração, a camada interna que estiver acoplada também sofrerá. A independência de interfaces, *frameworks*, banco de dados e agentes externos mencionada pelo autor se dá majoritariamente pela regra dependência. O benefício é ter a parte mais importante do seu sistema (regras

de negócio e casos de uso) protegidos contra eventos que não se pode controlar, como a atualização repentina de um *framework* por exemplo (MARTIN, 2017).

2.2.1.2 Camadas e suas responsabilidades

Como apresentado na Figura 6 a arquitetura limpa é separada em quatro círculos concêntricos sendo cada um deles uma das seguintes camadas, sequencialmente da mais externa para mais interna: *Frameworks and Drivers*; *Interface Adapters*; *Application Business Rules* e; *Enterprise Business Rules*. Cada uma dessas camadas possui sua própria responsabilidade e componentes. A Figura 7 ilustra as camadas e componentes com as camadas externas com partes mais específicas podendo sofrer alterações frequentemente sem afetar as camadas internas que possuem membros mais abstratos que dificilmente sofrem alterações (MARTIN, 2017).

A Figura 8 apresenta as camadas e acrescenta uma divisão entre os componentes dentro de uma camada, indicando que dentro de uma camada um grupo de componentes não deve ter conhecimento do próximo. Esta definição é um pouco subjetiva pois mesmo que esta dependência exista ela não viola a regra de dependência. Também observa-se que as camadas mais internas da arquitetura que envolvem regras de negócio não possuem recortes internos, pois elas só possuem um grupo unitário de componentes cada uma, os casos de uso (*Use Cases*) e as entidades (*Entities*).

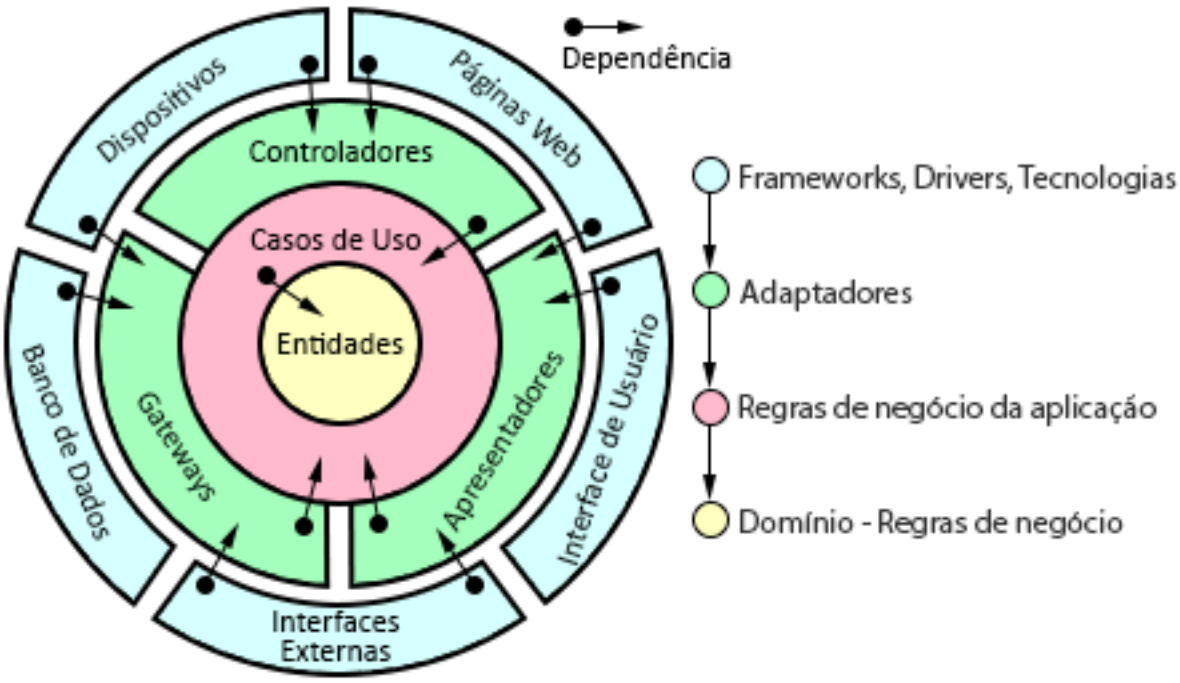


Figura 8 – Arquitetura limpa em camadas e subcamadas divididas.

A camada mais externa de *Frameworks* e *Drivers* é o local de maior dependência com fatores externos pois ela é composta de ferramentas e implementações específicas, como uma integração com um banco de dados (como MySQL, PostgreSQL ou MongoDB) ou a implementação visual utilizando um *framwork web* (como Angular, Laravel ou Spring). Resumidamente, as responsabilidades dos componentes desta camada são de realizar as solicitações e comunicações que as camadas internas disponibilizam e necessitam, ou seja, as camadas externas não possuem regras de negócio (MARTIN, 2017). Um bom exemplo deste projeto é a implementação da comunicação *Bluetooth*. Os componentes da camada externa não decidem que é necessário listar todos dispositivos *Bluetooth* disponíveis para conexão, pois isto foi decidido nas camadas internas, porém esta camada possui o código de implementação que irá realizar a busca e exibir esta listagem.

A próxima camada é a de adaptadores de interface (*Interface Adapters*) que possui uma responsabilidade bem simples: realizar a conversão de dados entre camadas internas para um formato mais conveniente para as camadas externas e vice-versa. Os dados que trafegam para dentro dos casos de uso (*Use Cases*) passam pelos controladores (*Controllers*), e os dados que trafegam para fora dos casos de uso passam pelos apresentadores (*Presenters*) e são exibidos nas visões (*Views*) (MARTIN, 2017). A aplicação (camada externa) solicita a lista de dispositivos *Bluetooth* por meio de um controlador, que conhece os casos de uso e sabe qual exatamente executar, após isso o caso de uso irá enviar os dados para um apresentador, que irá redirecionar os dados para uma visão presente na aplicação.

Por conseguinte, a camada de *Application Business Rules* possui os casos de uso (*Use Cases*). Esta camada é responsável por fazer o projeto funcionar pois as implementações de todos os casos de uso estão presentes e encapsulados nela. Os casos de uso realizam o manuseio do fluxo de dados com as entidades (*Entities*) para atender os objetivos específicos da solução, ou seja, são a realização de regras de negócio da aplicação, e não apenas do domínio. Ainda nesta camada, não há conhecimento de quem está chamando os casos de uso, como também não há conhecimento de aonde serão apresentados os dados e muito menos em que local de persistência a aplicação está buscando dados (MARTIN, 2017). O caso de uso de se conectar a um dispositivo está implementado nesta camada. O parâmetro que é passado para dentro do caso de uso representa um dispositivo e seu formato é definido como um DTO (*Data Transfer Object*), o controlador então sabe que deve passar o valor do dispositivo neste padrão.

No interior de toda arquitetura existe a camada de regras de negócio (*Enterprise Business Rules*) que representa as regras de negócio críticas do problema. Nesta camada existem apenas entidades e o relacionamento entre elas formando um modelo que representa o domínio do problema, ou domínio do negócio. Uma entidade pode ser um objeto com métodos ou um conjunto de estruturas de dados com funções, o importante é encapsular

as regras de negócio e objetos que representam a solução para o problema em uma única camada. Os objetos desta camada por exemplo não são afetados por um problema em um *framework* ou um erro de codificação em um caso de uso (MARTIN, 2017).

A Figura 9 ilustra brevemente os componentes principais da arquitetura limpa em suas diferentes camadas. Note que as camadas internas não possuem conhecimento dos componentes das camadas externas pois não há uma seta (indicando dependência) saindo de dentro para fora de uma camada. O contrário no entanto é verdadeiro, as camadas externas dependem das camadas internas, existem setas que saem dos componentes para uma camada interna, pois é o único sentido permitido pela regra de dependência. Observe também que a inversão de dependência está presente quando um componente interno precisa passar o controle para fora, como o *UseCase* na imagem fazendo uso da abstração de um *Presenter*.

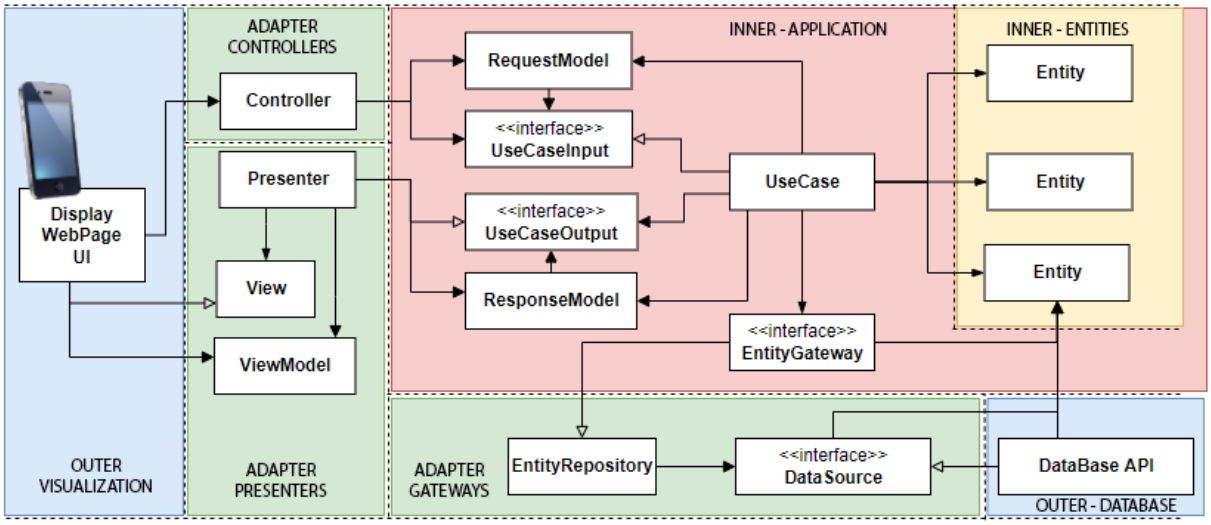


Figura 9 – Camadas e relação entre diferentes componentes na arquitetura limpa (Adaptado de Martin (2017)).

2.2.1.3 Fluxo de controle e componentes

O fluxo de controle entre os componentes de cada camada determina como a comunicação será feita entre as camadas. Os padrões de projeto usados para implementar os componentes permitem que uma camada interna passe o controle para uma camada externa sem criar dependência, e de mesmo modo, que uma camada externa se comunique com uma camada interna sem conhecer os detalhes da implementação. O princípio mais utilizado desta arquitetura, tornando possível o fluxo de controle, é o princípio da inversão de dependência do SOLID (MARTIN, 2017).

A inversão de dependência permite que o componente de uma camada interna passe o controle para o componente de uma camada externa sem criar dependência. A

Figura 10 ilustra a inversão de dependência entre duas camadas. Nota-se que o caso de uso possui dependência com uma interface que existe na mesma camada, isto quer dizer que o componente usa os métodos dessa interface. Neste caso a interface determina os métodos necessários para manusear os dados da camada externa.

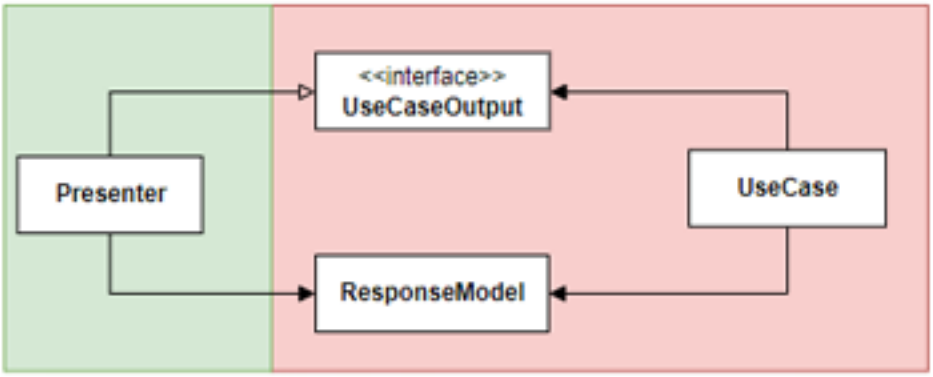


Figura 10 – Inversão de dependência entre caso de uso e apresentador.

Observe que como é uma interface (abstração), o caso de uso precisa de uma implementação concreta para que exista a comunicação. Esta implementação existe na camada externa, sendo instanciada e sua referência é passada para o caso de uso na sua inicialização, ou seja, esta dependência só existe em tempo de execução por meio da injeção de dependências.

A Figura 11 apresenta como o fluxo de controle é passado das camadas externas para os casos de uso na camada interna. Um componente na camada de visualização (um botão por exemplo) faz a chamada para um método do controlador, o controlador então faz a chamada de um caso de uso pela sua abstração, o caso de uso que implementa essa abstração pedirá os dados para a abstração de acesso de dados, o objeto que implementa essa abstração irá então pedir os dados para um objeto que realiza a implementação de um banco de dados específico.

Após isso os dados retornam para o caso de uso que irá manusear os dados com as regras de negócio para criar uma resposta para a abstração do apresentador. O objeto que implementa o apresentador irá receber e converter os dados a serem apresentados na visualização e então realiza a chamada para a abstração da visualização. Aqui o controle esta de volta à camada externa que iniciou o fluxo, o objeto que implementa a abstração de visualização recebe os dados que foram pedidos para o controlador no início da chamada, e os exibem em algum lugar da tela (uma página, um *dialog* de confirmação, etc.).

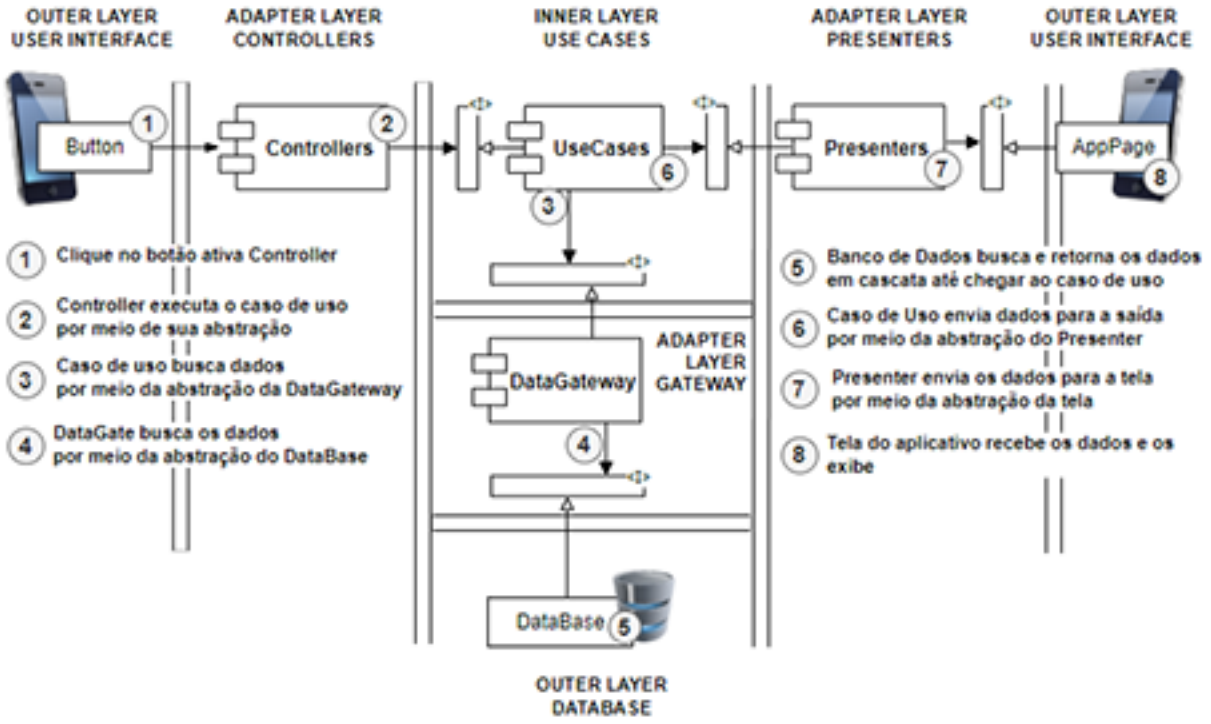


Figura 11 – Fluxo de controle entre diferentes componentes e camadas.

2.2.1.4 Injeção de dependência

Como demonstrado anteriormente a arquitetura limpa é separada em diversas camadas concêntricas, em que os componentes de uma camada interna não devem ter conhecimento dos componentes de uma camada externa, e que preferencialmente os componentes de uma camada externa devem fazer uso dos componentes de uma camada interna por meio de suas abstrações (MARTIN, 2017). A regra de dependência juntamente com o princípio de inversão de dependência do SOLID impõem uma complexidade adicional no sistema como um todo, pois os objetos concretos que implementam as abstrações devem ser referenciados em tempo de execução para os componentes que fazem uso destas abstrações. Isto implica que é necessário um padrão para ligar todas essas pontas soltas logo na inicialização do sistema.

Uma das possibilidades é o uso extensivo entre as camadas do padrão de criação *Abstract Factory* do GoF (Subseção 2.1.2). Este padrão permite o manuseio dos objetos concretos por meio de métodos em uma classe abstrata, como é ilustrado na Figura 12 (GAMMA et al., 1996; MARTIN, 2017). A outra alternativa, adotada por este projeto, é fazer uso de uma biblioteca de injeção de dependências no ponto de entrada do sistema, na camada externa de visualização, pois estas bibliotecas normalmente fazem o uso de padrões de projeto como *Abstract Factory* e *Singleton* para fornecer as dependências corretas quando é solicitada uma abstração. A Subseção 3.3.6 demonstra o uso desta biblioteca na inicialização do sistema.

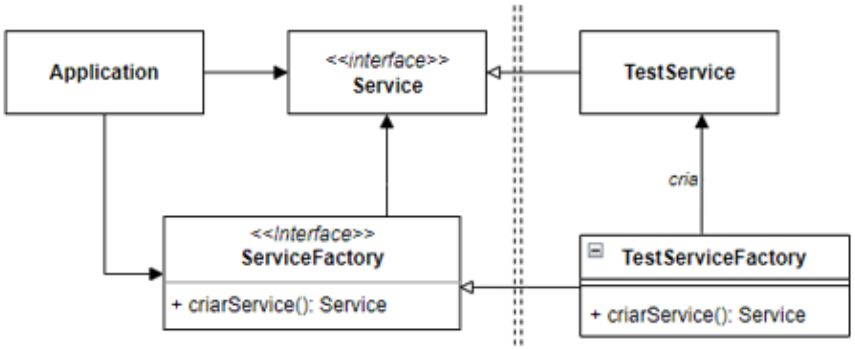


Figura 12 – Modelagem do padrão *Abstract Factory* (Adaptado de Gamma et al. (1996)).

2.3 Sensores

Sensores são produzidos a partir das diferentes interações entre os componentes químicos e físicos de diversos materiais existentes na natureza. Logo, dependendo da propriedade ou lei aplicada, é possível medir as variações de valores resultantes da interação do componente com algo fixado a ele, próximo ou até mesmo distante. Sensores inerciais por exemplo, fazem uso da lei da inercia, já a bussola digital usa uma propriedade magnética nomeada de efeito Hall (KEMPE, 2011; MACMANUS, 2015).

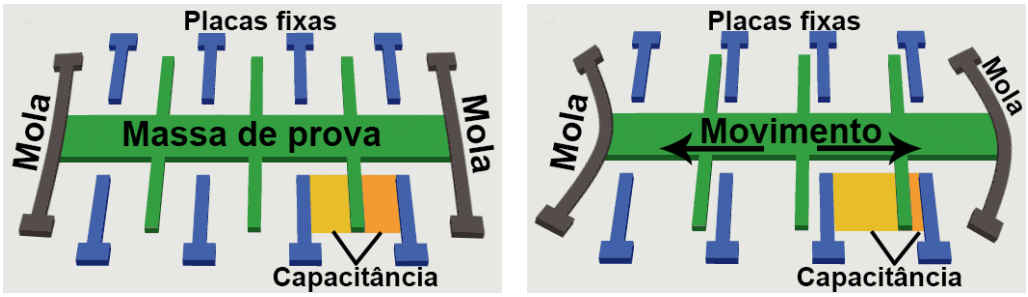
O avanço tecnológico permite que o tamanho dos dispositivos diminua consideravelmente. A área de microeletrônica (e nanoeletrônica) é a grande beneficiadora desse ramo e a criação dos MEMS (Sistemas microeletromecânicos) possibilitou o desenvolvimento de dispositivos não intrusivos. Embora sensores de maior precisão são ainda de difícil acesso ao público de modo geral, a tecnologia já é bem difundida em diversos aparelhos utilizados no cotidiano, contendo tamanho reduzido e módulos de comunicação sem fio, como *Bluetooth*, *Wi-Fi* e ondas de baixa frequência (HERSHBERGER, 1991; BEEBY et al., 2013; SURESH et al., 2014; MACMANUS, 2015).

2.3.1 Sensores Inerciais

O princípio da inercia aparece com Galileu e é constatado pela primeira lei de Newton que enuncia “Todo objeto permanece em seu estado de repouso ou movimento uniforme numa reta, a menos que seja obrigado a mudar seu estado devido às forças impri-midas sobre ele”. O uso dessa lei da física dinâmica possibilita a criação de componentes, que por meio de uma massa de prova, usam a inercia para “perceber” as variações de movimento (DELLIAN, 2003; KEMPE, 2011).

As unidades de medida inercial são sistemas microeletromecânicos construídos com métodos capazes de realizar esta ação. Os piezoelétricos ou piezoresistivos podem ser constituídos de PZT (Titanato zirconato de chumbo), cristais de quartzo ou turmalina, porém o mais encontrado são os capacitivos, que são construídos com *wafers* (cortes) de

silício. Como exemplificado na Figura 13, quando o objeto se movimenta, a massa de prova (eletrodo), que está presa em hastes de flexibilidade reduzida (molas), se movimenta de acordo. Logo, quando a distância entre a massa de prova e as hastes fixas diminui ou aumenta, a capacitância do sistema é alterada, o que torna possível a medição da capacitância e assim é atribuído um valor ao efeito de inercia (KEMPE, 2011).



(a) (b)

Figura 13 – Ilustração sobre o funcionamento de um acelerômetro em um eixo ortogonal, exibindo o efeito de inércia: a) Sistema em repouso, sem movimento; b) Movimento da massa de prova provocado pela inércia (Adaptado de Nedelkovski (2016)).

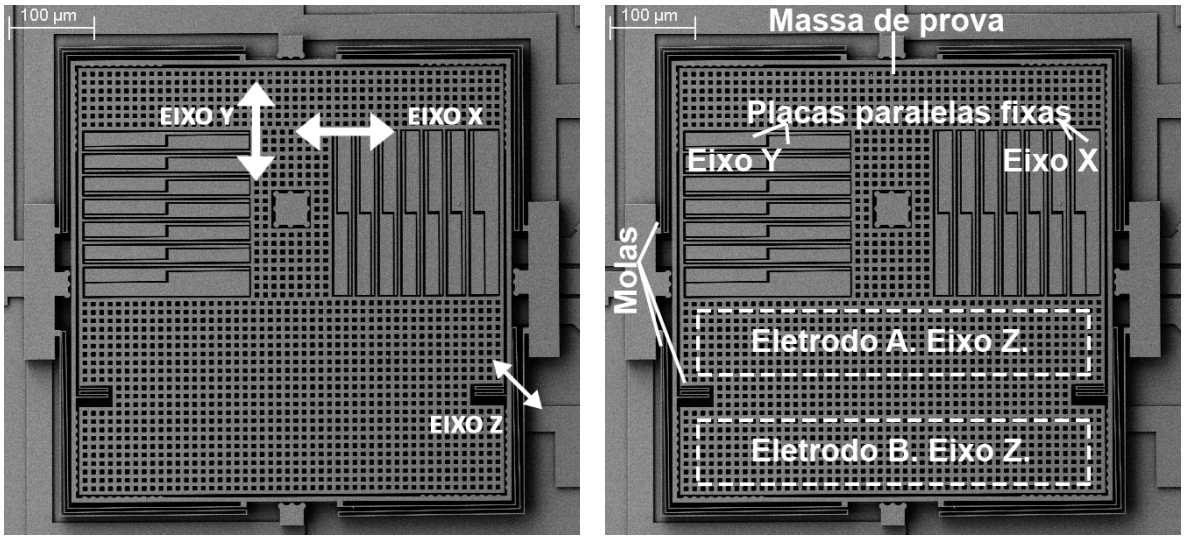
Os sensores inerciais podem ser analógicos ou digitais. Os analógicos fornecem uma saída de voltagem analógica indicando a alteração do componente inercial pela força do sinal, o qual pode ser medido diretamente ou por meio de um microcontrolador com um conversor analógico digital integrado. O digital, mais encontrado atualmente, fornece os dados de aceleração do componente inercial por meio de um barramento serial. Os mais populares são os padrões I^2C (*Inter-Integrated Circuit*) e SPI (*Serial Peripheral Interface*). Estas especificações explicam a arquitetura a ser usada para comunicação em curta distância, sendo muito usadas na construção de circuitos integrados, interligando os componentes com o processador e microcontroladores.

O potencial dos sensores inerciais para o monitoramento do movimento do corpo humano é bem exposto na literatura. Com a tecnologia de MEMS avançando a cada dia, temos sensores cada vez mais precisos, possibilitando diversos estudos na área de saúde e reabilitação (ZHAO et al., 2017). O baixo custo e mobilidade desses sensores fornecem alternativas promissoras para o público em geral, principalmente para o monitoramento remoto de um quadro clínico (PATEL et al., 2012; JARCHI et al., 2018). Ao comparar os dados provenientes de sensores posicionados no quadril e no joelho com um sistema óptico de marcadores infravermelho (STEP32), Agostini et al. (2015) obtiveram resultados com uma margem de erro clinicamente aceitável. Conclusão parecida é exibida no estudo feito por Chen et al. (2016), que constata a viabilidade dos sensores vestíveis como substituto de sistemas laboratoriais de análise da marcha, além de prover uma liberdade maior ao paciente e dados contínuos no dia a dia, o que reforça a necessidade de sistemas em código aberto de baixo custo para coleta e processamento destes dados.

2.3.1.1 Acelerômetro

O acelerômetro é um sensor que quando fixado a um objeto qualquer (como um *smartphone*), fornece a possibilidade de medir a aceleração linear em uma direção a partir de um ponto de repouso. Um mecanismo de inércia em si é capaz de fornecer apenas um eixo ortogonal, logo, quando fornecido nos três eixos ortogonais, são necessários três acelerômetros, porém, o conjunto todo é chamado de apenas acelerômetro, só que com três GDL (Graus de liberdade - *Degrees of Freedom*).

Mesmo que aceleração linear seja definida em m/s^2 pelo SI (Sistema Internacional de Unidades), a maioria dos componentes fornecem a medida em g (Força-G, força gravitacional da terra), sendo aproximadamente $9,806 m/s^2$. Isso se deve ao fato de que o acelerômetro, quando em repouso, vai acusar a medição de $1g$ devido à força gravitacional da terra. Logo, quando em queda livre, a medição deverá ser de $0g$. Conclui-se por esses detalhes a necessidade de um refinamento dos dados e o aprimoramento da precisão com a fusão de sensores.



(a) (b)

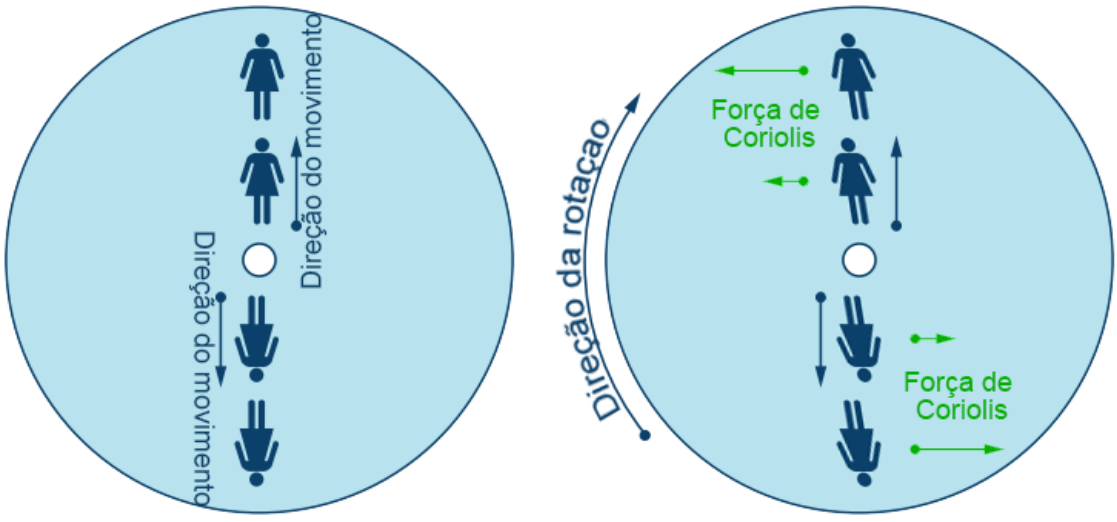
Figura 14 – Imagem microscópica (escala de $100\mu m$) de um acelerômetro capacitivo de três graus de liberdade: a) Eixos de movimento; b) Componentes (Adaptado de Milano (2018)).

A complexidade dos microcomponentes cresceu durante os anos. Na Figura 14 é mostrado o MEMS de um acelerômetro com percepção nos três eixos ortogonais em uma escala de $100\mu m$, feita por um microscópio eletrônico de varredura. É possível observar os dois eixos de movimento inercial e as placas capacitivas fixas que reagem ao deslocamento da massa de prova, assim alterando a capacitância do sistema fornecendo uma saída mensurável. O eixo Z é um pouco diferente dos demais, pois as placas fixas que são usadas para medir a capacitância ficam embaixo e em cima do sistema (KEMPE, 2011; MILANO, 2018).

Como o acelerômetro fornece dados sobre a aceleração linear nos três eixos ortogonais, quando fixado a uma parte do corpo o mesmo pode representar a aceleração desse membro. Os dados fornecidos por um acelerômetro bem calibrado possuem precisão suficiente para substituir alguns equipamentos clínicos, como uma placa de força ou câmeras com marcações infravermelho (JARCHI et al., 2018). Além disso, por se tratar de um componente de baixo custo, a disponibilização de equipamentos para os pacientes se torna viável. Os dados referentes à marcha do paciente durante o dia a dia podem complementar a análise inicial e revelar conclusões extras sobre o quadro clínico (BOUTAAYAMOU et al., 2012).

2.3.1.2 Giroscópio

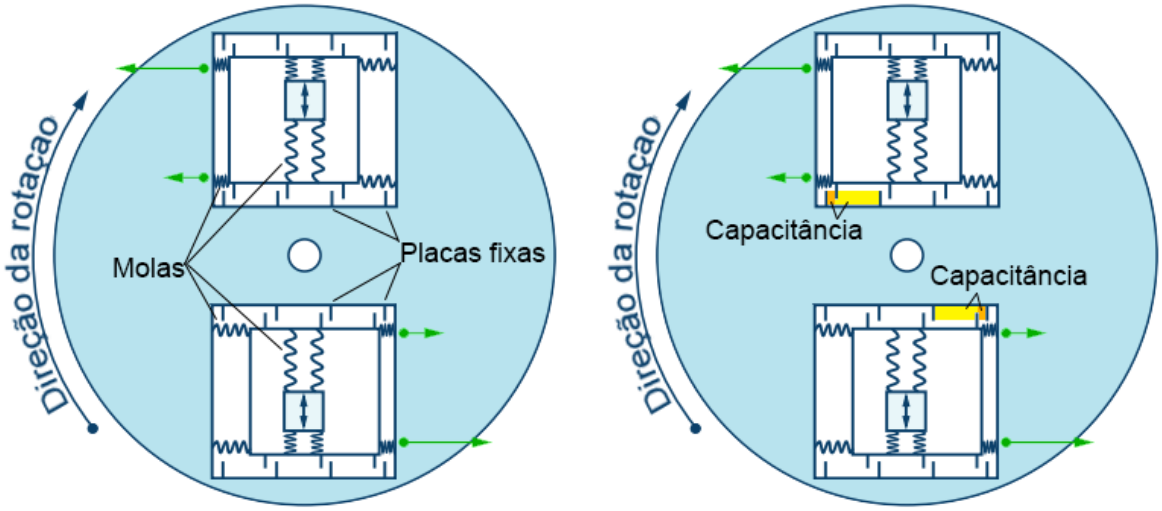
O giroscópio é similar ao acelerômetro, porém, no lugar de calcular a aceleração linear do objeto, é calculado a velocidade angular nos eixos ortogonais. O seu mecanismo inercial é similar ao acelerômetro, porém, se apoia no efeito de Coriolis (ou força inercial de Coriolis). O efeito de Coriolis constata que, ao aplicar uma rotação sobre uma massa que está se movendo em uma certa direção, uma força perpendicular a essa direção é gerada, causando o deslocamento da massa, como representado na Figura 15. Similarmente ao acelerômetro, esse deslocamento irá causar alterações na capacitância do sistema tornando possível mensurar essas alterações, e após processadas, elas corresponderão à velocidade angular da rotação que foi aplicada (KEMPE, 2011).



(a) (b)

Figura 15 – Ilustração sobre o funcionamento do efeito de Coriolis: a) Direção do movimento contínuo; b) Força de Coriolis resultante. Uma pessoa (massa de prova) se movendo em uma direção sofre uma força resultante ao ser aplicada uma rotação sobre o sistema (Adaptado de Watson (2016)).

Na Figura 16 é apresentado um esquema rudimentar de como funcionaria um sistema microeletromecânico de um giroscópio com um GDL. Um *frame* fixo com hastes fixas possui em seu interior um *frame* móvel fixado com molas. Observe que a massa de prova, presa por molas no *frame* móvel interno, está em constante oscilação em uma certa direção para medir a variação que irá ocorrer em apenas um eixo ortogonal. Quando uma rotação é aplicada sobre o sistema, a massa de prova vai se mover perpendicularmente à direção da oscilação causando a aproximação das hastes fixas promovendo a alteração na capacitância (KEMPE, 2011; WATSON, 2016).



(a) (b)

Figura 16 – Ilustração do funcionamento de um giroscópio: a) Componentes de um giroscópio; b) Diferença de capacitância gerada pela rotação. As massas de prova (em constante movimento) geram uma alteração na capacitância proveniente da força de Coriolis gerada pela rotação no sistema (Adaptado de Watson (2016)).

Analogamente ao acelerômetro, o giroscópio é muito citado em diversos trabalhos na literatura pois a maioria dos estudos fazem uso de uma unidade de medida inercial de pelo menos seis eixos, contemplando acelerômetro e giroscópio (GONZALEZ et al., 2010; ZHAO et al., 2017; DEHZANGI; TAHERISADR; CHANGALVALA, 2017). No entanto, isoladamente, os dados referentes à velocidade angular dos membros são de extrema importância para certos casos. Como sugerido por Gouwanda e Senanayake (2010), o giroscópio pode classificar uma marcha anormal de acordo com o índice de simetria, podendo substituir os sistemas ópticos de análise *in loco*.

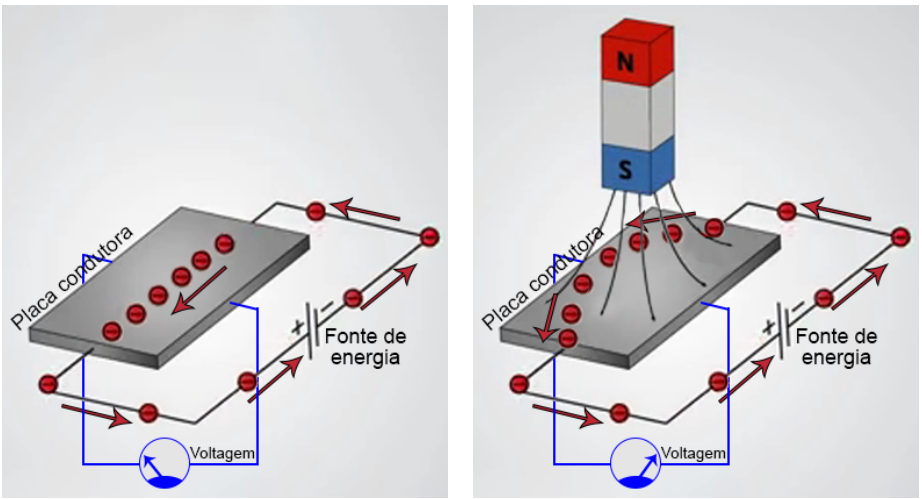
Observa-se que em diversos trabalhos os dados de velocidade angular provenientes do giroscópio produzem resultados satisfatórios clinicamente (AMINIAN et al., 2002; DOHENY; FORAN; GREENE, 2010). Como concluiu Lau e Tong (2008), mesmo quando usados em pacientes com patologia na pisada (pé equino), os sensores inerciais conseguem

distinguir entre as diferentes fases da marcha, no entanto, dependendo da patologia a posição dos sensores deve ser cuidadosamente escolhida para maximizar a diferenciação do padrão da marcha. Além disso, como evidenciado por Zebin, Scully e Ozanyan (2015) o uso de múltiplos sensores pode auxiliar na análise da marcha por meio da fusão de sensores.

2.3.2 Magnetômetro

O magnetômetro fornece a direção angular segundo o campo magnético mais próximo, quando não há interferências magnéticas próximas ao sistema microeletromecânico. A referência é o campo magnético da terra, fornecendo um referencial angular ao norte magnético da terra. Porém, muitas vezes é necessária uma calibração inicial, pois o magnetismo varia de lugar para lugar devido aos diferentes tipos de terreno, altitude, partículas carregadas pelo sol e suas interações com a magnetosfera da terra (BEEBY et al., 2013).

Diferentemente do acelerômetro e giroscópio, o magnetômetro não faz uso da inércia e trabalha com o efeito Hall, uma propriedade magnética constatando que, supondo a existência de uma placa condutora com corrente elétrica passando por ela, os elétrons irão fluir uniformemente de um lado para o outro da placa (Figura 17a), porém, ao adicionar um campo magnético próximo à placa, o fluxo de elétrons é perturbado, criando uma densidade maior de elétrons de um lado da placa (Figura 17b). Quando isso acontece, é possível medir a voltagem entre ambos os lados da placa, perpendicular a direção da corrente elétrica, como ilustrado na Figura 17. Assim, é possível realizar a medição dessa diferença de potencial elétrico entre os dois lados da placa, que vai depender da força do campo magnético e de sua direção (BEEBY et al., 2013).



(a) (b)

Figura 17 – Ilustração sobre o funcionamento de um magnetômetro com o efeito Hall em um eixo ortogonal: a) Placa condutora com corrente elétrica; b) Campo magnético adicionado ao sistema (Adaptado de Nedelkovski (2017)).

O magnetômetro também é conhecido como bússola digital e fornece a orientação angular para o norte magnético nos três eixos. Os múltiplos eixos disponíveis são usados caso a orientação do objeto mude de um estado horizontal para um estado vertical, ou para quaisquer variações entre estas. Assim, é possível manter a direção do norte magnético em qualquer posição do objeto relativo.

Na literatura, o magnetômetro é comumente citado em trabalhos para determinar a posição de um membro durante uma marcha, os pés podem apontar para fora (*out-toeing*) ou para dentro (*in-toeing*), podendo indicar uma rotação excessiva da tíbia e a possibilidade de algumas patologias ósseas. Isso pode ser calculado pelo ângulo de progressão da marcha (HUANG et al., 2016). Os joelhos podem se inclinar para fora (joelho varo) ou para dentro (joelho valgo), também indicando predisposição a algumas patologias quando associado à angulações excessivas nas caminhadas (KAWANO et al., 2007; KUN et al., 2011).

2.3.3 Dados espaço-temporais

Quando acoplados a algum objeto, os sensores inerciais possuem a característica de representar o movimento de certo ponto na estrutura do objeto, sendo ele um robô, um carro, um *smartphone* ou mesmo uma parte do corpo humano. Os dados gerados por uma unidade inercial de nove GDL vão representar o movimento espaço-temporal do membro que está sendo observado. O acelerômetro fornecerá a aceleração em três direções para cada instante do tempo. Similarmente, o giroscópio fornecerá a velocidade angular e o magnetômetro o referencial angular (KEMPE, 2011).

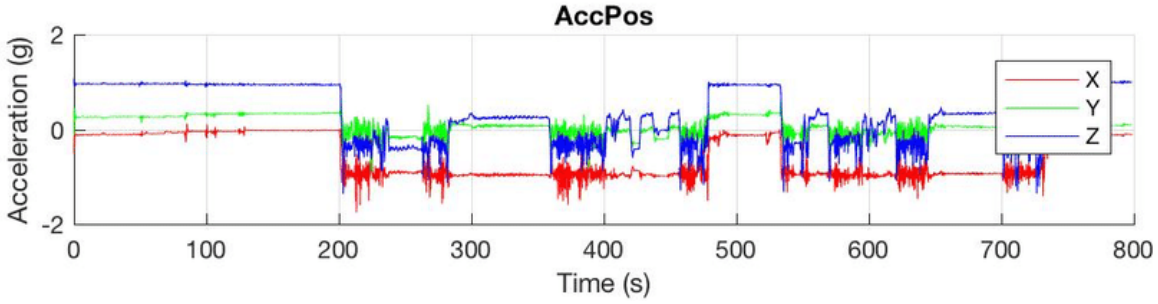


Figura 18 – Dados espaço-temporais de acelerômetro de três GDL. Três eixos ortogonais e sensibilidade de $\pm 2g$ (Adaptado de Makni e Lefebvre (2018)).

Como é apresentado na Figura 18, um simples movimento realizado em um acelerômetro de três GDL irá gerar três informações espaço-temporais diferentes que representam os três eixos (MAKNI; LEFEBVRE, 2018). Os dados brutos podem conter ruídos e estarem sujeitos ao efeito de deriva (*drift*), que são erros provocados por diferenças no desempenho do equipamento após uma calibração inicial. Essas diferenças costumam ocorrer quando o componente é exposto a mudanças de temperatura extremas ou utilizado por muito tempo (KEMPE, 2011; ABYARJOO et al., 2015).

2.4 Fusão de sensores

Metaforicamente, a fusão de sensores pode ser comparada com o cérebro humano que faz uso dos cinco sentidos: a visão; o olfato; o paladar; a audição e; o tato. O cérebro então processa as múltiplas fontes de informações e produz um resultado de maior precisão, ou de melhor interpretação para o ser humano. No entanto, mesmo que as pesquisas relacionadas a fusão de sensores tenham avançado nos últimos anos, imitar a capacidade que a mente humana possui para analisar diferentes tipos de dados simultaneamente, ainda continua sendo um grande desafio (FUNG; CHEN; CHEN, 2017).

Em especial ao conceito de internet das coisas (IoT - *Internet of Things*), um dos desafios é ser capaz de reconhecer precisamente o que está acontecendo no ambiente. No caso deste projeto, pode-se usar os sensores para observar uma parte do corpo humano, como o joelho. Em vista disso, podemos equipar essa parte do corpo com diferentes sensores. O uso de múltiplos sensores podem trazer vantagens e desvantagens, porém, com vários sensores acompanhando o mesmo membro, torna-se possível combinar a informação de todos sensores para obter uma qualidade e fidelidade maior dos dados de saída. A técnica da combinação de múltiplos sensores é chamada de fusão de sensores (LLINAS; HALL, 1998; KHALEGHI et al., 2013).

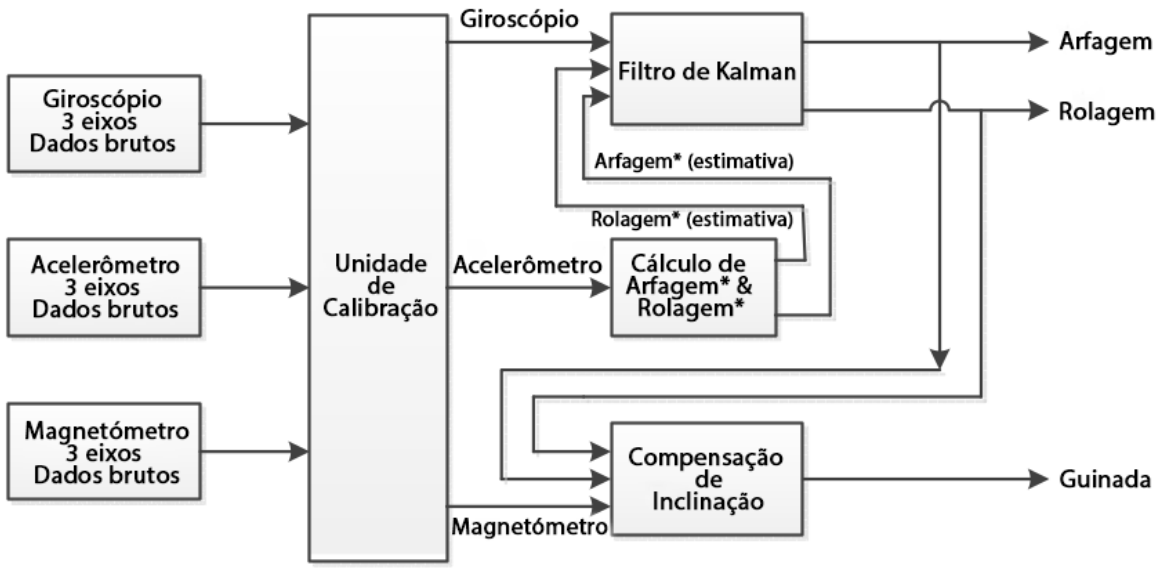


Figura 19 – Fusão dos dados de três sensores diferentes para obter dados limpos de Arfagem (*Pitch*), Rolagem (*Roll*) e Guinada (*Yaw*) (Adaptado de Abyarjoo et al. (2015)).

Na Figura 19 é ilustrado o fluxograma de dados em uma técnica de fusão de sensores. Observa-se que os dados de diferentes sensores passam por técnicas de fusão diferentes para que no final, sejam complementados, fornecendo uma melhor representação do ambiente exterior. Neste fluxograma, o autor realiza uma fusão inicial na Unidade de

Calibração, a qual irá melhorar a precisão dos dados de cada sensor utilizando os dados dos outros sensores, etapa essencial para remover erros de deriva (*Drift*). Após isso, o acelerômetro é usado para calcular os dados de arfagem (*Pitch*) e rolagem (*Roll*) iniciais. O resultado desta etapa inicial é utilizado em conjunto com os dados do giroscópio em uma fusão de sensores bastante conhecida, o Filtro de Kalman, que neste caso irá produzir os dados de arfagem e rolagem finais. Assim, na etapa final, estes dados são usados em conjunto com os dados do magnetômetro para melhorar a precisão da guinada (*Yaw*) (KHALEGHI et al., 2013; FUNG; CHEN; CHEN, 2017).

Como demonstrado em MENDES JR. et al. (2016), existem diversas aplicações para o monitoramento de atividades humanas. Nos esportes, a fusão de sensores contribui com um aumento de fidelidade e precisão no reconhecimento de dados fisiológicos, físicos e técnicos, fornecendo uma quantidade mais completa de informações para tomadas de decisão (SIIRTOLA et al., 2011; JUNG; LIM; KONG, 2013). Na biomedicina, grande parte da literatura direciona os estudos para fornecer diagnósticos mais precisos juntamente com a indicação de prognósticos (ANTINK; BRÜSER; LEONHARDT, 2014). Além disso, a reabilitação é muito beneficiada por meio do monitoramento das características físicas do quadro clínico em acompanhamento (PATEL et al., 2012).

2.4.1 Filtro de Kalman

O filtro de Kalman (FK) é resumidamente descrito como um algoritmo de estimação otimizada que pondera uma variável de interesse na presença de ruídos, erros ou incertezas. É majoritariamente usado para estimar um estado, ou conjunto de variáveis, que não podem ser medidas diretamente. Alguns exemplos de utilização são sistemas de orientação e navegação, sistemas de visão computacional e processamento de sinais (HUMPHERYS; REDD; WEST, 2012; LI et al., 2016).

É definido como um estimador recursivo, indicando que para estimar o estado atual é necessário apenas a estimação do estado anterior e as medições atuais, não sendo necessário um histórico de observações e estimações. Possui duas etapas distintas: a etapa de predição e a etapa atualização. A primeira utiliza a estimação do estado anterior para produzir a estimação do estado atual e, a última utiliza as observações do estado atual para corrigir a predição realizada e obter uma estimação mais precisa (HUMPHERYS; REDD; WEST, 2012; LI et al., 2016).

Como ilustrado na Figura 19 e na Figura 20 é possível usar o filtro de Kalman para a combinação de diferentes fontes de dados com intuito de produzir um resultado que não é possível de ser calculado pelos dados puros (direto da fonte), ou então, realizar correções de erros e ruídos existentes nos dados brutos (LI et al., 2016; GRAVINA et al., 2017). Os sensores inerciais estão comumente propensos a apresentar erros de ruído. Quando expostos a temperaturas elevadas ou em uso por muito tempo costumam apresentar

erros de deriva (*drift*), que são pequenos erros de aceleração ou velocidade angular sendo integrados progressivamente causando erros em escala maior na velocidade ou posição final. Estes erros normalmente são solucionados por meio de correções que são aplicadas ao combinar os dados do acelerômetro e do giroscópio em um modelo matemático, como o filtro de Kalman (WATSON, 2016; GRAVINA et al., 2017; BÖTZEL et al., 2018).

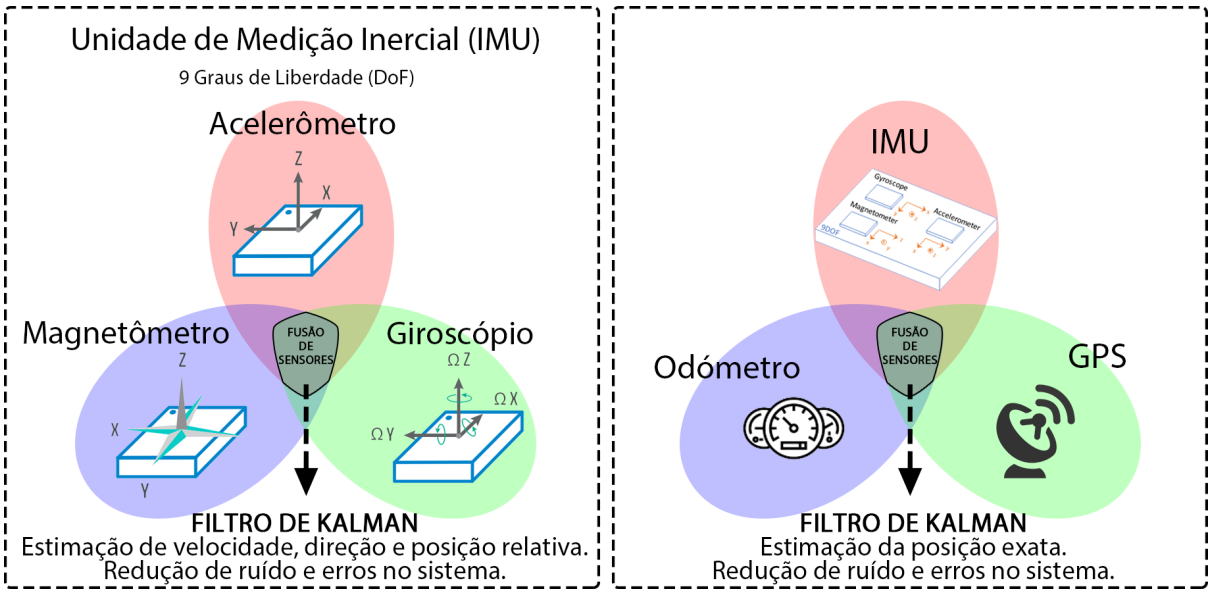


Figura 20 – Ilustração da fusão de sensores com diferentes fontes de dados.

Segundo o autor, o filtro de Kalman possui certas limitações, pois é necessário que o sistema seja linear e possua ruído branco gaussiano (*Gaussian white noise*) (KALMAN, 1960). Para sanar tais limitações existem variações: O filtro de Kalman Estendido (FKE) e; O filtro de Kalman *Unscented* (FKU). Ambos surgiram para apresentar soluções para sistemas não lineares. O FKE realiza a linearização do sistema não linear por meio da aproximação de Taylor de primeira ordem. Por se tratar de uma aproximação linear, o FKE não apresenta bons resultados quando o sistema não linear não possui muita compatibilidade com sua versão linear. Para isso, existe o FKU que utiliza uma amostra dos estados para escolher um conjunto de pontos (pontos *sigma*), usados para realizar uma melhor estimativa do estado e de sua covariância (KHALEGHI et al., 2013).

2.5 Plataforma Arduino

Os componentes utilizados nesta pesquisa são compatíveis com a plataforma Arduino, uma tecnologia nascida na Itália em que os criadores a idealizaram como de baixo custo, aberta e também para proporcionar facilidade nas experimentações. O sucesso foi tanto que Arduino se tornou a placa que permitia facilmente a programação de diferentes componentes microeletrônicos conectados. Desde então a família de componentes microeletrônicos disponíveis e compatíveis com a plataforma vem crescendo, como os componentes

usados por este projeto (BARRETT, 2020). Além disso, a plataforma Arduino e opções similares em código aberto possibilitaram um grande impulso no conceito de internet das coisas (IoT - *Internet of Things*). Este projeto pode ser enquadrado neste conceito e pode também ser a base para muitas aplicações deste tipo (BARRETT, 2021). Mais detalhes sobre a composição do *hardware* adquirido são encontrados na Seção 3.4.

A linguagem de programação mais utilizada por estes microcontroladores é C, ou uma variante desta como C++ ou a própria linguagem de programação Arduino (INO) que é um subconjunto da linguagem C++. Este projeto é programado em INO e C++ por meio do ambiente de desenvolvimento chamado Arduino IDE, que fornece um facilidade no momento de programar, compilar e carregar o código no microcontrolador escolhido (BARRETT, 2020). O sistema desenvolvido (Subseção 4.3.1) no dispositivo é simples e foi feito com apenas um intuito: o de enviar dados inerciais de múltiplos sensores, componentes da plataforma TinyCircuits.

2.6 Trabalhos relacionados

Até o presente momento e no limite da busca realizada, não foram encontradas aplicações em código aberto que apresentem as mesmas características deste projeto. Existem esforços similares no trabalho de Delp et al. (2007) com um sistema em código aberto para criação de modelos músculo-esqueléticos capazes de gerar simulações de movimento nomeado de OpenSim. Após isto, Rosa, Silva e Matias (2015) fornecem um *framework* que faz uso do OpenSim, porém é necessário que os dados inerciais sejam alimentados. Recentemente, os trabalhos de Vajs et al. (2020), Lavikainen et al. (2021) e Stanev et al. (2021) apresentaram soluções similares para análise de dados inerciais em tempo real com foco na análise da marcha e reabilitação.

O trabalho de Vajs et al. (2020) é uma proposta em código aberto, porém não foi encontrado um repositório de acesso público para que seja feita uma análise da arquitetura adotada. Os repositórios abertos encontrados aparentemente não possuem foco na arquitetura de sistema (DELP et al., 2007; LAVIKAINEN et al., 2021; LÉTOURNEAU et al., 2021). Létourneau et al. (2021) apresenta uma proposta para importar, visualizar, gerenciar, processar e exportar dados de unidades inerciais, porém só fornece compatibilidade com poucos dispositivos e o algoritmo de processamento é fixo pois o intuito do sistema é a actigrafia, monitorar ciclos de atividade e descanso humano em um período de tempo.

Os estudos citados realizam a análise por meio de sensores proprietários e são focados em seus objetivos funcionais, assim diferem desta proposta que possui o intuito de fornecer também o protótipo de um dispositivo para transmissão de dados inerciais em código aberto e de baixo custo. O trabalho de Lavikainen et al. (2021) é apenas compatível

com tecnologias da Xsens e Delsys que possuem custo elevado. Em contrapartida, o trabalho de Létourneau et al. (2021) utiliza dispositivos como o Actigraph GTX3, um *smartwatch* e também fornece uma alternativa de coleta dos dados inerciais em código aberto chamada de OpenIMU-MiniLogger. O diferencial do trabalho de Létourneau et al. (2021) está na organização e visualização dos dados importados, sendo possível uma personalização por grupo de participantes, participantes, grupos de dados e resultados, além de possibilitar a exportação dos dados gerados para CSV.

O sistema OpenSim de Delp et al. (2007) possui uma ferramenta de trabalho nomeada de OpenSense que permite a análise de movimento com sensores inerciais por meio da leitura e conversão dos dados em um formato de orientação único. Após isso o usuário pode registrar o sensor com o segmento corporal que está atrelado, possibilitando no final o processamento dos ângulos das juntas corporais. Os dados no entanto devem ser importados ou integrados por meio de uma API personalizada, pois os comandos são restritos a linhas de comando e codificação com Matlab e Python. A ferramenta é compatível com dispositivos da marca Xsens, porém existe também a compatibilidade para dados no formato APDM o que possibilita que qualquer aplicação construa seu próprio conversor de dados para utilizar o sistema (OPENSENSE, 2021). O OpenSim é bem conhecido e frequentemente utilizado na literatura até o presente momento, em contrapartida a ferramenta OpenSense é mais recente e não foram encontrados publicações com seu uso (ZAMAN et al., 2021; YU et al., 2020; WEI et al., 2013; YU et al., 2020).

O trabalho BROWN et al. (2021) demonstra o processo de refatoração de um sistema legado para um modelo arquitetura limpa, solucionando diversos problemas previamente existentes e provendo uma melhor manutenibilidade e legibilidade do código atual. Martínez Z et al. (2021) discutem os benefícios da arquitetura limpa em projetos Java contrastando com outras arquiteturas e apresentando um modelo de organização com o gerenciador de dependências Graddle. A linguagem e o *framework* adotados no trabalho de Boukhary e Colmenares (2019) são os mesmas deste projeto, porém a proposta deles é de utilizar a arquitetura limpa como solução estrutural geral para aplicações móveis em Flutter.

A proposta de Boukhary e Colmenares (2019) é mais focada em uma solução específica para a atualização da tela e componentes em aplicativos feitos com Flutter. Esta característica é chamada de gerenciamento de estados (*state management*) e existe diversas bibliotecas que fornecem suas próprias soluções. O diferencial é a utilização da separação de responsabilidades e camadas da arquitetura limpa como uma solução estrutural nativa para a atualização do estado.

Porém, algumas decisões adotadas em Boukhary e Colmenares (2019) não se mantém fieis aos princípios definidos por Martin (2017), pois o estudo produz uma biblioteca que ajuda na estruturação de aplicativos com a estrutura de arquitetura limpa, ao optar

pela utilização desta biblioteca é necessário se comprometer com a dependência em todas suas camadas correndo o risco de depreciação no caso em que a biblioteca pare de fornecer suporte. O mesmo acontece na criação de componentes nas diversas camadas, pois todas abstrações são pré fornecidas como moldes na biblioteca forçando o aplicativo a se comprometer totalmente com a dependência.

Os trabalhos de Kimura (2019) e Bueno (2021) utilizam arquitetura limpa como solução arquitetural para seus sistemas. Bueno (2021) propõe uma estrutura mais orientada a dados visto que a camada de apresentadores (*Presenters*) é responsável pela gerência de estados e dependente de uma camada nomeada de dados que é responsável pelos casos de uso e regras de negócio. Martin (2017) separa estas responsabilidades na camada de aplicação e na camada de entidades, sendo a camada de aplicação responsável pela implementação dos casos de uso e regras de negócio da aplicação, já a camada de entidades não conhece os casos de uso e é responsável apenas pelas regras de negócio.

O trabalho de Kimura (2019) apresenta uma estruturação mais fiel à arquitetura limpa e inclui também toda coleta e análise de requisitos do sistema a ser desenvolvido. Porém, a única ressalva é a não utilização dos apresentadores (*Presenters*) no desenvolvimento da API agregando toda responsabilidade de apresentação do resultado nos controladores, criando baixa coesão neste componente. Martin (2017) deixa subjetivo aos leitores a relação entre os controladores e apresentadores porém ambos são necessários. Mesmo assim, em um de seus exemplos nota-se que o controlador é dependente do apresentador (MARTIN, 2017, p. 84), já em outros modelos ambos são independentes (MARTIN, 2017, p. 196).

Como a proposta deste projeto é fornecer um repositório aberto a contribuição, a arquitetura desacoplada adotada facilita a contribuição e análise de código, permitindo uma futura integração com a API do sistema OpenSim, como também de outros sistemas. A conversão e exportação de dados do formato adotado por este projeto para formatos adotados pelas APIs de outros sistemas também pode ser viabilizada facilmente, sem a necessidade de alterações nas camadas internas da arquitetura. Acredita-se ser de extrema importância para o futuro de pesquisas nas áreas relacionadas, viabilizar tecnologias de fácil acesso que favorecem a prática e a experimentação, servindo como porta de entrada para mais pesquisadores tanto na área de ciência da computação como na área de saúde.

3 INTRACS - Inertial Tracking Computing System

O INTRACS (Inertial Tracking Computing System) é um aplicativo em código aberto, proposto nesta pesquisa para realizar a coleta e processamento de dados de múltiplos sensores inerciais em tempo real. A coleta de dados é realizada utilizando tecnologias de transmissão sem fio. O protótipo inicial faz uso da tecnologia *Bluetooth*, porém a arquitetura do sistema permite a integração com qualquer tecnologia de transmissão como *Wi-Fi*, por exemplo. Os dados dos múltiplos sensores são transmitidos e coletados em tempo real pelo aplicativo, logo após esses dados são estruturados em um formato padrão e redirecionados ao método de processamento selecionado. Ao receber os dados neste formato, os métodos possuem a liberdade de retornar os dados processados em um formato customizado, assim, estes dados processados são exibidos na tela do aplicativo em paralelo com os dados inerciais puros (não tratados). A Figura 21 ilustra a ideia principal do funcionamento do sistema, recebendo os dados, processando e ao final do fluxo, exibindo na tela para o usuário.

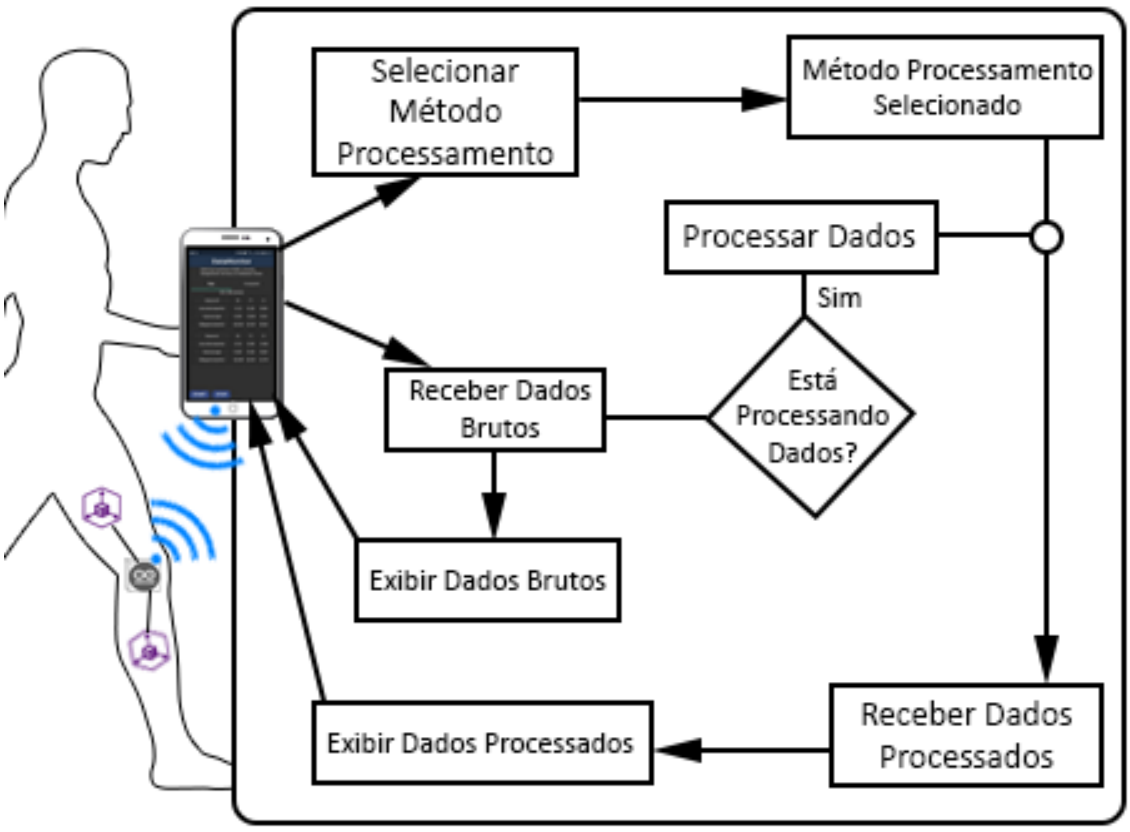


Figura 21 – Fluxo de atividades em alto nível do INTRACS.

3.1 Propósito do projeto

Um dos objetivos no desenvolvimento do aplicativo é fornecer um repositório em código aberto em que a contribuição de métodos de processamento seja simples e não demande do desenvolvedor (ou pesquisador) um profundo conhecimento da estrutura e tecnologias utilizadas no aplicativo. Para isto, usa-se a arquitetura limpa como solução estrutural do sistema juntamente com diversos padrões de projeto para manter o desacoplamento e divisão de responsabilidades entre os componentes. A adoção do padrão *Strategy* foi essencial para a implementação dos métodos de processamento customizados, visto que busca-se uma contribuição contínua da comunidade com métodos diversos, que podem incluir fusões de sensores, aprendizado de máquina, integração com APIs de terceiros, entre outros.

Este projeto busca se manter o mais fiel possível à não adoção de *frameworks* nas camadas de adaptadores e camadas internas. O comprometimento com *frameworks* e bibliotecas de terceiros só existe nas camadas externas. A Figura 22 ilustra que as dependências adotadas nas camadas internas e na camada de adaptadores são apenas a linguagem escolhida, porém nas camadas externas existe o compromisso com as dependências, com os sistemas de armazenamento escolhidos, com a tecnologia de comunicação sem fio escolhida (e sua respectiva biblioteca) e por fim com o *framework* de visualização escolhido para o desenvolvimento do aplicativo.

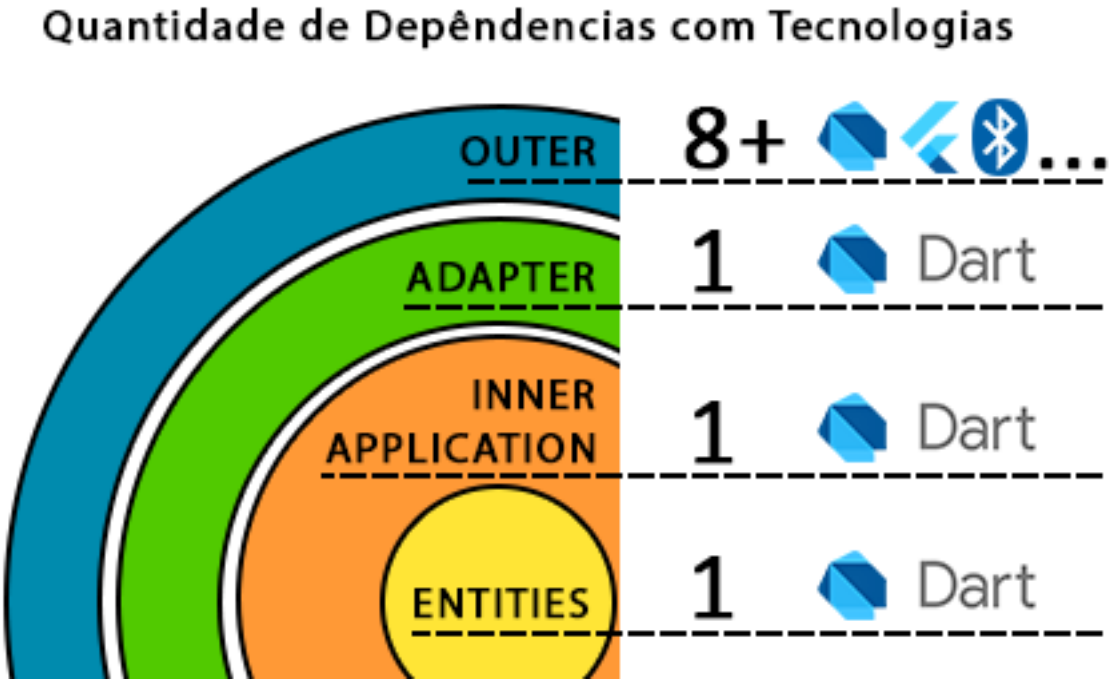


Figura 22 – *Frameworks* e dependências adotados nas camadas do projeto.

Além disso o projeto também fornece um protótipo de dispositivo capaz de enviar dados por *Bluetooth* de múltiplos sensores inerciais. O apresentado utiliza dados de

dois sensores porém é capaz de suportar até 32 unidades inerciais de nove eixos. Os dados de cada sensor são enviados sequencialmente e identificados em cada mensagem. Este dispositivo também é disponibilizado em código aberto e possui uma montagem e configuração bem simples. O dispositivo possui um tamanho reduzido o que torna viável a criação de vestimentas e suportes para acoplar os sensores em diversas partes do corpo humano. A aquisição dos componentes é fácil e o custo é relativamente baixo quando comparado com as alternativas no mercado.

3.2 Tecnologias utilizadas

Para atender um dos objetivos deste projeto, que é fornecer um repositório em código aberto, uma base de código única fornece maior simplicidade para buscar contribuições externas no projeto. Com isso se fez necessária a escolha de uma linguagem de programação multiplataforma. Existem muitas alternativas populares no mercado atualmente como JavaScript com React Native, C# com Xamarin e Dart com Flutter. Dentre as possibilidades, este projeto optou por Dart e Flutter. Flutter é o *framework* de interface do usuário recomendado a ser usado com Dart. O desacoplamento fornecido pela estrutura arquitetônica implementada neste projeto fornece a possibilidade da substituição deste *framework* por outras alternativas como AngularDart, OverReact ou VueDart. A estrutura interna se mantém pura em Dart com a dependência do Flutter apenas nas camadas externas. A Figura 23 mostra que a troca de tecnologia implica somente no desenvolvimento da camada externa afetada, mantendo as camadas e componentes laterais ou internos consistentes, sem sofrer alteração.

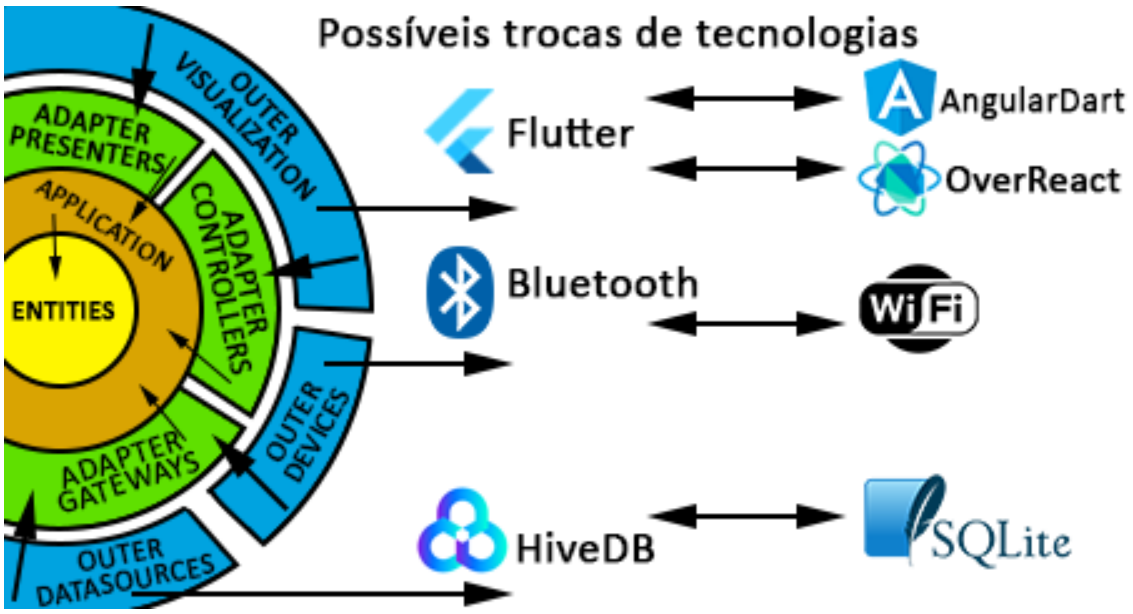


Figura 23 – Tecnologias utilizadas nas camadas externas e possíveis substituições.

O projeto necessitava de um foco grande em características de orientação a objetos e também de uma linguagem de fácil adaptação e entendimento, para que a codificação de um método de processamento (algoritmo) não se torne complicada por questões de sintaxe. A linguagem de programação Dart supriu todas estas necessidades por ser uma linguagem com conceitos e sintaxe próximos aos de outras linguagens tipadas populares atualmente, como Java, C# e TypeScript, tornando a curva de aprendizado mais suave. Consequentemente à decisão de usar Dart na estrutura interna, o aplicativo faz uso do Flutter, que é um *framework* em código aberto que vem se tornando muito popular ao longo dos últimos anos com mais de 900 contribuidores diferentes em seu repositório oficial. Além disto ambas tecnologias são fornecidas pela Google e mantidas pela própria empresa e pela comunidade externa.

O aplicativo em seu estado atual utiliza uma biblioteca Dart de armazenamento NoSQL que trabalha com um modelo de banco de dados em chave-valor chamada HiveDB. Além disto também foi escolhida uma biblioteca para realizar a comunicação sem fio por *Bluetooth Low Energy* (BLE) entre o aplicativo e dispositivo inercial, a FlutterBlue. Estas dependências só existem nas camadas externas e podem ser facilmente substituídas por um banco de dados relacional, ou por uma biblioteca para realizar a comunicação por *Wi-Fi* por exemplo.

O aplicativo em Flutter só existe na camada de visualização e foi utilizada a bilioteca GetX para realizar o gerenciamento de estados, como também a biblioteca get_it para o gerenciamento de dependências cuja responsabilidade é implementar a injeção de dependências, parte importante da arquitetura limpa. A Figura 24 lista a localização destas dependências nas camadas externas da arquitetura do projeto.

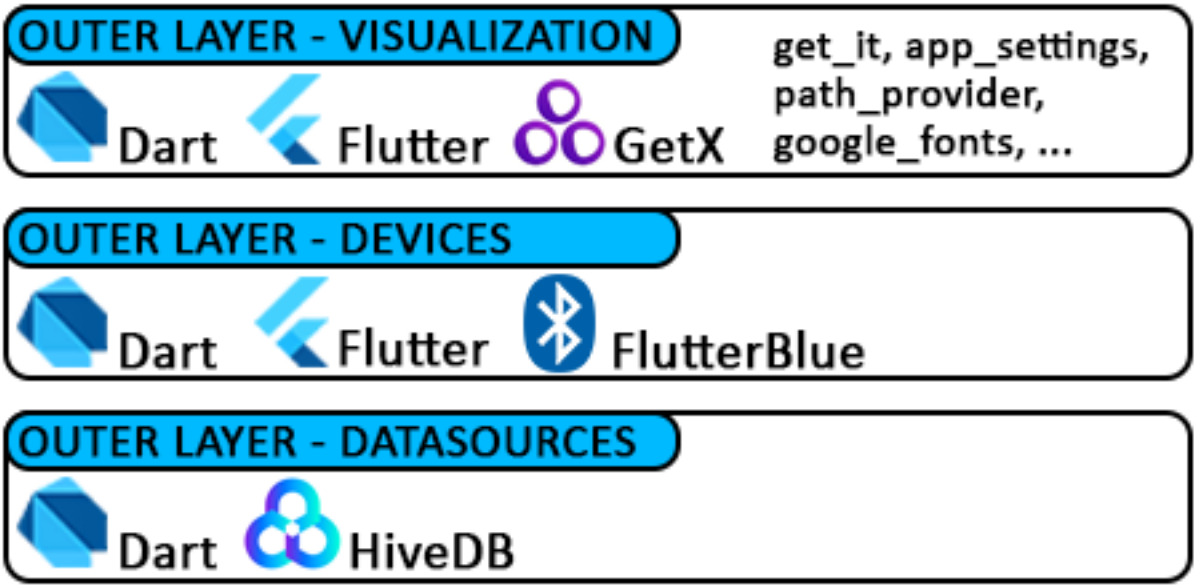


Figura 24 – Dependências com tecnologias utilizadas nas camadas externas.

O protótipo multisensor para transmissão de dados inerciais foi feito com componentes de tamanho reduzido compatíveis com a plataforma Arduino, utilizou-se os componentes da empresa TinyCircuits por conta da sua praticidade de montagem, compatibilidade com múltiplos sensores inerciais (suportando até 32 unidades de medição inercial de nove eixos), seu tamanho reduzido e também sua flexibilidade com sua plataforma chamada *Wirelings*, que fornece diversos sensores isoladamente com a conexão através de um cabo maleável, o que possibilita a criação de vestimentas personalizadas podendo afixar os sensores em diversas partes do corpo facilmente.

A Figura 25 ilustra a portabilidade desta plataforma e sua flexibilidade para criação de vestimentas. É importante ressaltar que por ser compatível com a plataforma Arduino, o dispositivo oferece inúmeras possibilidades de personalização embora tenha sido codificado apenas para o envio de dados inerciais por *Bleutooth*. Os detalhes dos componentes presentes estão detalhados na Seção 4.3.

Figura 25 – Demonstração de protótipos de vestimentas que podem ser criadas com o dispositivo de tamanho reduzido e alta portabilidade.

3.3 Padrão arquitetônico do projeto

O foco principal no desenvolvimento deste projeto é na arquitetura aplicada na estrutura do código fonte para proporcionar facilidade em contribuições de métodos de processamento pela comunidade e, em paralelo fornecer fácil legibilidade, manutenibilidade e longevidade ao projeto. A arquitetura limpa apresentada no Subseção 2.2.1 foi adaptada parcialmente para se adequar aos requisitos deste projeto. Todos os princípios propostos por Martin (2017) foram respeitados juntamente com os princípios apresentados na Seção 2.1. Assim como na arquitetura limpa, a arquitetura deste projeto é dividida em quatro camadas: Camadas externas; camadas intermediárias; e duas camadas internas. A Figura 26 ilustra as camadas existentes na arquitetura junto com a regra de dependência, em uma disposição mais linear quando comparada com a proposta por Martin (2017), da Figura 6.

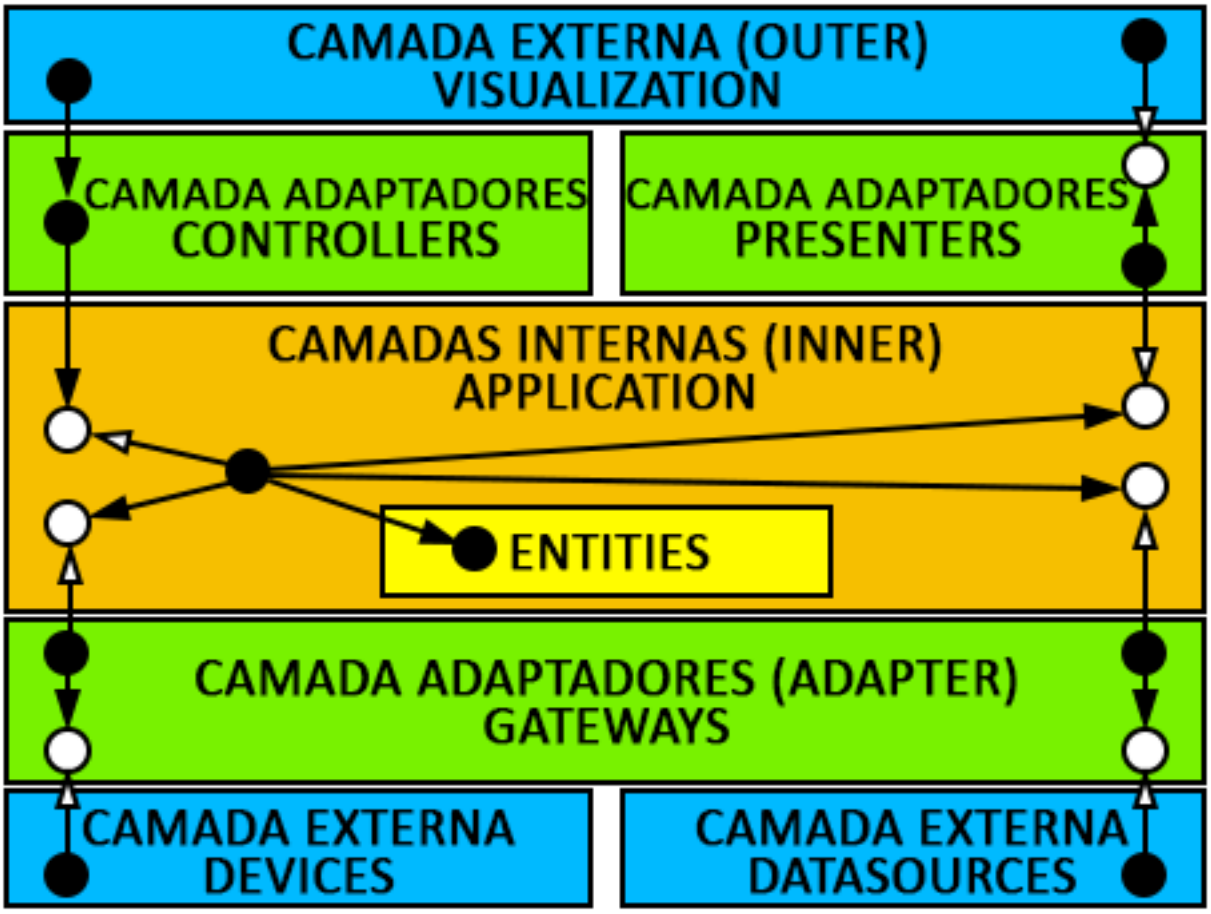


Figura 26 – Organização das camadas da arquitetura do projeto.

As próximas seções detalham todas as características da arquitetura, tendo como base um dos principais casos de uso do projeto que é apresentado na Subseção 3.3.1 e inicia-se na Subseção 3.3.2 com uma visão geral de como a arquitetura funciona entre as camadas, detalhando o funcionamento do clique de um botão até a exibição da resposta visualmente. Após isto todas as camadas são detalhadas separadamente iniciando pelas

camadas internas que possuem menos dependências na Subseção 3.3.3, seguindo pelas camadas intermediárias que dependem somente das camadas internas na Subseção 3.3.4 e finalizando nas camadas externas na Subseção 3.3.5 que possuem o maior grau de complexidade e dependência com fatores externos. Após isto alguns detalhes importantes existentes na inicialização do sistema são explicados na Subseção 3.3.6.

3.3.1 Funcionalidade: Processamento customizado em tempo real

O aplicativo móvel permite que o usuário selecione um dentre diferentes métodos de processamentos existentes para que os dados inerciais sendo coletados sejam processados em tempo real. Esses métodos de processamento podem ser uma simples fusão de sensores para calcular a orientação de cada sensor ou métodos mais complexos para calcular os ângulos de extensão e flexão do joelho durante uma caminhada ou até mesmo criar uma projeção 3D baseada no posicionamento, orientação e movimento dos sensores.

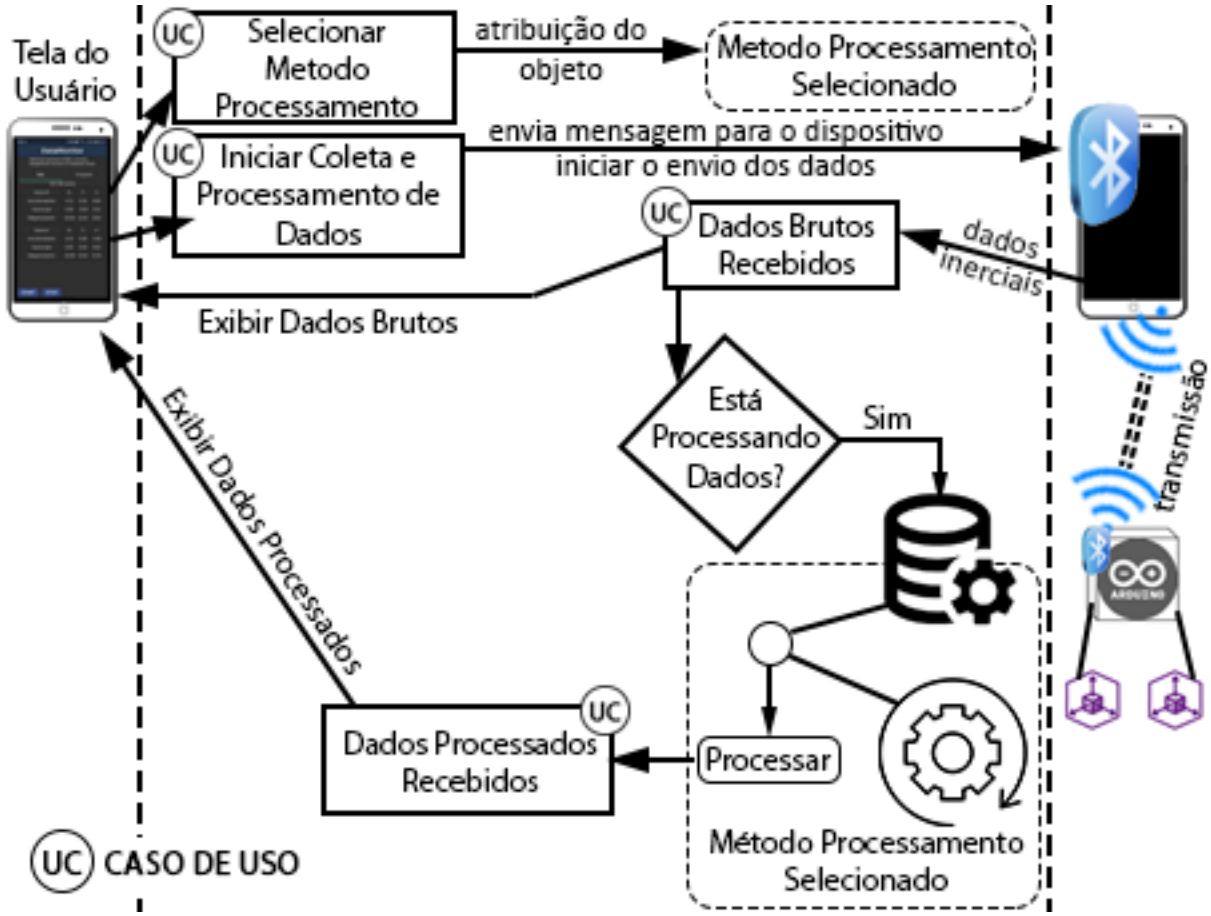


Figura 27 – Diagrama de atividade da funcionalidade com quatro casos de uso.

A Figura 27 apresenta o diagrama de atividades da funcionalidade em questão. O usuário inicia selecionando um dos métodos de processamento existentes (Figura 28 tela A) sendo redirecionado para a tela de inicio de coleta e processamento (Figura 28 tela B). Após isso o usuário decide iniciar a coleta e processamento fazendo com que o

aplicativo envie o comando para o dispositivo inercial para iniciar o envio de dados inerciais. Ao receber os dados o aplicativo já envia os dados inerciais brutos para a exibição na tela (Figura 28 tela C) e, como um método de processamento foi selecionado, também redireciona estes dados para o método de processamento que ao final do algoritmo retorna os dados processados que são enviados também para a tela (Figura 28 tela D).

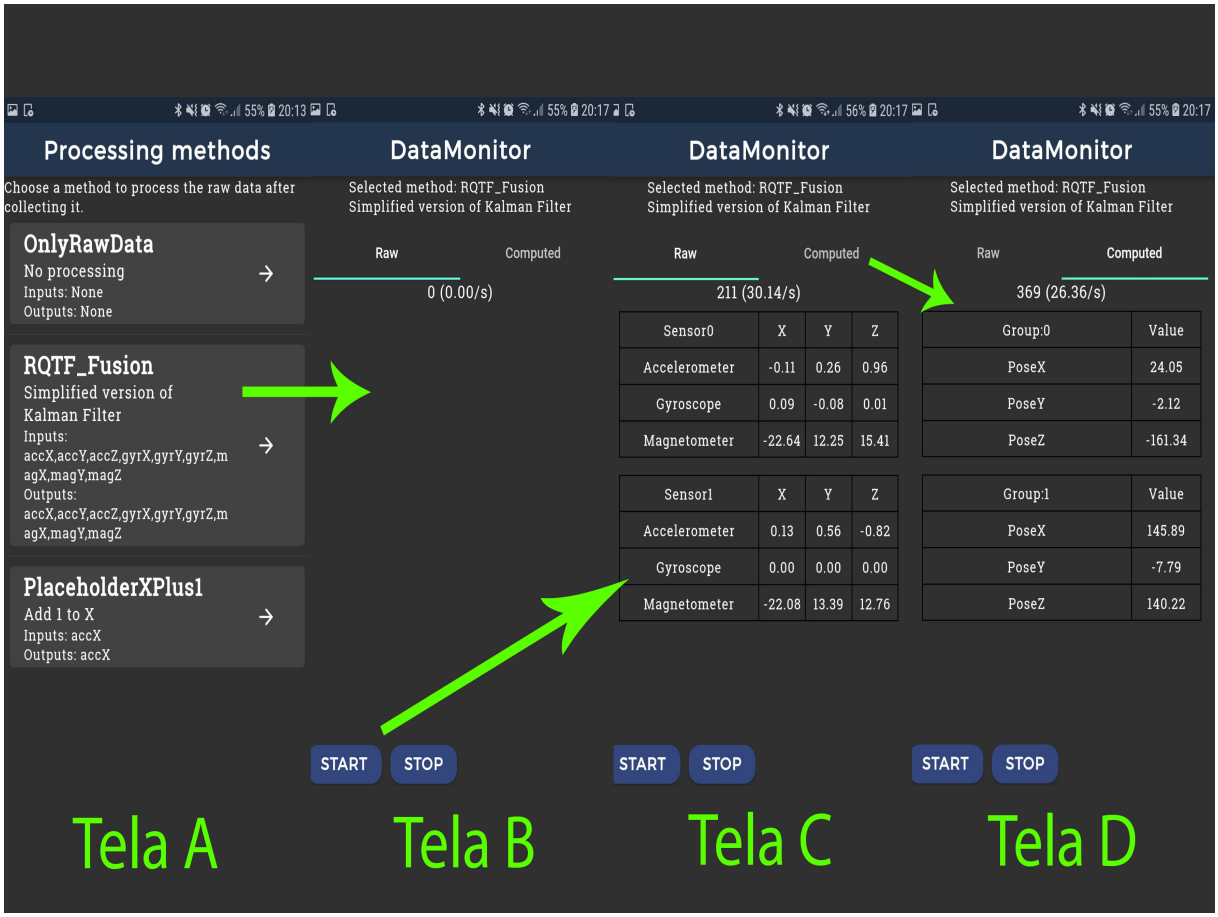


Figura 28 – Telas do aplicativo que envolvem a funcionalidade.

3.3.2 Fluxo de controle

A regra de dependência (Subseção 2.2.1.1) entre as camadas é respeitada nessa arquitetura. O fluxo de controle é mais claro na disposição ilustrada na Figura 26 e é possível visualizar o clique de um botão acionando um controlador (*Controller*), passando por um caso de uso (*Use Case*) específico que busca as informações e envia para o apresentador (*Presenter*) exibir na interface do usuário. É importante notar que o fluxo se inicia pela camada externa (pelo *entrypoint*, Subseção 3.3.6), passa pelas camadas internas buscando informações na camada externa quando necessário e termina o fluxo enviando novamente a camada externa para visualização.

Como cada camada e componentes possuem suas responsabilidades bem definidas, o fluxo de controle acontece naturalmente e o desenvolvimento é simplificado, pois os

componentes são forçados a serem pequenos e coesos. Todas as interações entre camadas devem respeitar a regra de dependência, portanto a interação de uma camada interna para uma camada externa deve ser feita por meio da inversão de dependência (Subseção 2.1.3). Já o contrário, de camadas externas para internas, pode ser feito por meio de uma referência direta ou abstração, o que cria a dependência com a camada interna.

A funcionalidade inicia na tela A da Figura 28 com o usuário selecionando um método de processamento. A Figura 29 ilustra a relação de dependência entre os componentes iniciando no botão acionado pelo usuário na camada de visualização, passando pelo controlador na camada de adaptadores e chegando na implementação do caso de uso. A camada de visualização que contém o botão a ser clicado (seleção de um método de processamento) mantém uma referência direta com o controlador. O botão aciona o método correspondente no controlador enviando os parâmetros necessários. Já na camada de adaptadores o controlador mantém a referência dos casos de uso que pertencem ao seu escopo. Esta referência é feita pelas abstrações e as instâncias concretas são atribuídas em tempo de inicialização na injeção de dependência, o controlador então passa o controle para o caso de uso, chamando o seu método de execução chamado "*call()*". O caso de uso possui toda implementação necessária para a ação de seleção do método de processamento que foi selecionado e passado por parâmetro.

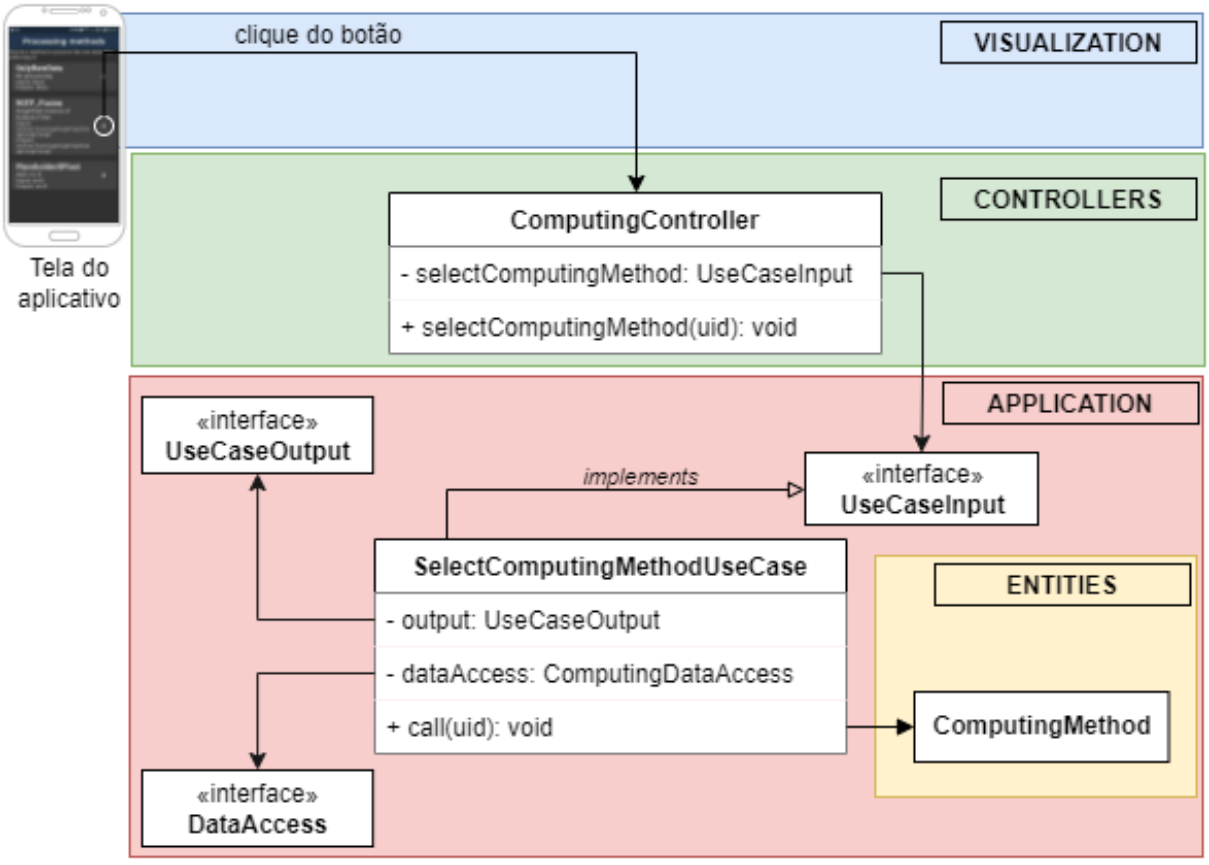


Figura 29 – Fluxo de controle da ação do usuário até a implementação do caso de uso.

Após o controle da chamada estar na implementação do caso de uso como ilustrado na Figura 29, este caso de uso em particular deve enviar para camada externa um comando dizendo que o método de processamento foi selecionado. Esta ação irá permitir a preparação do padrão *Strategy* com a instância concreta do método selecionado. Na Figura 30 é ilustrado o princípio da inversão de dependência entre as camadas internas e externas. O princípio é aplicado na relação entre o caso de uso (*UseCase*) e o repositório (*ComputingRepository*), como também entre o repositório e a fonte de dados (*ComputingDataSource*). O retorno é o método que foi corretamente selecionado e então o caso de uso passa o controle para o apresentador (*Presenter*) que, por conseguinte, irá passar o controle para uma implementação da visão (*View*). Neste ponto o aplicativo já possui todos os dados e pode exibir a tela de coleta e processamento.

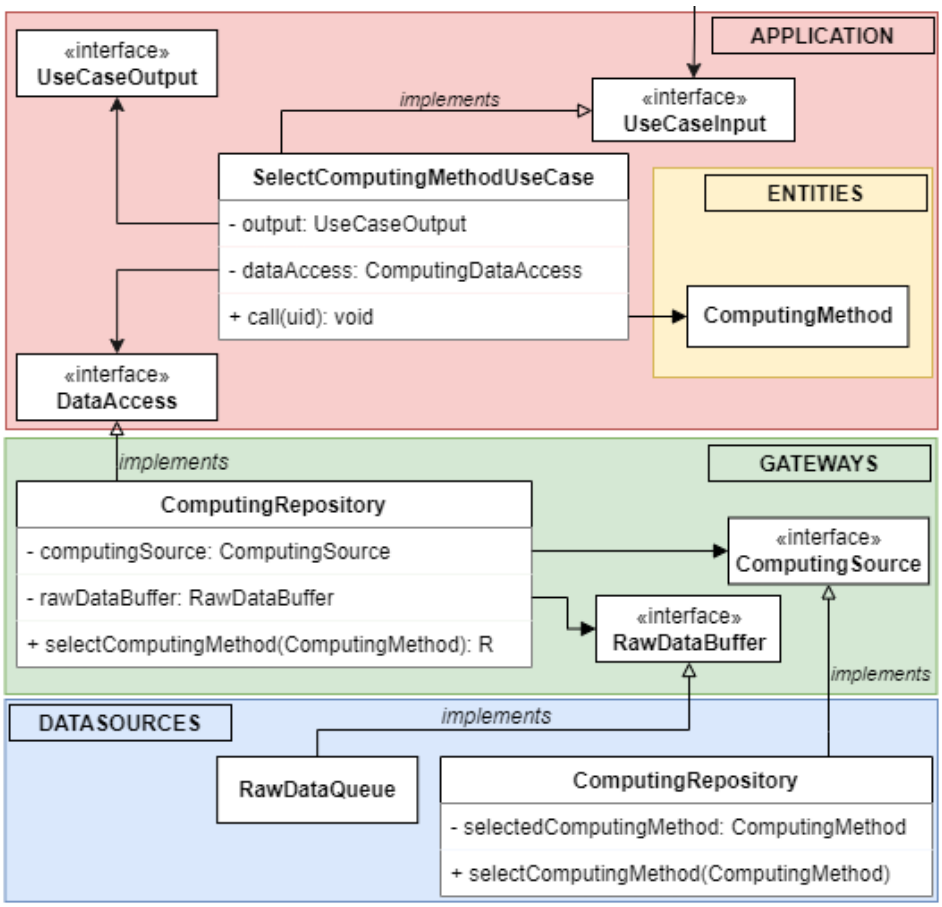


Figura 30 – Relação do caso de uso com camadas externas de dados por meio da inversão de dependência.

Na Figura 28 tela B o próximo passo do usuário é a decisão de iniciar a coleta e processamento, por meio do botão *Start* cujo fluxo completo de controle está ilustrado na Figura 31. Este botão irá repetir o mesmo processo anterior que é acionar um controlador que executa um caso de uso e, por inversão de dependência irá enviar o comando para iniciar a coleta e processamento de dados para o repositório. O repositório irá repassar o comando para implementação correta na camada externa de dados, esta camada possui

a implementação do dispositivo *Bluetooth* e, ao receber este comando deverá enviar uma mensagem para o dispositivo que irá iniciar o envio dos dados inerciais. O retorno completo desta chamada para visualização é apenas uma confirmação de que o comando foi executado com sucesso, assim o aplicativo pode se preparar para iniciar a exibição dos dados brutos e processados em tempo real (tela C e tela D na Figura 28).

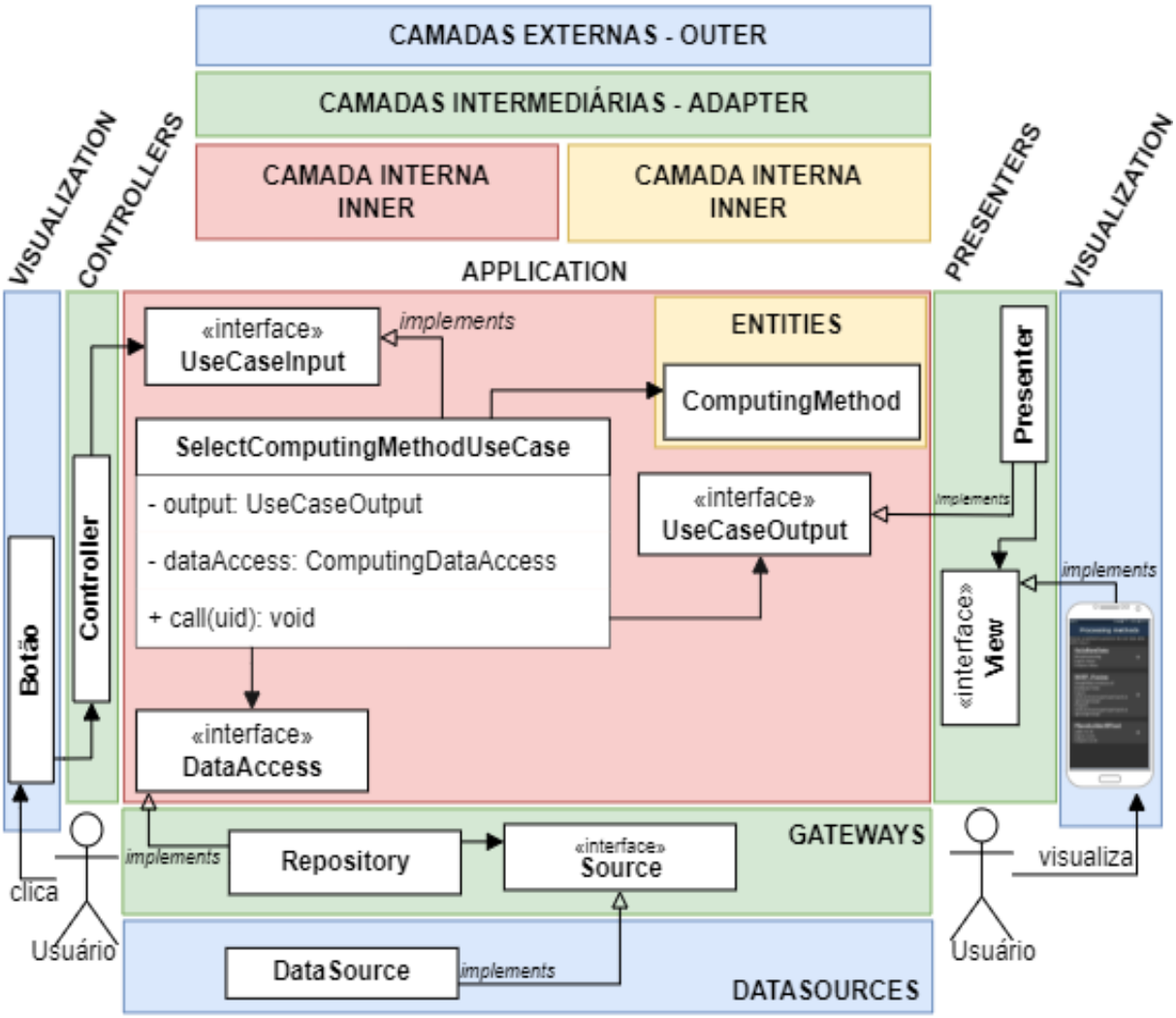


Figura 31 – Fluxo de controle completo da ação do usuário até o retorno do resultado.

A classe que implementa a comunicação com o dispositivo inercial fica na camada externa. Assim que o dispositivo inicia o envio de dados inerciais, esta classe: recebe as mensagens em bytes (20 bytes por mensagem) e; realiza a conversão dos dados para os valores inerciais na estrutura necessária para ativar um controlador que existe na camada de adaptadores. Este controlador irá redirecionar o fluxo para o caso de uso respectivo responsável por implementar a regra de negócio de quando se recebe dados inerciais do dispositivo, como ilustrado na Figura 32. Observe que nesta situação em específico o caso de uso não foi ativado por uma ação do usuário e sim por uma ação do dispositivo externo. Por esta razão optou-se por diferenciar o nome dos controladores, sendo este nomeado de controlador de evento (*EventController*).

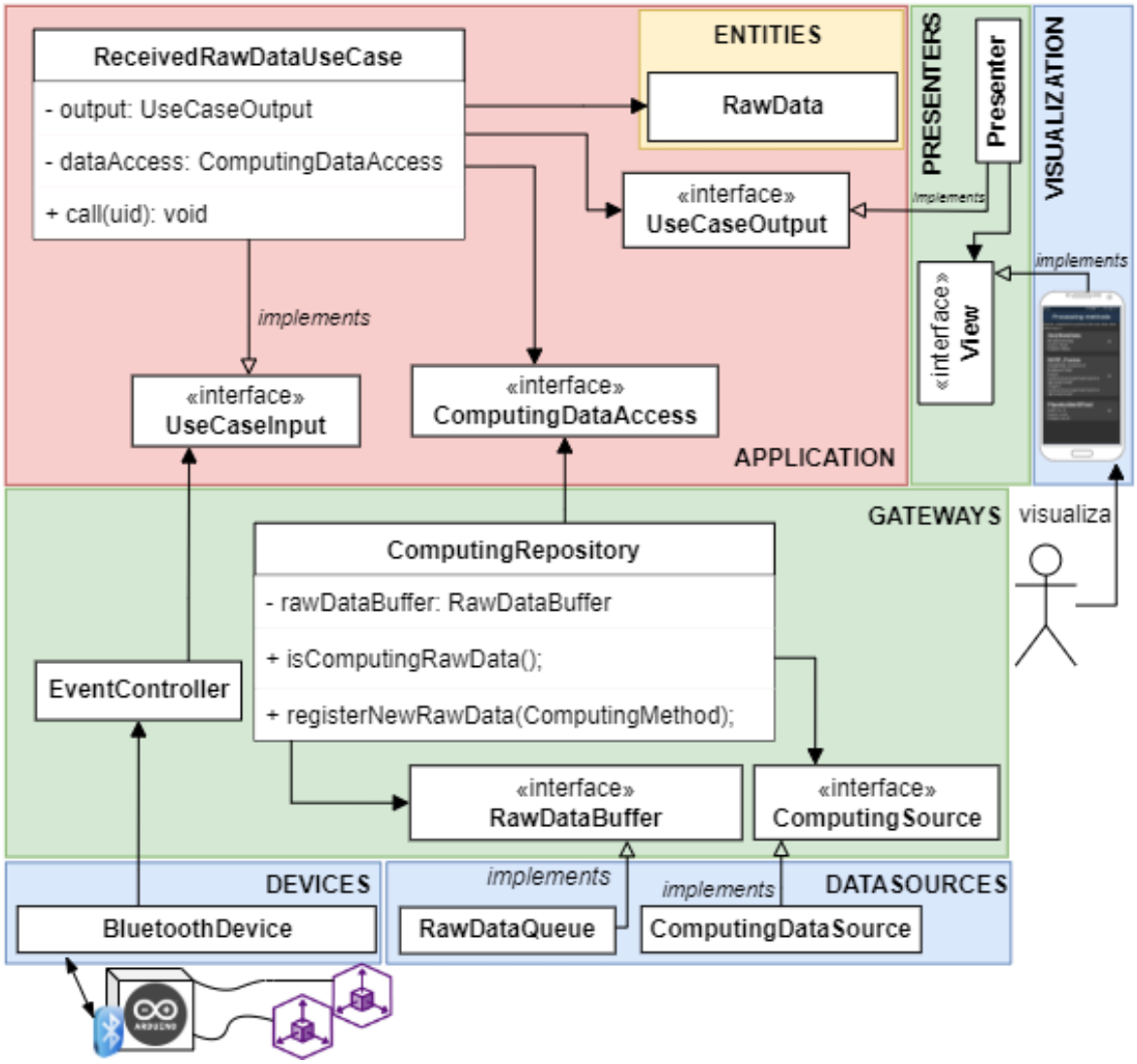


Figura 32 – Fluxo de controle completo da ação gerada pelo recebimento de um dado inercial do dispositivo.

Ao receber os dados inerciais um caso de uso deverá enviar os dados para o apresentador que por conseguinte irá redirecionar os dados para visualização no formato correto, exibindo os dados inerciais brutos para o usuário. Além disso o caso de uso deve perguntar para as camadas externas se foi iniciado o processamento de dados. Em caso afirmativo a implementação irá também repassar os dados inerciais para o repositório de processamento, que por conseguinte irá adicionar no *buffer* de dados inerciais recebidos que é implementado na camada externa e será usado pelo motor de processamento.

Observe que o caso de uso que recebeu os dados inerciais não é responsável por processar os dados, mas possui uma responsabilidade extra de saber se a aplicação está processando dados, isto é necessário pois um motor de processamento usa os dados inerciais presentes em um *buffer* para determinar quando processar os dados com o método de processamento selecionado. Logo, este caso de uso envia uma cópia do dado inercial que acabou de ser recebido para o repositório de processamento existente na camada

de adaptadores. O motor de processamento existe na camada de adaptadores e verifica constantemente o *buffer* de dados inerciais e, enquanto houver dados a serem consumidos e o aplicativo estiver processando, o motor de processamento processa os dados e envia o resultado para o controlador de eventos, que dispara o caso de uso que é responsável por receber os dados processados.

Como ilustrado na Figura 33, o motor de processamento (*ComputingEngine*) consome os dados de uma fila existente na camada externa que é a implementação do *RawDataBuffer* existente na camada de adaptadores e alimenta os dados para o método de processamento. A figura também ilustra como os métodos de processamento customizados são implementados por meio do padrão *Strategy*, explicado na Subseção 2.1.2. Mais detalhes sobre o motor de processamento e os métodos são encontrados na Seção 4.2.

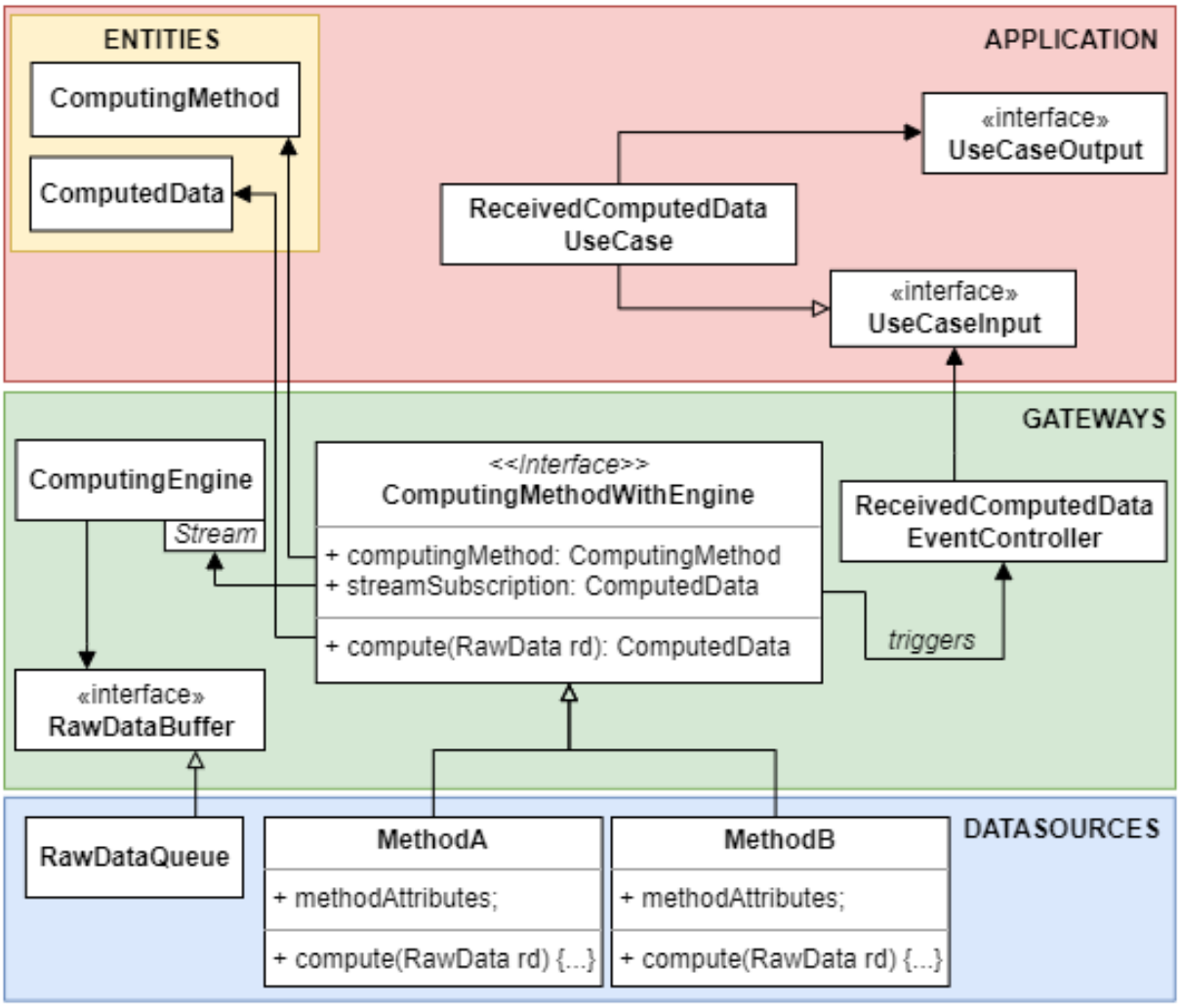


Figura 33 – Motor de processamento consumindo dados inerciais, enviando para métodos de processamento customizados usando o padrão *Strategy* e enviando os dados processados para o controlador responsável.

Assim que o controlador de evento *ReceivedComputedDataEventController* é acionado e por conseguinte o caso de uso responsável pelos dados processados, os dados são

direcionados para o apresentador responsável que por fim chega na visão dos dados processados, terminando o fluxo de controle do caso de uso na tela D da Figura 28. Observe que durante toda essa comunicação a regra de dependência não foi quebrada entre as camadas, pois a inversão de dependência é sempre aplicada na comunicação de dentro para fora. Como as responsabilidades são bem definidas dentro de cada componente e dentro de cada camada o desenvolvimento dos componentes são rápidos e focados.

3.3.3 *Inner* - Camadas internas

As camadas internas compõem o núcleo funcional do aplicativo. Elas por si só devem conseguir demonstrar o motivo do aplicativo existir sem a necessidade de *frameworks* e interfaces de usuário. As duas camadas existentes não devem possuir nenhuma dependência que apresente risco futuro ao sistema, isso quer dizer que não se deve adotar riscos que não se pode controlar como a dependência com *frameworks* externos com intuito de reduzir ou facilitar o desenvolvimento de componentes nessa camada. Neste projeto ambas camadas são escritas na linguagem de programação escolhida sem a adoção de *frameworks* ou bibliotecas externas. A única dependência é a linguagem e as bibliotecas fornecidas pela própria linguagem por padrão.

3.3.3.1 *Entities* - Regras de negócio

A camada *Entities* (entidades) desta arquitetura existe para representar as regras de negócio. Ela pode ser vista como apenas uma modelagem estrutural do negócio, definindo as entidades, relações entre as entidades e regras e funções de cada entidade ou relação. As entidades encapsulam o nível mais alto e genérico de regras, pois não existe sistema ou aplicação dentro desta camada. Por exemplo uma alteração de interface, de pesquisa ou de segurança não devem alterar entidades nesta camada pois as responsabilidades de operações do sistema estão alocadas na camada vizinha.

As entidades são usadas pelos casos de uso para executar as funcionalidades que a aplicação exige. Note que as entidades podem existir sem os casos de uso, e os casos de uso existem para atender as necessidades do projeto fazendo uso das entidades. A relação das entidades com os dados armazenados são situacionais e, podem se refletir totalmente, parcialmente ou não se refletir. A modelagem das entidades nesta camada, também conhecida como o domínio do problema, não precisa necessariamente ter relacionamento direto com os dados, pois o armazenamento só existem na camada externa. Além disso, os dados devem passar pelos repositórios (camada de adaptadores) para serem convertidos em entidades, tornando possível o seu uso nas implementações dos casos de uso.

O domínio do problema para este projeto é bem simples e, a modelagem desta camada como ilustrada na Figura 34, possui poucas entidades, como a definição dos dados inerciais brutos e processados, dos métodos computacionais e do dispositivo de trans-

missão. Observe que neste modelo arquitetônico pode-se tomar decisões de tecnologias mais abrangentes. A decisão de modelar o estado da transmissão como uma transmissão abstrata, ao invés de se fixar na tecnologia Bluetooth, deixa o sistema mais aberto para outros tipos de dispositivos, como: *Wi-Fi*; ondas de rádio ou; transmissão cabeada.

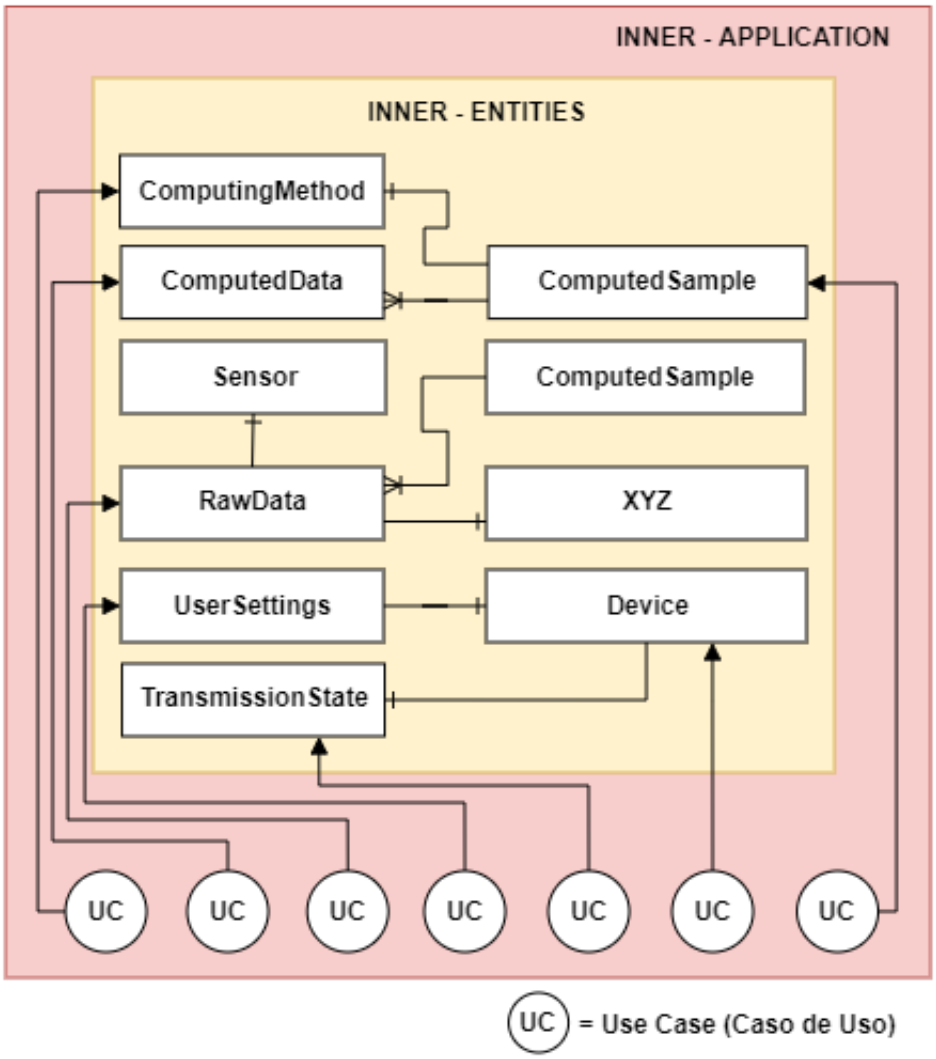


Figura 34 – Modelagem da camada *Entities*.

3.3.3.2 *Application* - Regras da aplicação com casos de uso

Quando as regras de negócio entram no contexto técnico de uma aplicação, a camada *Entities* não se enquadra mais nesta responsabilidade. Para isto a camada interna que define estas regras de negócio e interações é a camada *Application* (aplicação). Os componentes principais desta camada são os casos de uso e podem ser definidos como uma descrição de um processo automatizado, que pode receber informações (parâmetros) fornecidos pelo usuário ou agentes externos e produzir um resultado que é enviado para um apresentador. Os casos de uso representam as regras de negócio da aplicação, são responsáveis por fazer o uso correto das entidades de acordo com sua responsabilidade, que pode também incluir a interação com dados nas camadas externas.

A Figura 35 ilustra os casos de usos da funcionalidade apresentada na Subseção 3.3.1. Observe que existe a dependência direta com as entidades da camada *Entities* e também dependência com diversos componentes internos que aplicam a inversão de dependência com as outras camadas. O componente abstrato que define o acesso a dados é chamado de *DataAccess*. Este define os métodos e o retorno que as próximas camadas devem produzir ao serem chamadas. A implementação do *DataAccess* é um repositório (*Repository*) existente na camada de adaptadores. A definição dos dados que entram e saem desta camada também é especificada aqui. Os componentes das outras camadas tem que obedecer a estes formatos de dados para poder se comunicar com a camada de aplicação por meio da implementação de seus componentes abstratos.

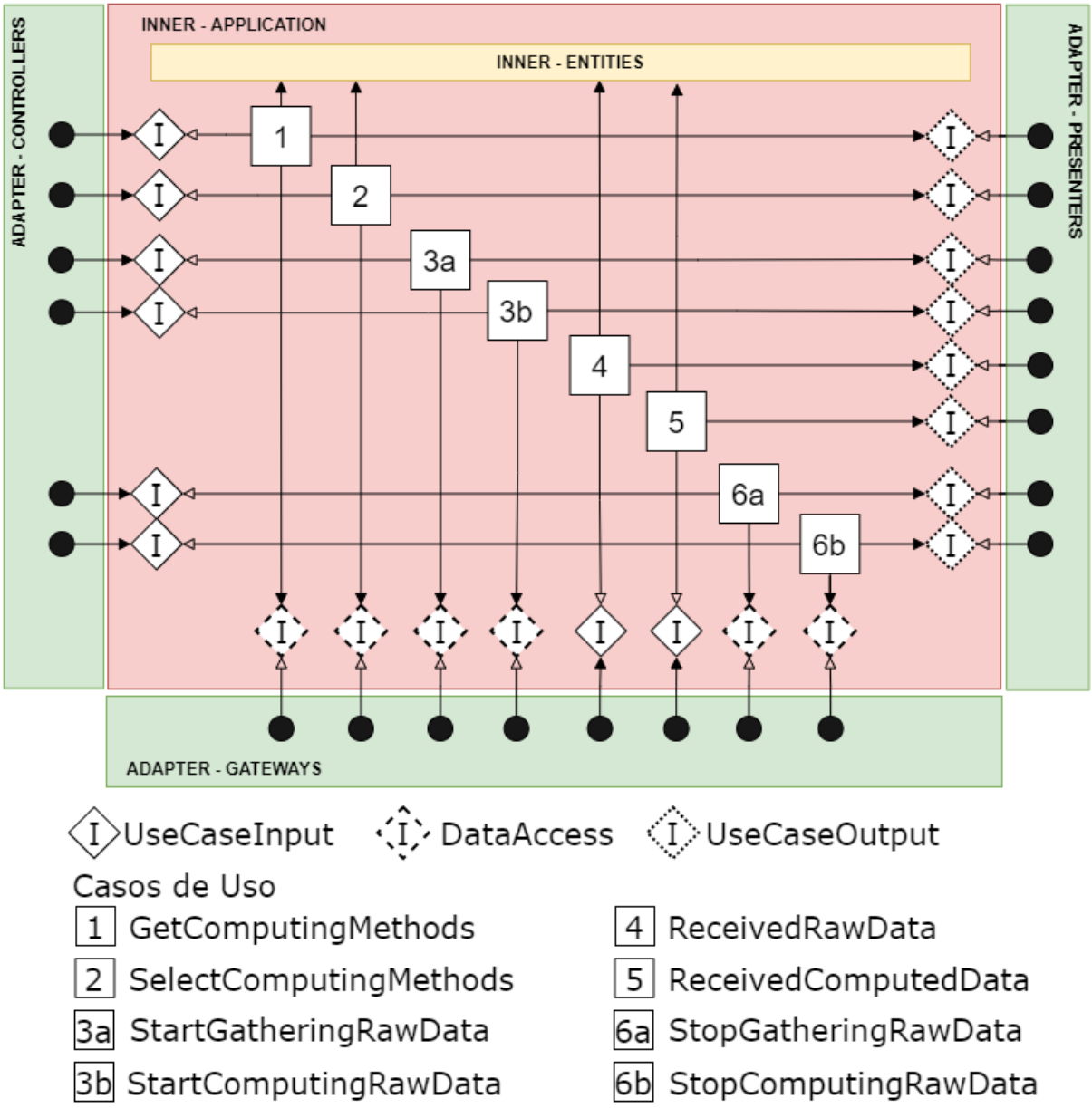


Figura 35 – Modelagem dos casos de uso e abstrações apresentados existentes na camada *Application*.

O formato de dados que entra para um caso de uso e sai do caso de uso para um apresentador é definido pela camada de aplicação e pode ter ou não relação direta com as entidades, porém é importante notar que é de responsabilidade do caso de uso definir estes padrões. O caso de uso possui em sua descrição os parâmetros que deve receber para ser executado (dados de entrada - *InputDTO*) pelo controlador, e também cria a definição de como os dados devem ser enviados para o apresentador (dados de saída - *OutputDTO*). A codificação de um caso de uso novo, na maioria das vezes, não é apenas a implementação de uma classe, mas sim todo um conjunto de classes que definem a comunicação com as outras camadas por meio da inversão de dependência.

3.3.4 *Adapter* - Camadas intermediárias

Camadas intermediárias são a ponte entre as regras de negócio e os agentes externos. Os componentes destas camadas possibilitam a comunicação entre um botão na interface de usuário com um banco de dados na nuvem por exemplo. É função dos adaptadores manter a consistência dos dados que entram e que saem das camadas internas. Isto posto, existe uma responsabilidade de saber e definir o padrão para os agentes externos (interface de usuário, banco de dados) seguirem (COCKBURN, 2005; MARTIN, 2017).

A camada de adaptadores neste projeto é composta pela camada de entradas externas *Gateways*, a camada de apresentação *Presenters* e a camada de controladores *Controllers*, apesar dessas camadas serem vizinhas seus componentes internos não devem criar dependência laterais por serem parte da camada intermediária. Por este motivo optou-se por uma representação mais horizontal da arquitetura como ilustrado na Figura 26, diferentemente da arquitetura original na Figura 6 que não possui separação dos componentes nesta camada, estando mais próxima da Figura 8.

3.3.4.1 *Controllers* - Controle na entrada de dados e execuções internas

O principal ponto de entrada para agentes externos é o conjunto de componentes chamados de controladores (*Controllers*) que existem nesta camada. Os controladores são adaptadores dos dados de entrada e são responsáveis pela conversão destes dados para o padrão mais conveniente para o caso de uso. O controlador também tem o poder de decidir como os parâmetros de entrada necessários para uma certa execução são expostos aos agentes externos. Outro ponto importante é que o controlador não possui dependência direta com o caso de uso, o que cria uma proteção na responsabilidade e coesão deste componente, pois este não possui conhecimento dos detalhes dos casos de uso. Por conseguinte este desacoplamento também gera a possibilidade da troca de dependência em tempo de execução. Por exemplo seria possível a execução de casos de usos diferentes de acordo com o perfil de acesso que o usuário está acessando (MARTIN, 2017).

A Figura 36 ilustra a modelagem dos controladores que permitem a execução do caso de uso apresentado na Subseção 3.3.1. Os componentes são de implementação simples e possuem poucas responsabilidades favorecendo a alta coesão do controlador. A responsabilidade do controlador na arquitetura limpa é menor e mais coesa do que os controladores de outros padrões arquitetônicos como MVC (*Model-View-Controller*) evitando más práticas como classes promíscuas ou classes deuses (*God Classes*). Como a arquitetura limpa naturalmente fornece uma diversa gama de componentes e favorece a separação de responsabilidades entre eles, a coesão dos componentes é alta pois as responsabilidades são poucas e bem definidas (LARMAN, 2004; ANICHE et al., 2017; MARTIN, 2017).

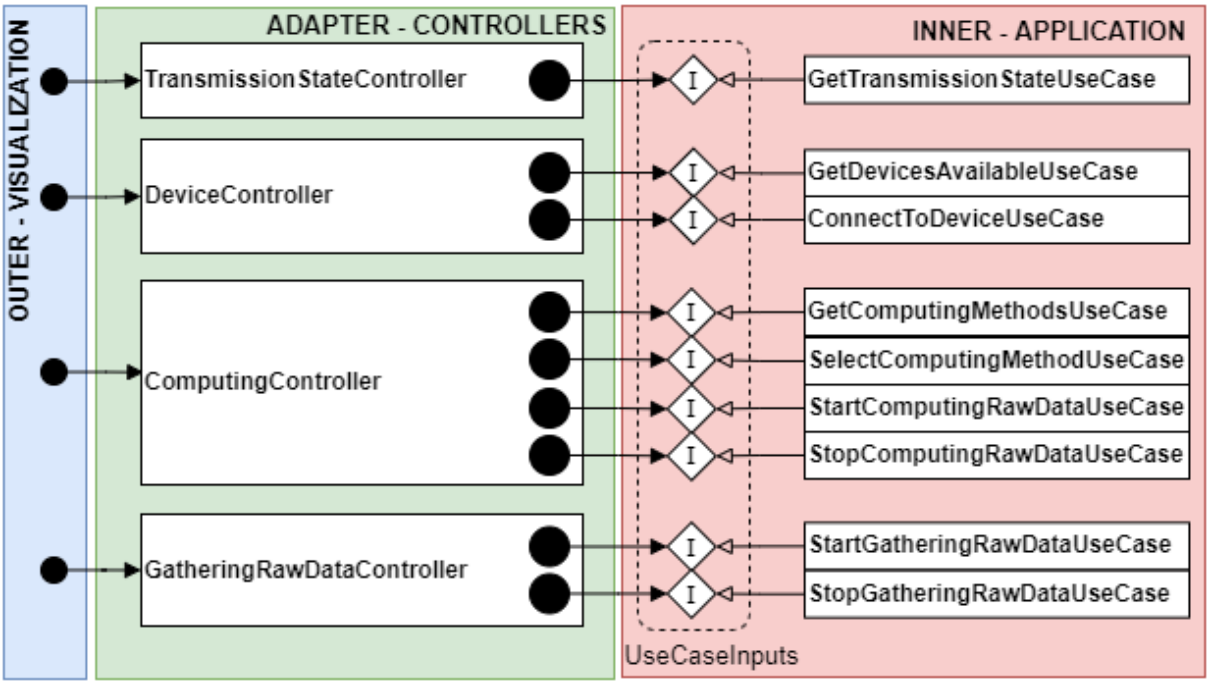


Figura 36 – Camada de controladores e dependência com casos de uso apresentados.

3.3.4.2 *Gateways* - Acesso a fontes de dados e comunicação com o exterior

Quando um caso de uso necessita de dados por algum motivo é criada uma dependência dentro da camada de aplicação com um componente abstrato chamado *DataAccess*. Esse componente é implementado por um repositório (*Repository*) nesta camada. Os repositórios são os componentes que viabilizam a comunicação com dados externos, porém ainda não há dependência com qualquer *framework* ou com a implementação de tecnologias de banco de dados. Apesar de mais próximo dos dados, os repositórios ainda possuem certo nível de abstração em suas responsabilidades. Isto permite que o repositório possa tomar decisões de qual fonte de dados buscar em caso de uma indisponibilidade temporária por exemplo (MARTIN, 2017).

Martin (2017) exemplifica que os componentes desta camada poderiam ter uma dependência de tecnologia pois é mencionado que se a escolha de banco de dados for SQL, todos os SQLs deveriam estar nesta camada, e a escolha de um SGBD (Sistema Gerenciador de Banco de Dados) como Postgres ou SQLite ficaria na camada externa. Neste projeto optou-se por não seguir esta metodologia para não criar dependências de tecnologia nesta camada, pois caso a escolha de tecnologia mude de SQL para NoSQL ou para uma API externa, não haveria impacto nos repositórios e apenas nas implementações dos componentes nas camadas externas.

A Figura 37 ilustra a implementação de um repositório (*ComputingRepository*) e como é feita a inversão de dependência com a camada *Application*. Note que as implementações dos *DataAccess* são forçadas a seguir os padrões de sua abstração, isto garante a implementação dos métodos abstratos e suas funcionalidades. O relacionamento entre dados e entidades pode ser adaptado aqui nesta camada por meio de *DataMappers* que fazem a conversão dos dados externos para entidades, porém como a estrutura destas entidades é similar aos dados que são armazenados, optou-se por forçar o padrão estabelecido já nas entidades evitando a criação de *DataMappers* nesta camada.

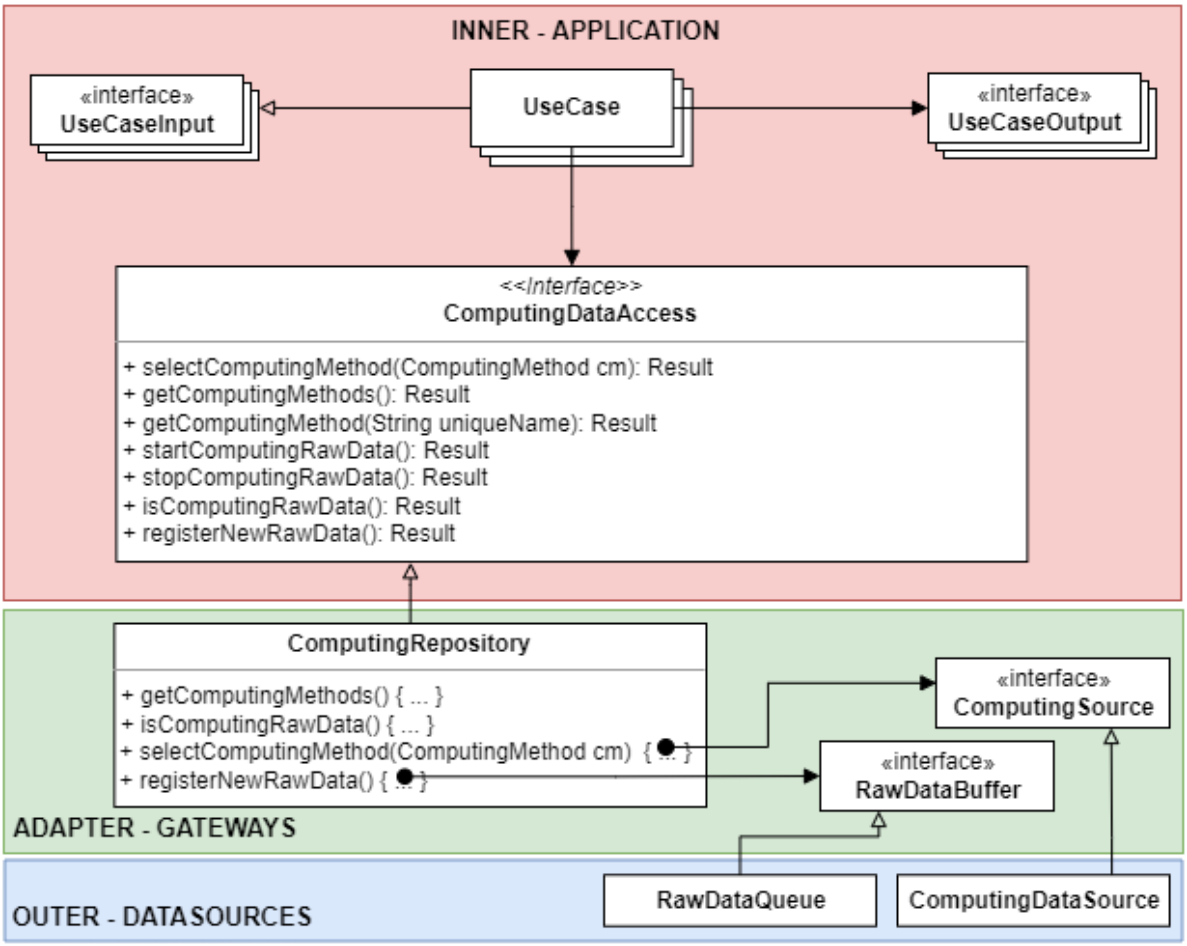


Figura 37 – Modelagem do repositório ComputingRepository e sua relação com as camadas vizinhas.

Para que os repositórios não criem dependência com as implementações específicas de dados cria-se um modelo abstrato para seguir o padrão da inversão de dependência novamente. As abstrações chamadas de *Sources* (fontes) são criadas para serem implementadas na camada externa por componentes que devem realizar a comunicação com uma tecnologia em específico, podendo ser um banco de dados, uma API de terceiro, ou até mesmo uma biblioteca de um dispositivo móvel que permite acessar o estado da transmissão de dados. Como este projeto usa o *Bluetooth* do *smartphone* para se conectar ao dispositivo e transmitir os dados, existe a necessidade de uma biblioteca que permita a comunicação com o sistema operacional do *smartphone* para utilizar a tecnologia. A dependência com essa biblioteca é feita em uma classe na camada externa que implementa a abstração *Source* desta camada.

3.3.4.3 *Persenters* - Controle para apresentação de dados

A execução de um caso de uso gera o resultado para um apresentador em vez de retornar para o ponto de chamada. Este processo não é natural pois ao chamar um método espera-se um resultado de retorno, já neste caso, quando o controlador executa um caso de uso o retorno que o controlador recebe não é importante, pois este não possui responsabilidade de realizar ações de acordo com o retorno da chamada. Em vez disso, o caso de uso gera o resultado da chamada e envia para o apresentador, seja um resultado de sucesso ou falha. A decisão de enviar o fluxo de controle para o apresentador em vez de um retorno para o solicitante é arquitetural. O desacoplamento sugerido por Martin (2017) menciona que o controlador não deve ter conhecimento dos apresentadores para não gerar uma dependência lateral na camada.

Os apresentadores (*Presenters*) são responsáveis pela visualização e formatação dos dados produzidos pelos casos de uso, isto quer dizer por exemplo, que não é responsabilidade do caso de uso saber se o idioma do aplicativo está em Português ou Inglês. Quando houver necessidade de internacionalização são componentes desta camada que são os responsáveis. Toda responsabilidade que envolve a preparação e apresentação de dados para uma visão é realizada nesta camada. Para isto, além dos *Presenters* existem os componentes de visão (*View*) e o modelo da visão (*ViewModel*) (MARTIN, 2017).

A Figura 38 ilustra os componentes envolvidos ao receber o resultado de um caso de uso. Note que a *View* é o componente abstrato que realiza a inversão de dependência com a camada externa e os dados utilizados pela *View* são estruturados no *ViewModel*. Este desacoplamento de dados permite ao *Presenter* receber os dados no modelo que foi definido na camada de aplicação e prepará-los para o formato adequado do *ViewModel* para ser utilizado por um componente externo que implementa a *View*. Neste caso em específico os dados processados são estruturados para um modelo mais conveniente para serem exibidos por alguma visão, e na camada externa temos os componentes em *Flutter*

que realizam a exibição destes dados para o usuário na tela D da Figura 28.

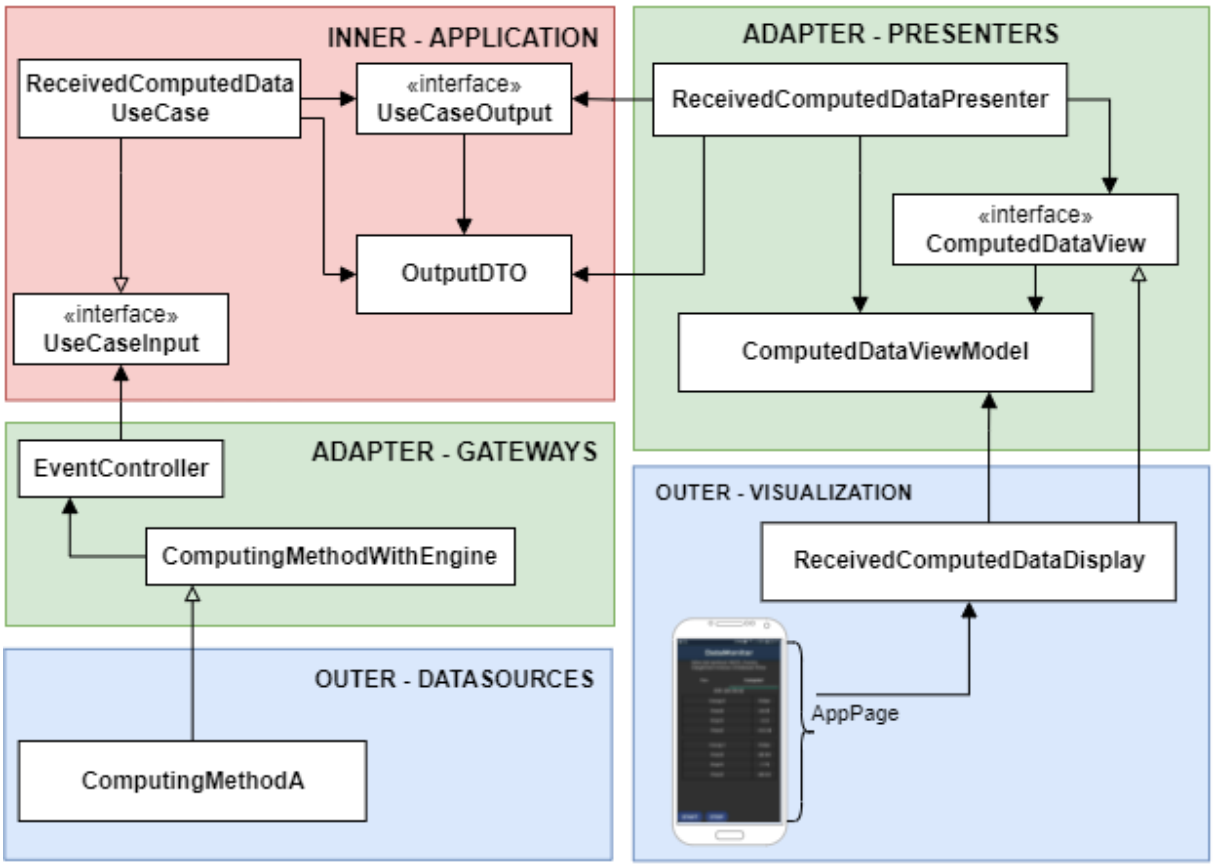


Figura 38 – Apresentador *ReceivedComputedData* e relação com camadas vizinhas.

3.3.5 *Outer* - Camadas externas

As camadas internas e intermediárias possuem dependência apenas com a linguagem escolhida Dart e as bibliotecas nativas da linguagem. Diferentemente, nas camadas externas é possível assumir riscos maiores com tecnologias e bibliotecas que fogem do controle dos donos do projeto por exemplo. Os detalhes do sistema estão presentes apenas nas camadas externas, isto quer dizer que os componentes são menos abstratos que os componentes localizados internamente na arquitetura. Por exemplo, componentes como *frameworks* de interface de usuário, bibliotecas de banco de dados ou de mapeamento objeto-relacional, configurações específicas necessárias para comunicação com APIs, dispositivos ou sensores são implementados nesta camada (MARTIN, 2017).

A Figura 39 ilustra que a camada externa desta arquitetura possui três cortes para separar melhor as responsabilidades e as fronteiras com as camadas intermediárias. A camada de visualização (*Visualization*) é aonde está presente a implementação do aplicativo em Flutter e também é o ponto de entrada do sistema. A camada de dados (*DataSources*) é para implementações de persistência de dados e comunicação com APIs externas se necessário. Por fim, a camada de dispositivos (*Devices*) implementa a comunicação com

quaisquer tipo de dispositivos, seja ele externo como o dispositivo inercial ou interno, como o próprio *smartphone*.

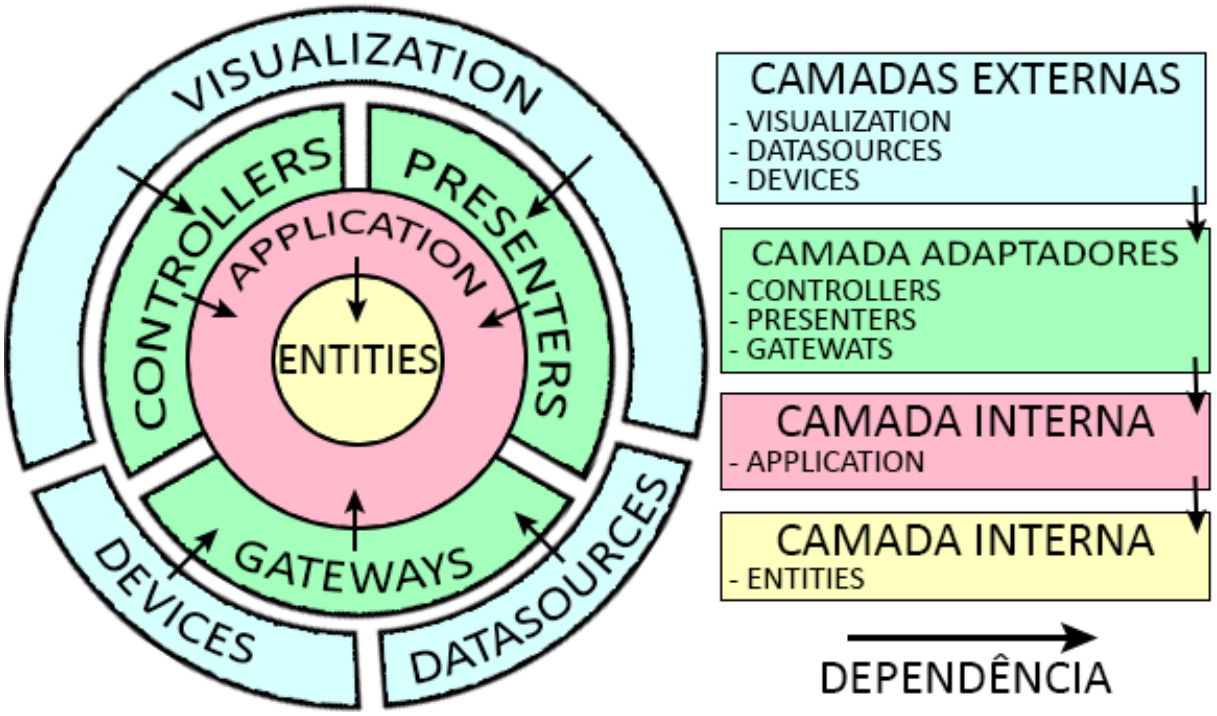


Figura 39 – Visualização de alto nível de todas camadas da arquitetura.

3.3.5.1 *Visualization* - Implementação visual e interfaces de usuário

A camada externa de visualização (*Visualization*) concentra as responsabilidades de visualização final para o usuário. As interfaces de usuário são definidas em componentes nesta camada e são dependentes da tecnologia escolhida. Por exemplo uma interface web poderia ser feita em Angular ou ReactJS nesta camada. Este projeto implementa um aplicativo móvel multiplataforma nesta camada com o *framework* Flutter. A dependência com a tecnologia de visualização escolhida só existe nesta camada, pois o desacoplamento da arquitetura permite a alteração desta tecnologia por outras como AngularDart ou VueDart. A refatoração que seria necessária no código não se estenderia para outras camadas, pois elas não possuem dependências com este *framework* (MARTIN, 2017).

As telas do aplicativo deste projeto possuem objetos "controladores" para realizar as ações por meio dos comandos do usuário. Esses objetos possuem dependência com os controladores da camada de adaptadores (Subseção 3.3.4.1) e com o objeto que implementa a visão (Subseção 3.3.4.3) correspondente para receberem os resultados das chamadas. A Figura 40 ilustra os componentes envolvidos no clique de um botão por parte do usuário. Note que o controlador de página é o responsável por gerar a ação no controlador da camada adaptadores e obter a resposta por meio de uma visão.

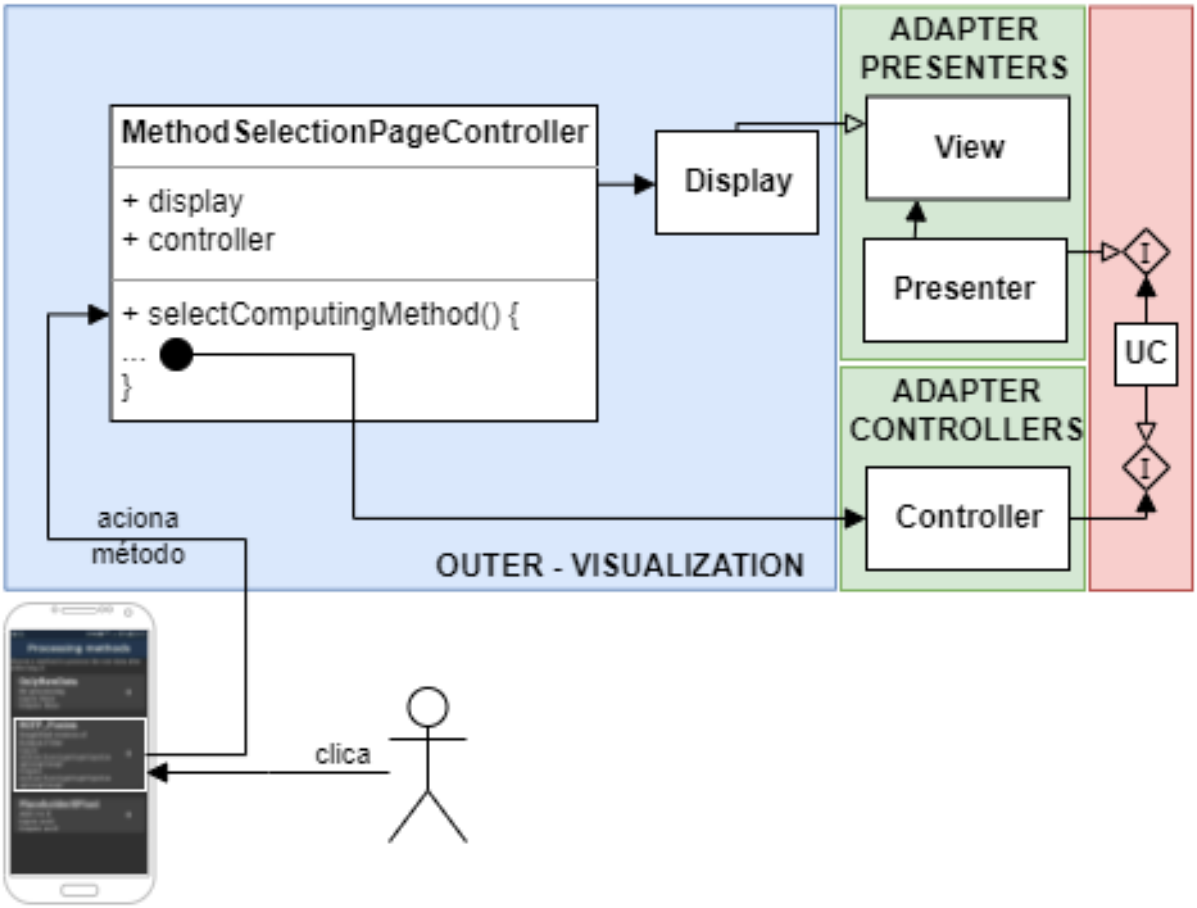


Figura 40 – Exemplo do controlador de tela chamando adaptador e recebendo resposta pela visão.

Este projeto usa um componente nomeado de controlador de página (*PageController*) para realizar as ações de cada tela. Este objeto é responsável por capturar as ações do usuário e responder de acordo, como também acionar as atualizações na tela de acordo com a resposta obtida. O aplicativo do projeto possui várias telas, que não necessariamente precisam refletir as classes de visão da camada de adaptadores. Observe que uma visão é a representação visual da resposta de um caso de uso acionado. A implementação visual do aplicativo pode diferir de acordo com as tecnologias adotadas (por exemplo, criar uma tela para aplicativo é diferente de uma tela para desktop), assim o padrão de projeto e a organização de pacotes e classes aplicados nesta camada podem diferir entre as aplicações existentes atualmente.

3.3.5.2 *Data Sources* - Implementação das fontes de dados

Quando os casos de uso precisam de acesso aos dados é feita uma chamada aos repositórios, que são responsáveis pela definição abstrata que as fontes (*Source*) devem seguir. A implementação destas fontes de dados (*DataSource*) é feita nesta camada e pode criar dependências com tecnologias de terceiros como bancos de dados (como PostgreSQL

ou SQLite), API's, armazenamento interno do dispositivo, entre outros. É importante ressaltar o benefício que o desacoplamento destas tecnologias trás para um projeto, pois torna possível a troca de fonte de dados sem a necessidade de atualizar o funcional do projeto, apenas trocando a referência da implementação fonte de dados utilizada pelo repositório por meio da inversão de dependência.

A Figura 41 exibe as tecnologias utilizadas por este projeto nesta camada. O aplicativo utiliza uma implementação simplória de fila (*Queue*) para realizar o armazenamento dos dados brutos enviados pelo dispositivo. Esta fila então é consumida pelo motor de processamento para realizar o processamento dos dados inerciais no método escolhido. Além disso, algumas outras informações de contexto do usuário são armazenadas em um banco de dados NoSQL internamente no dispositivo pelo *framework* HiveDB. Esta implementação permite a futura persistência de amostras dos dados coletados e processados para exportação em outros formatos de arquivos por exemplo.

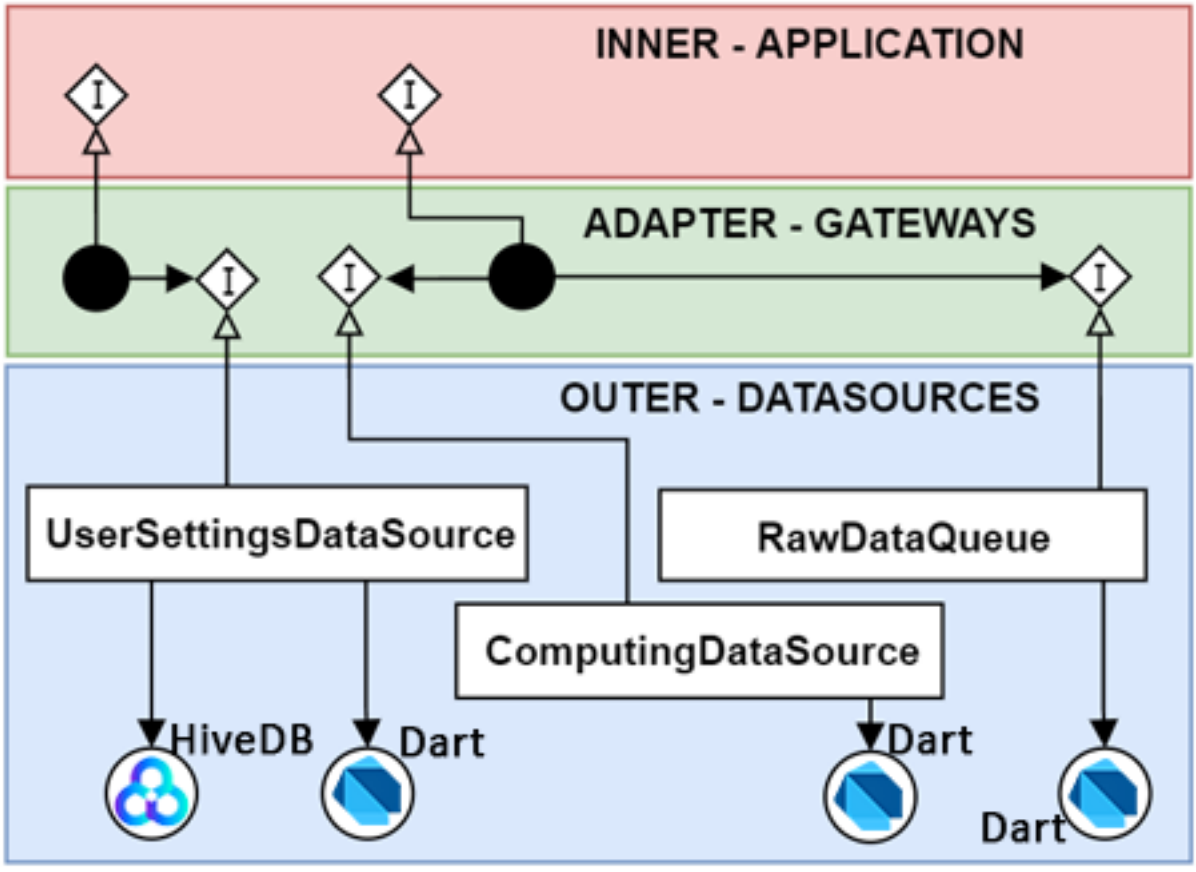


Figura 41 – Tecnologias utilizadas na camada de dados.

3.3.5.3 *Devices* - Implementação da comunicação com dispositivos

A camada de dispositivos (*Devices*) possui a implementação da comunicação com os dispositivos externos e internos, como um terminal de atendimento, uma máquina de cartão de crédito ou, no caso deste projeto, o dispositivo que irá enviar dados inerciais

e o *smartphone* que possui a tecnologia Bluetooth para realizar a comunicação com o dispositivo. Ambos são considerados dispositivos e não devem ser considerados como fonte de dados pois já existe outra camada para esta responsabilidade. Esta camada poderia ser implementada juntamente com a camada externa de fontes de dados (*DataSource*), porém para manter a coesão alta dos componentes foi decidido separá-las.

Observe na Figura 42 que mesmo na camada de dados é adicionada a dependência com Flutter, pois neste caso utiliza-se a biblioteca flutter_blue para possibilitar o uso do Bluetooth no *smartphone*, o que permite realizar a conexão e receber os dados enviados pelo dispositivo inercial. A dependência com Flutter na camada de dispositivos não é obrigatória, visto que é possível utilizar outras tecnologias e *frameworks* para realizar esta comunicação Bluetooth. Porém a dependência maior é com a linguagem Dart que foi escolhida e utilizada nas camadas internas, tendo em vista que a comunicação entre camadas é toda feita por conceitos de orientação a objetos desta linguagem.

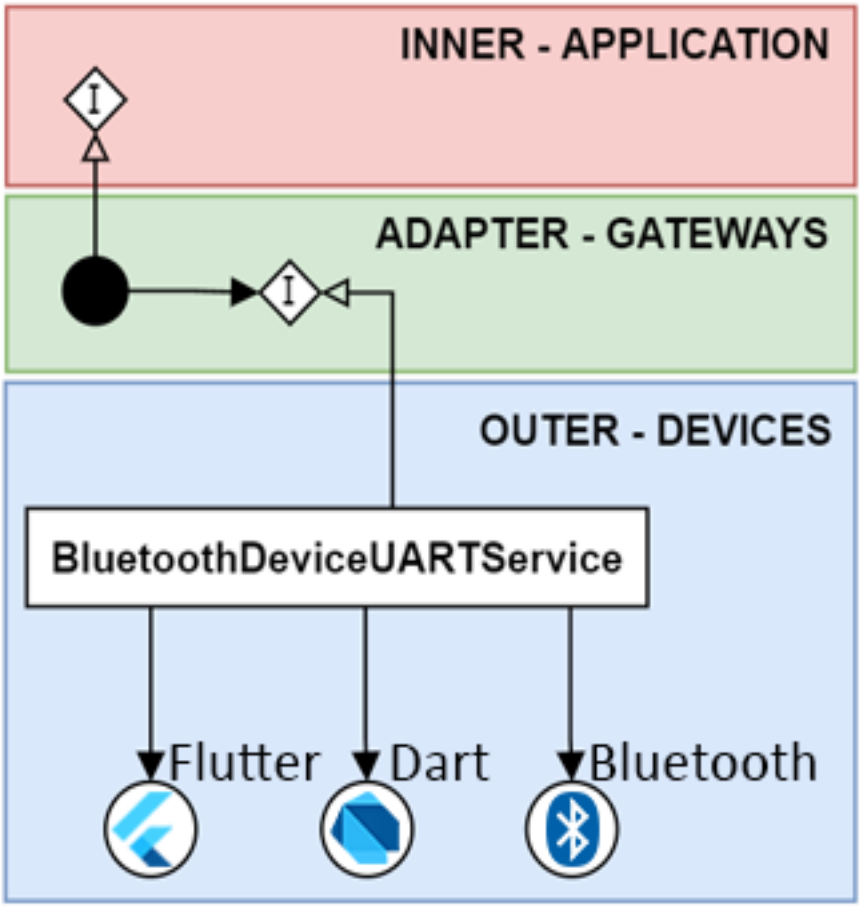


Figura 42 – Tecnologias utilizadas na camada de dispositivos.

A implementação da comunicação é particular ao dispositivo utilizado pelo projeto. Neste projeto utiliza-se um dispositivo inercial de baixo custo, que pode ser adquirido e montado facilmente por ser construído com um processador compatível com a plataforma Arduíno (Seção 4.3). Isto posto, nada impede que sejam feitas outras implementações com

diferentes dispositivos, sejam eles de tecnologia proprietária ou não. Qualquer dispositivo que possibilite a conexão e a transmissão de dados inerciais, pode ser integrado a este projeto desde que forneça a documentação necessária para o mesmo.

3.3.6 *Entrypoint* - Inicialização do sistema e injeção de dependências

O ponto de entrada do sistema é o local que acontece a inicialização do sistema e todos os processos envolvidos nesta etapa. O aplicativo móvel em Flutter é o ponto de entrada deste projeto por ser o componente externo que irá captar as entradas do usuário e receber os dados inerciais por Bluetooth. Observe que até agora a comunicação entre camadas é majoritariamente realizada por meio da inversão de dependência, porém o componente que utiliza este padrão para se comunicar com outra camada faz uso de um objeto abstrato, que não foi inicializado e precisa de uma instância concreta para que o sistema completo seja funcional.

O componente que realiza a injeção de dependências fica na camada externa desse projeto junto com o aplicativo Flutter. Note que é possível também realizar o desacoplamento deste injetor em outra camada externa lateral, isto seria interessante caso exista diferentes interfaces de usuário, como uma interface web ou desktop por exemplo. A biblioteca utilizada para realizar a injeção de dependências chama-se get_it e ela permite a criação de instâncias concretas no padrão *Singleton* para classes abstratas e concretas e, é considerada pelos autores como um localizador de serviços (*Service Locator*), um conceito que também se enquadra na funcionalidade de injeção de dependências.

Toda a arquitetura do sistema possui componentes que fazem uso de abstrações para funcionar corretamente e, todos esses componentes são acoplados com as instâncias concretas na inicialização do sistema. Um exemplo são os controladores que fazem uso das abstrações dos casos de uso, como o caso de uso que é responsável por listar os métodos de processamento customizados. Isso possibilita a troca deste caso de uso por outra instância concreta caso a regra de negócio mude, mantendo a instância anterior no sistema caso seja necessário.

Como explicado anteriormente, neste projeto os repositórios fazem uso de abstrações nomeadas *Source* para aplicar a inversão de dependência. A implementação de um *Source* é um *DataSource* na camada externa de dados. A Figura 43 exemplifica como o injetor de dependências realiza a ligação entre os repositórios e as fontes de dados. O objeto "sl" (*ServiceLocator*) é responsável por criar e recuperar as instâncias correta (como o padrão *Abstract Factory*). Note que a instância concreta do *DevicesDataAccess* é o repositório *DevicesRepository* e, por conseguinte, a instância concreta do *DevicesSource* é o serviço *BluetoothService*.

A inicialização do repositório é dada pela chamada de seu construtor "Devices-Repository(sl())", note que é necessário um parâmetro, que é do tipo *DevicesSource*. A chamada ao injetor de dependências ("sl()") para este parâmetro, irá devolver a instância concreta que foi inicializada por aquela abstração, que é o *BluetoothService*. O serviço *BluetoothService* realiza a integração com o Bluetooth do *smartphone* e é responsável pela conexão com o dispositivo inercial, possuindo dependência com biblioteca flutter_blue. Essa instância pode ser alterada futuramente por implementações com uma biblioteca *Wi-Fi* ou USB por exemplo, sem que exista impacto nas camadas internas, pois note que a mudança é apenas a instância no injetor de dependência e um componente novo na camada externa.

```dart
final sl = GetIt.instance;

/// Concrete Flow of Control:
/// Controllers --> UseCases --> Repositories --> DataSources
///                   ┌<-    <--       .       <--     ┘
///                UseCases -->  Presenters  --> Displays
///
/// Abstract Flow of Control
/// Controllers --> InputBoundary --> DataAccess --> Sources
///                       ┌<-      <--      .      <--   ┘
///                InputBoundary --> OutputBoundary --> View
///
Future<void> init() async {
  /// Instantiate Controllers
  // ...
  /// Instantiate UseCases
  // ...
  /// Repositories
  sl.registerLazySingleton<DevicesDataAccess>(() => DevicesRepository(sl()));
  // ...
                                  (new) DevicesRepository(DevicesSource _devicesDataSource)
                                  package:intracs_gateways/src/devices/devices_repository.dart
  /// DataSources
  sl.registerLazySingleton<BluetoothService>(() => BluetoothService(sl()));
  sl.registerLazySingleton<DevicesSource>(() => sl<BluetoothService>());
  sl.registerLazySingleton<TransmissionStateSource>(() => sl<BluetoothService>());
  // ...

  /// Instantiate Presenters
  // ...
  /// Instantiate Displays
  // ...
}
```

Figura 43 – Trecho de código da injeção de dependências na inicialização do sistema.

3.4 Hardware do Protótipo Inercial

O protótipo apresentado na Figura 44 é composto por três placas diferentes que são empilháveis entre si por meio de um conector SPI (*Serial Peripheral Interface*) de 32 pinos, localizado em cada placa que podem ser substituídas por outras futuramente. A primeira placa é um processador Atmel SAMD21 32-bit ARM Cortex M0+, o mesmo usado nas placas do Arduino Zero. Encaixada acima do processador está uma placa de *Bluetooth Low Energy* 4.1, que possui o chipset BlueNRG-MS da STMicroelectronics. E por cima, está um adaptador com um multiplexador de quatro canais, tornando possível a conexão de quatro sensores ao mesmo tempo, que são chamados de *Wirelings* pelo fabricante.

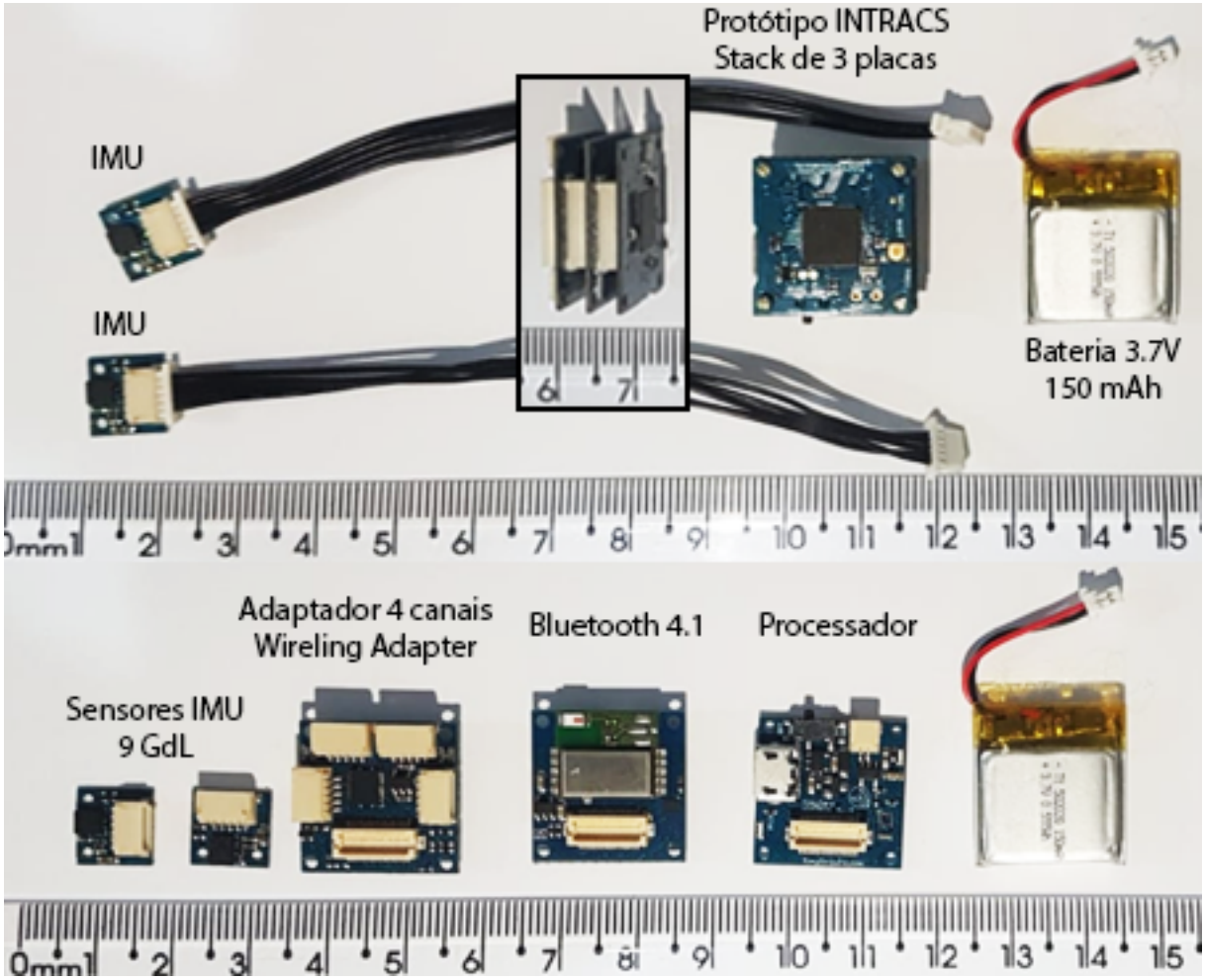


Figura 44 – Tamanho dos seis componentes que integram o protótipo.

Este projeto usa dois sensores iguais, que são unidades de medida inercial (IMU) de nove eixos presentes no chipset STLSM9DS1, que contém: um acelerômetro de três eixos; um giroscópio de três eixos e; um magnetômetro de três eixos (totalizando os nove eixos do componente). Apesar deste projeto usar apenas dois sensores é possível empilhar oito placas de adaptadores em um mesmo sistema, tornando possível a presença de 32 sensores (oito placas com quatro sensores cada). No entanto, a adição de cada IMU de

nove eixos implica em uma partilha extra na taxa de transmissão, pois os dados de cada sensor são enviados sequencialmente como explicado na Subseção 4.3.2.

Os componentes empilháveis apresentados na Figura 44 e escolhidos por este projeto são da empresa TinyCircuits[1]. Além de possuírem o tamanho reduzido fornecendo uma portabilidade alta para o protótipo, eles permitem a atualização de tecnologia entre eles. É possível por exemplo, trocar a placa de Bluetooth LE 4.1 por uma placa de *Wi-Fi* ou de Bluetooth 5.0 (ainda não disponível) e, com poucas alterações no sistema o protótipo continuaria funcionando. Isto torna o custo de atualização da tecnologia extremamente baixo, o que não acontece com as outras opções, pois seria necessário adquirir um novo produto por completo.

[1] <https://tinycircuits.com/>

4 Resultados e Contribuições

O desenvolvimento da pesquisa resultou em aplicativo móvel multiplataforma com uma arquitetura personalizada. Apresentado na Seção 4.1, o aplicativo coleta dados inerciais de múltiplos sensores em tempo real por comunicação sem fio. Como também, permite processar estes dados em tempo real, por meio de métodos de processamento customizados em que os algoritmos são desacoplados da estrutura principal do projeto (Seção 4.2). Também foi produzido um protótipo pequeno, portátil e maleável, compatível com múltiplos sensores para enviar dados para o aplicativo móvel (Seção 4.3). Por fim, todas essas produções estão em um repositório público de código aberto que contém diversos documentos de apoio, como detalha a Seção 4.4.

4.1 Aplicativo móvel

O aplicativo entregue por esta pesquisa possui um foco grande no padrão arquitetônico do projeto para que a adição de funcionalidades e métodos de processamento seja facilitada, e ao mesmo não prejudique a longevidade e manutenibilidade do projeto. A versão atual do aplicativo é bem simples e conta com relativamente poucos casos de uso, exibidos na Figura 45.

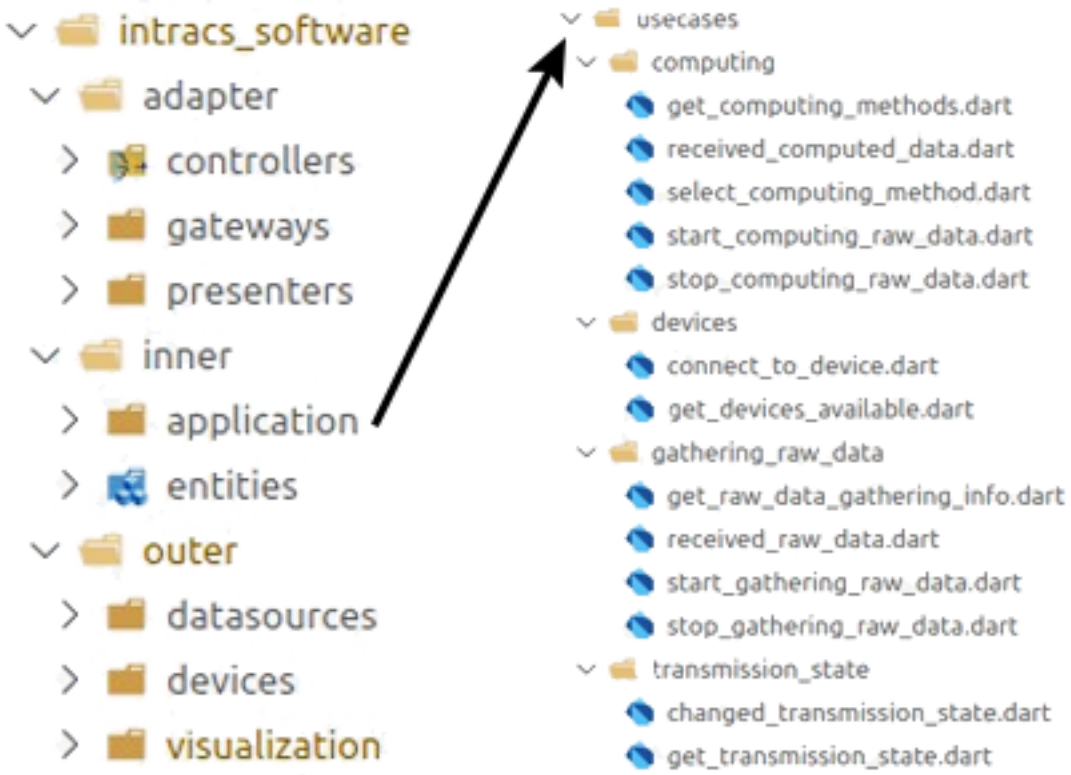
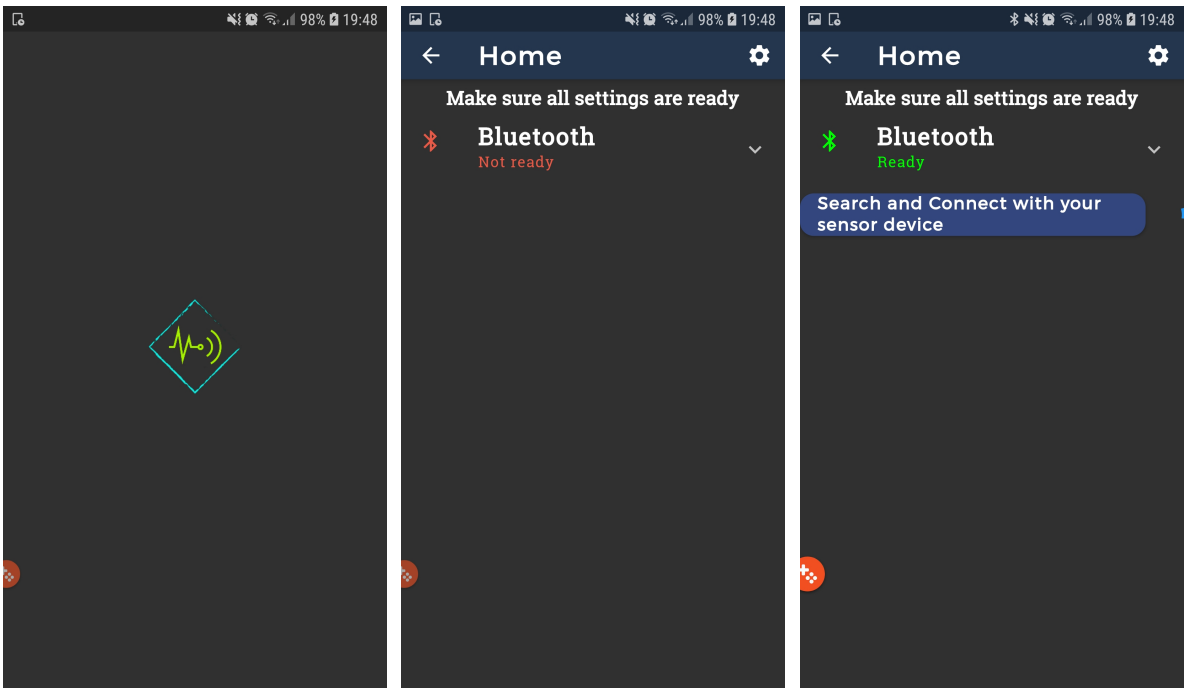


Figura 45 – Casos de uso existentes na camada *Application*.

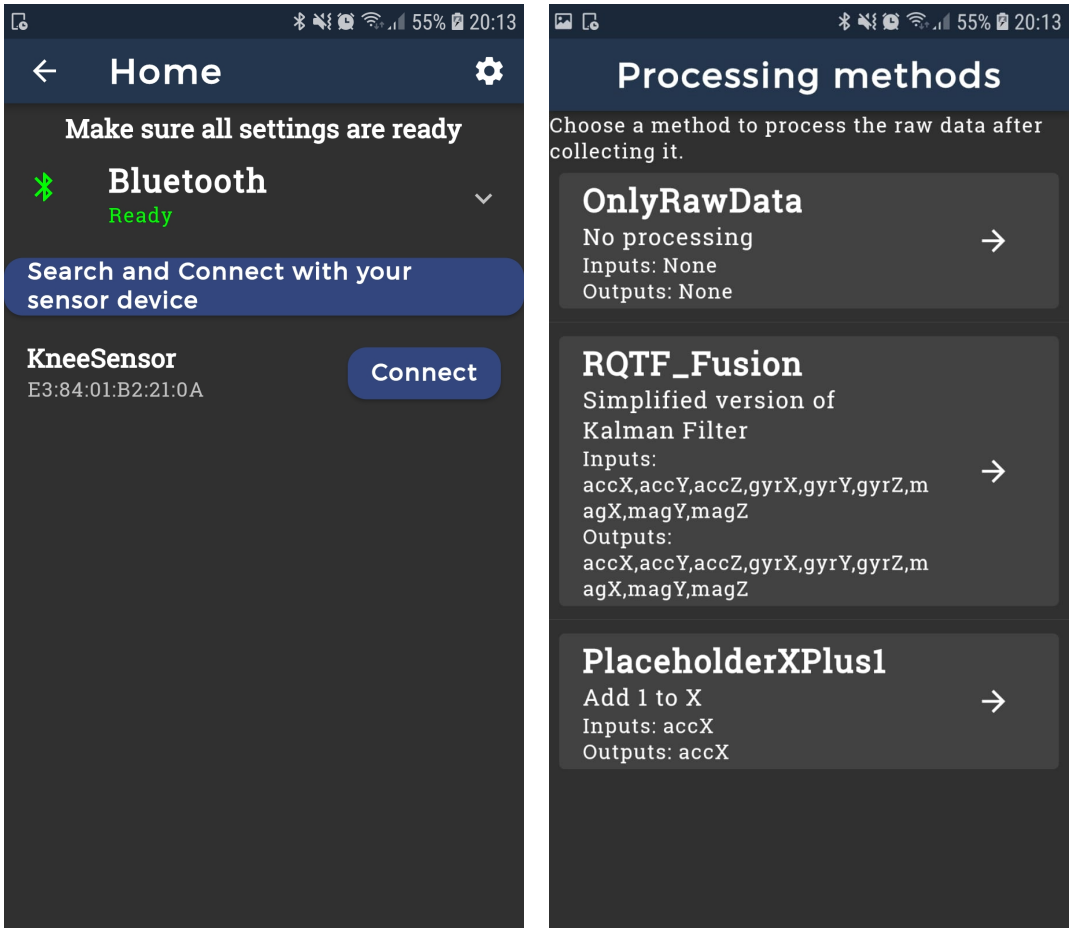
Apenas lendo os nomes dos arquivos dos casos de uso já é possível imaginar o que o aplicativo faz, inicialmente. Ao abrir o aplicativo é feita a injeção de dependência como explicado na Subseção 2.2.1.4 e a inicialização dos sub-sistemas e *frameworks* necessários, por isso é apresentado ao usuário uma tela de abertura com o logo do aplicativo em uma animação de carregamento, como exibido na Figura 46a. Após isso a primeira tela é apresentada (Figura 46b) exibindo o estado atual do Bluetooth. Caso esteja desligado, ao ligar o Bluetooth do *smartphone* a tela atualiza automaticamente e já realiza a busca dos dispositivos Bluetooth ativos que estão próximos e aguardando conexão, como exibido na Figura 46c. As telas exibidas na Figura 46 usam dois casos de uso: o *GetTransmissionState* que ao ser executado retorna o estado atual da tecnologia de transmissão e; o *ChangedTransmissionState* que é acionado pelo *smartphone* toda vez que há uma atualização no estado da transmissão da tecnologia.



(a) (b) (c)

Figura 46 – Telas inicias do aplicativo: a) Tela de abertura com animação de carregamento; b) Primeira tela do aplicativo com Bluetooth desligado e; c) Primeira tela do aplicativo com Bluetooth ligado, buscando dispositivos.

Com o Bluetooth ativo, um terceiro caso de uso é executado, o *GetDevicesAvailable*, que retorna uma listagem dos dispositivos que estão próximos, ativos e aguardando conexão como exibido na Figura 47a. O usuário então pode selecionar o dispositivo corretamente e solicitar a conexão por meio do botão *Connect*, que aciona o caso de uso *ConnectToDevice* que é responsável por realizar a conexão. Quando o dispositivo Bluetooth aceita a conexão com um retorno positivo, o caso de uso retorna com sucesso e o usuário é redirecionado para próxima tela, a de seleção de um método de processamento Figura 47b.

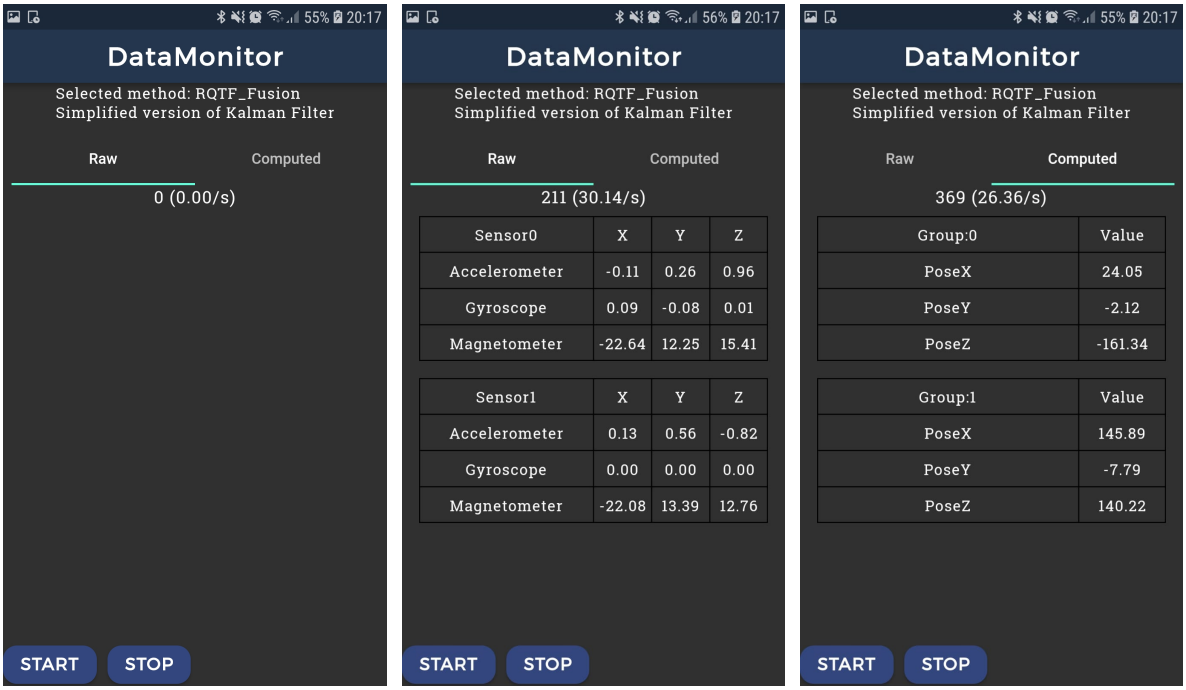


(a) (b)

Figura 47 – Telas de: a) conexão com um dispositivo e; b) lista de métodos de processamento disponíveis.

Os métodos listados na Figura 47b são os métodos de processamento (classes) que foram implementados na camada externa no padrão correto e possuem sua instância concreta registradas na lista estática de métodos existentes. Esta lista é recuperada pelo caso de uso *GetComputingMethods*. Ao clicar no botão para selecionar um método de processamento, o usuário dispara o caso de uso *SelectComputingMethod* que é responsável por fazer a seleção da instância concreta correta do método de processamento que é conectada a um motor de processamento. A Seção 4.2 apresenta mais detalhes sobre a estrutura interna do processamento customizado de dados.

Após selecionar o método, o usuário é redirecionado para a tela de inicio de coleta e processamento. Na Figura 48a é apresentado ao usuário o método selecionado, duas abas que são utilizadas para exibir os dados coletados e processados e por fim, botões para iniciar e parar a coleta e processamento dos dados. O botão *Start* executa os casos de uso *StartGatheringRawData* e *StartComputingRawData*, após isso o usuário pode optar por pausar a coleta e processamento, o botão *Stop* executa os casos de uso *StopGatheringRawData* e *StopComputingRawData*.



(a) (b) (c)

Figura 48 – Telas de coleta e processamento de dados inerciais: a) tela inicial; b) dados brutos sendo coletados em tempo real e; c) dados brutos processados em tempo real.

Por mais que a responsabilidade destes casos de uso sejam simples, pois eles só devem iniciar ou interromper a coleta e processamento de dados, toda a estrutura do sistema é acionada por estes, sendo a funcionalidade principal do aplicativo e já detalhada na Subseção 3.3.1. Durante a coleta e processamento múltiplos casos de uso são ativados pelo sistema, o caso de *GetRawDataGatheringInfo* é acionado periodicamente para recuperar as informações sobre a coleta exibidas na Figura 48b. O caso de uso *ReceivedRawData* é acionado por seu respectivo controlador de evento da camada *Adapters* quando um dado inercial é recebido, assim o dado é exibido como mostra a Figura 48b. Por fim o caso de uso *ReceivedComputedData* é similar, executado pelo controlador de eventos que é acionado pelo motor de processamento após o método selecionado retornar os dados processados, que são exibidos na Figura 48c.

Em suma, o aplicativo móvel faz uso do Bluetooth do *smartphone*, orientando visualmente o usuário para ligar caso necessário e então lista os dispositivos Bluetooth próximos que estão ativos e aguardando conexão. Com a lista de dispositivos exibida na tela, o usuário pode escolher qual dispositivo conectar. Ao se conectar, o aplicativo exibe os métodos de processamento disponíveis. Por fim, ao selecionar o método desejado o aplicativo redireciona o usuário para tela que o permite iniciar ou parar a coleta e processamento dos dados, e conforme os dados são coletados e processados eles são exibidos em tempo real no aplicativo para o usuário.

4.2 Métodos de processamento customizados

Este projeto possibilita contribuições de métodos de processamento customizados, que são algoritmos diversos que irão receber os dados inerciais sendo coletados e produzir um resultado processado. Como é um projeto e repositório em código aberto e possui uma licença abrangente, os contribuidores podem criar uma cópia do repositório (*fork*) para testarem seus próprios algoritmos e, por conseguinte, optar por contribuir com esse método para o repositório oficial.

A arquitetura do projeto permite que a contribuição de um método seja feita apenas com a adição de uma classe na camada externa do projeto, sem a necessidade de compreender toda a estrutura interna do projeto. Principalmente, não existe a obrigatoriedade de conhecer os diversos *frameworks* e dependências existentes no projeto, ou seja, a contribuição dos métodos necessita apenas de lógica de programação e um conhecimento básico de sintaxe da linguagem Dart.

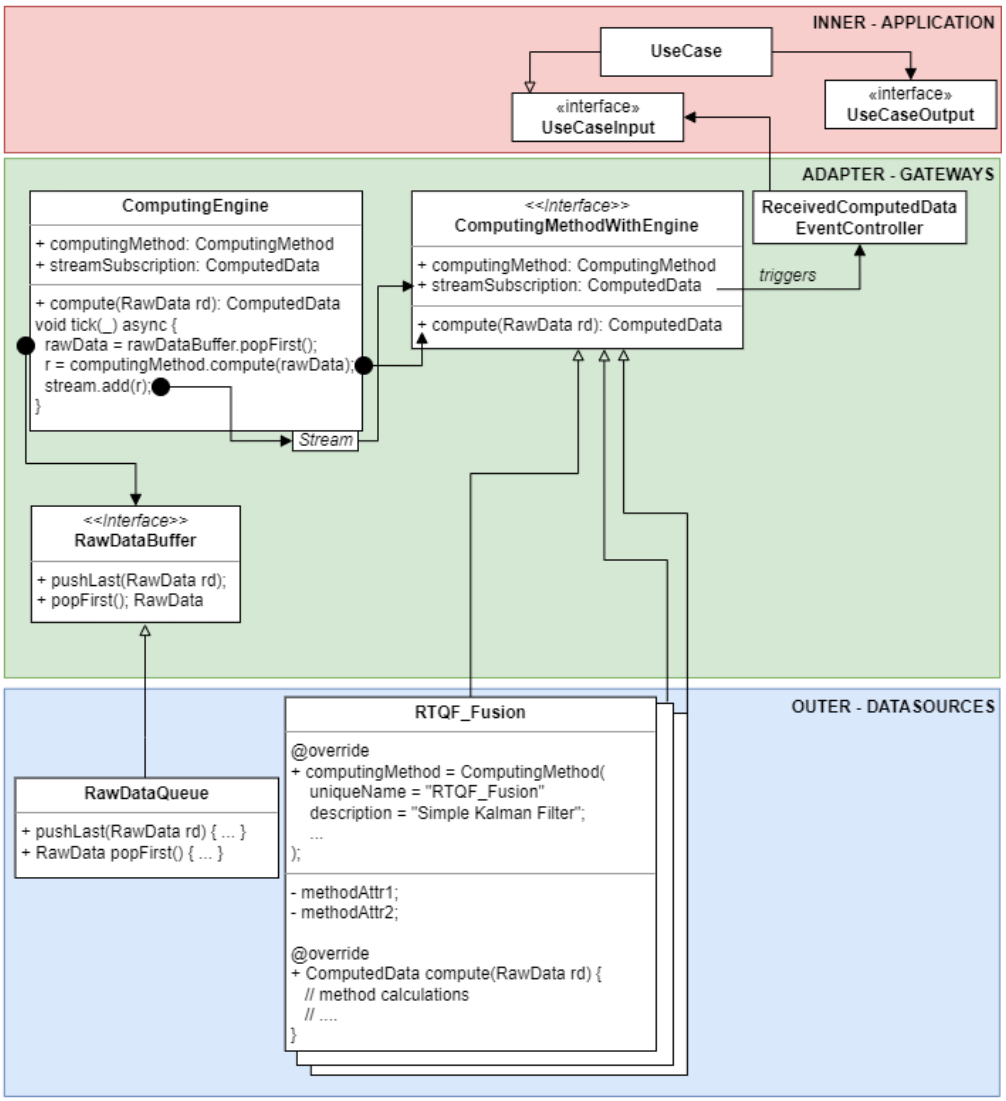


Figura 49 – Padrão *Strategy* dos métodos de processamento customizados.

Como ilustrado na Figura 49 os métodos de processamento são implementados como classes na camada externa seguindo o padrão de projeto comportamental *Strategy*. A interface abstrata que define o padrão a ser seguido é chamada de *Strategy* por Gamma et al. (1996) e neste trabalho é a classe *ComputingMethodWithEngine* que existe na camada *Adapters*. A classe *ComputingMethodWithEngine* (*Strategy*) é implementada por uma *ConcreteStrategy* que pode variar de acordo com o contexto. Estas implementações concretas são os métodos de processamento customizados existentes na camada externa.

Note que só a aplicação do padrão de projeto não resolve todo o fluxo de dados que tem que ser passado pela arquitetura, pois "algo" tem que fazer uso do padrão para que os diversos algoritmos sejam executados. Para isto, foi utilizado um recurso de programação conhecido como *Stream* assíncrona de dados, que basicamente pode ser considerada como uma sequência de eventos assíncronos, ou seja, em vez de pedir implicitamente o próximo evento, a *Stream* avisa que há um evento pronto para ser utilizado. Este "aviso" é enviado aos assinantes (*Subscribers*) dessa *Stream*.

A inscrição a uma *Stream* é chamada de *Subscription* e é uma função (algoritmo) que será executada quando a *Stream* assíncrona emitir um evento. Neste projeto o motor de processamento (ComputingEngine) é uma *Stream* assíncrona que consome os dados do *Buffer* de dados inerciais, os processa por meio do método selecionado e após isso os dados processados são emitidos. Assim as inscrições (funções) recebem estes dados por parâmetro. Observe na Figura 49 que a interface *ComputingMethodWithEngine* possui uma referência ao motor de processamento *ComputingEngine* e, a inscrição que foi feita na *Stream* do motor de processamento.

```
@override
Future<Result<Exception, bool>> startComputingRawData() async {
  // ...
  // creates the engine for the method
  selectedComputingMethod!.engine = ComputingEngine(
    selectedComputingMethod!,
    _rawDataBuffer,
  );

  // creates the subscription that will trigger the event controller
  selectedComputingMethod!.subscription =
      selectedComputingMethod!.engine!.stream.listen(
      (computedData) {
        _computedDataEventController.register(Success(computedData));
      },
    );
  }
  // ...
}
```

Figura 50 – Inscrição na *Stream* de dados ao selecionar o método de processamento.

Como exibido na Figura 50, a criação do motor de processamento e a inscrição a *Stream* de dados é feita no momento do inicio do processamento de dados, que é ativado pelo caso de uso *StartComputingRawData*. A inscrição na *Stream* do motor de processamento é um algoritmo que irá receber os dados processados quando prontos e entregar ao controlador de evento responsável por ativar o caso de uso respectivo (*ReceivedComputedData*), assim o dado processado trafega pela arquitetura até ser exibido na tela para o usuário.

Existe uma diferença entre o padrão de projeto *Strategy* original proposto por Gamma et al. (1996) e o padrão aplicado neste projeto, pois no original o objeto que faz uso das implementações concretas é fixo e apenas o contexto se altera (referência à instância concreta). Neste projeto é criado um motor de processamento para cada método uma única vez durante a seleção, porém a arquitetura deste projeto permite a possibilidade de que mais de um método seja selecionado, podendo contrastar o mesmo dado processado entre diferentes métodos.

4.3 Protótipo inercial de baixo custo

Este projeto está totalmente disponível em código aberto e gratuito. Para isto, também foi desenvolvido um protótipo de dispositivo inercial com aporte a múltiplos sensores e de baixo custo. O custo total de aquisição e montagem deste dispositivo é menor do que alternativas proprietárias disponíveis no mercado como contrastado na Tabela 1. Entretanto, como se trata de peças importadas e de tamanho reduzido, o custo deste protótipo pode ser reduzido futuramente com a criação e aquisição de componentes no mercado brasileiro.

Tabela 1 – Opções de dispositivos para coleta de dados inerciais com transmissão sem fio.

Marca	**Nome**	**Preço Aprox.**	**Qtd. Sensores Suportados**	**Repositório de código**
-	INTRACS (peças TinyCircuits)	80 USD (com 1 sensor)	Até 32 (15 USD por sensor)	Público
MBIENTLAB	MetaMotionRL+	103 USD (por sensor)	-	Privado
YOSTLABS	3-Space™ Bluetooth	160 USD (por sensor)	-	Privado
Delsys	Mobile EMG Suite	3000 USD[1] (com 2 sensores)	Até 4	Privado
GaitUp	GaitUp Lab (2x Pyshilog 5)	7500 EUR[1] (com 2 sensores)	- (755 EUR por sensor)	Privado

Fonte – Produzido pelo autor. [1] Preços consultados em agosto de 2019.

Todas as opções exibidas na Tabela 1 são capazes de coletar dados inerciais por transmissão sem fio. Os dispositivos 3-Space™ Bluetooth[1] e MetaMotionRL+[2] possuem aplicativo móvel gratuito porém não há opções de processamento dos dados inclusas. Os dispositivos que possuem o custo mais elevado, como o Mobile EMG Suite[3] da Delsys, que também inclui sensores de EMG (Eletromiografia) além dos sensores inerciais, e o Physilog 5 da GaitUp[4], ambos possuem sistemas mais completos que fornecem até certo grau de processamento dos dados com métodos pré-definidos. Além disso, todas as opções da Tabela 1 não fornecem um repositório de código aberto e de livre contribuição, como disponibilizado por esta pesquisa.

4.3.1 Sistema do protótipo inercial

A primeira etapa do aplicativo produzido é a conexão com o dispositivo. Ao ligar o dispositivo, ocorre uma inicialização de valores e variáveis, preparação e teste dos componentes e calibração do giroscópio dos sensores inerciais de nove eixos. Após isso, o componente de Bluetooth se coloca em espera aguardando uma conexão. O aplicativo então lista os dispositivos inerciais passíveis de conexão e o usuário pode selecionar o correto para se conectar. Após a conexão, o dispositivo aguarda apenas duas mensagens de comando, "#START" para iniciar o envio de dados inerciais e "#STOP" para interromper. Estas mensagens são enviadas quando o usuário decide iniciar ou parar a coleta e processamento de dados, botão ilustrado na Figura 28, tela B. O código de envio está presente na camada externa de dispositivos, na classe que implementa a comunicação entre o dispositivo e o *smartphone*.

Por fim, o sistema também conta com um algoritmo que ajusta a taxa de transmissão de dados com base na frequência dos erros de pacotes enviados que não foram recebidos. A taxa de transmissão é aumentada caso não ocorram erros após uma certa quantidade de dados enviados com sucesso, e reduzida caso uma quantidade de erros apareça durante o envio desta mesma quantidade. Esta rotina é essencial para estimar a taxa de transferência obtida pelo aplicativo, que é explicada na Subseção 4.3.2 juntamente com a estrutura de transmissão adotada para os dados.

4.3.2 Transferência de dados sem fio

Como este protótipo apenas realiza o recebimento e envio de mensagens, não é necessário um processador mais avançado e, em paralelo com a tecnologia de Bluetooth Low Energy 4.1 utilizada para realizar a transferência dos dados inerciais, possibilita um gasto de energia reduzido. Nos testes não controlados realizados durante o desenvolvimento

[1] <https://yostlabs.com/product/3-space-bluetooth/>
[2] <https://mbientlab.com/store/metamotionrl-p/>
[3] <https://www.delsys.com/mobile-emg-suite/>
[4] <https://shop.gaitup.com/>

deste projeto, o protótipo conseguiu atingir no mínimo 20 minutos de transferência de dados contínua com uma bateria de íon de lítio pequena, de 290mAh.

A taxa de transmissão média atingida pela tecnologia e este protótipo foi estimada em 50 pacotes por segundo, como cada pacote contém 20 bytes e representa os dados de identificação e coleta de um sensor, conclui-se uma taxa de transmissão de 1 kilobyte por segundo, que se mostra de acordo com os valores apresentados no trabalho de Bulić, Kojek e Biasizzo (2019). Na tecnologia BLE 4.1, o tamanho da seção de dados do pacote a ser enviado não deve ultrapassar 20 bytes de acordo com a especificação do componente. Com 20 bytes por mensagem, a estrutura do pacote foi definida conforme ilustrado na Figura 51.

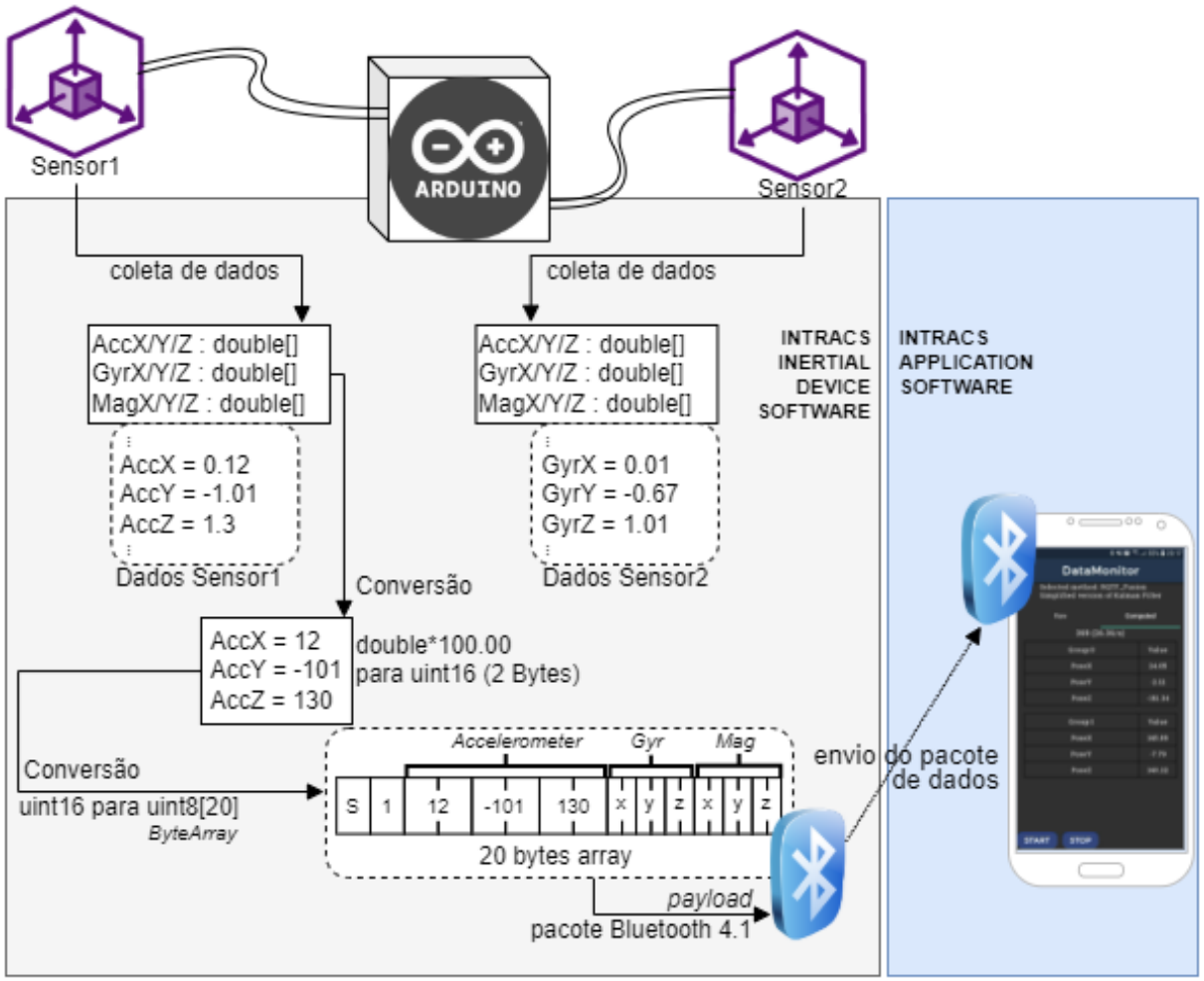


Figura 51 – Estrutura de dados da mensagem enviada por Bluetooth.

Como a taxa de transferência da tecnologia foi estimada em 1 kilobyte por segundo, ou 50 pacotes com dados inerciais por segundo, cada sensor adicionado irá partilhar desta taxa de transferência. O protótipo atual possui dois sensores inerciais de nove eixos iguais, em que seus dados são enviados sequencialmente logo após a coleta. Os dados do sensor um são convertidos para bytes na estrutura mencionada e enviados, consecutivamente os dados do sensor dois são enviados, e então ocorre uma nova coleta de dados, repetindo

o processo. Este sistema com cinco sensores por exemplo estaria limitado a 10 amostras de dados inerciais por segundo, pois cada amostra contém os dados de cinco sensores diferentes (cinco mensagens enviadas por Bluetooth).

De acordo com Bulić, Kojek e Biasizzo (2019), a atualização de tecnologia para a versão 5.0 do Bluetooth Low Energy resultaria em um aumento na taxa de transmissão de 1 kilobyte por segundo para 12.5 kilobytes por segundo. Como o pacote deste projeto possui 20 bytes, esta atualização de tecnologia forneceria a possibilidade de enviar 625 pacotes com dados inerciais por segundo, um aumento significativo quando comparado aos 50 pacotes da tecnologia atual.

Note que a taxa de coleta de dados é limitada pela taxa de transmissão. É possível coletar mais amostras de dados inerciais caso o dispositivo inercial conte com algum meio de armazenamento interno, como um cartão MicroSD por exemplo, porém, os dados ainda estariam sujeitos a sincronização pela taxa de transferência da tecnologia sem fio. Outro modo seria a remoção manual do cartão de armazenamento para passar os dados para um sistema, contudo, como a proposta deste projeto é fazer uso da portabilidade e da transmissão sem fio, a melhor alternativa seria a atualização da tecnologia BLE 4.1 para BLE 5.0, ou alteração para *Wi-Fi*, no entanto esta pode ter grande impacto no consumo da bateria.

4.4 Código aberto e repositório público

Esta pesquisa tem como um de seus objetivos criar um projeto de fácil contribuição, com um código e arquitetura que incentive o engajamento da comunidade para o crescimento do projeto. Sistemas em código aberto (OSS - *Open Source Software*) permitem aos usuários usar, alterar e aperfeiçoar o projeto e, de acordo com a licença aplicada, também autoriza a distribuição do subproduto em uma forma modificada ou não. Ademais e de certo modo projetos em código aberto aplicam um nível de influência na sociedade, considerada necessária para padronizações e inovações em tecnologias (BITZER; SCHRÖDER, 2006; ABERDOUR, 2007; TEIGLAND et al., 2014; YONG et al., 2021).

Apenas disponibilizar um projeto em código aberto em um repositório público com contribuições abertas para a comunidade externa não é o suficiente. Existem diversas barreiras de entrada que impedem um possível contribuidor de se engajar em um projeto deste tipo e, muitas destas são relacionadas ao suporte e aspectos sociais da comunidade envolvida. Em contrapartida, muitas outras estão relacionadas aos aspectos técnicos do projeto, como documentação em vários níveis (de arquitetura, de contribuição, comentários de código, estrutura do projeto) e em diferentes idiomas, baixa qualidade de código, dificuldade para contribuição, dificuldade na configuração inicial, entre outras (VIR; MING; YONG, 2007; PADHYE; MANI; SINHA, 2014; STEINMACHER et al., 2015).

Logo, com o intuito de melhorar a qualidade e incentivar contribuições, este projeto conta com diversos documentos de apoio, seguindo as diretrizes de repositório público estabelecidas pela Creative Commons[5] e GitHub[6], e conjuntamente adere a exemplos de sucesso presentes em repositórios públicos de grande engajamento como Flutter[7] (*framework* usado por este projeto) e EthicalSource[8].

4.4.1 Documentação e documentos de apoio

Os documentos existentes no repositório público[9] fornecem uma grande variedade de informações sobre o projeto, primeiramente os documentos AUTHORS e CITATION.cff disponibilizam informações sobre os autores do projeto e habilitam a citação do repositório em formatos acadêmicos. Outro documento importante é o arquivo CODEOWNERS que define os responsáveis pelo repositório, pois diferentemente dos autores, estes são notificados quando existem pendências no repositório, como analisar as sugestões de alterações enviadas pela comunidade (*pull requests*).

O arquivo README.md é o documento de apoio mais importante do projeto pois ele é exibido na tela inicial do repositório e realiza uma apresentação em alto nível do projeto orientando também a leitura dos outros documentos, e inclusive documenta a instalação e configuração do projeto localmente. Em paralelo, os arquivos GET_STARTED.md e CONTRIBUTING.md são guias iniciais para novos colaboradores que quiserem contribuir com o projeto. Nestes documentos são definidas leituras essenciais antes de realizar uma contribuição ao repositório, como a instalação e configuração do ambiente de desenvolvimento, como abrir *Issues* para relatar problemas e sugestões, entre outras.

Após isto, o documento CODE_OF_CONDUCT.md define o código de conduta e diretrizes sociais para serem seguidas durante o ciclo de vida do projeto em contribuições e interações com a comunidade. O arquivo LICENSE define o tipo de licença pública aplicada no projeto. A licença determina restrições e condições em características como compartilhamento, alteração de código, distribuição e uso comercial, vendas, entre outros. A licença escolhida por este projeto é a MIT[10] por ser bastante usada na comunidade de código aberto e ser implicitamente mais abrangente que a alternativa BSD 3.0[11] (três cláusulas). Um comparativo entre as opções que foram consideradas por este projeto é exibido na Tabela 2.

[5] <https://opensource.creativecommons.org/contributing-code/github-repo-guidelines/>
[6] <https://docs.github.com/en/repositories>
[7] <https://github.com/flutter/flutter>
[8] <https://github.com/EthicalSource>
[9] <https://github.com/brunotacca/INTRACS>
[10] <https://opensource.org/licenses/MIT>
[11] <https://opensource.org/licenses/BSD-3-Clause>

Tabela 2 – Comparativo entre licenças de código aberto consideradas para o projeto.

Licença	Versão	Tipo	Proteção copyright	Modificação	Uso proprietário (código fechado)	Patente explícita	Licenciamento
Apache	2.0	Conteúdo livre	Sim	Sim	Sim	Sim	Sim
BSD	3.0	Conteúdo livre	Sim	Sim	Sim	Não	Parcial
MIT	-	Conteúdo livre	Sim	Sim	Sim	Não	Sim
GPL	3.0	*Copyleft*	Sim	Sim	Não	Não	Sim
LGPL	3.0	*Copyleft*	Sim	Sim	Não	Não	Sim
CC0	1.0	Domínio Público	Não	Sim	Sim	Sim	Não

Fonte – Produzido pelo autor.

Adiante, o arquivo PROJECT_ARCHITECTURE.md detalha a arquitetura completa do projeto incluindo a divisão de camadas, componentes e suas respectivas responsabilidades como explicado na Seção 3.3. O arquivo CHANGELOG.md apresenta as alterações feitas entre os incrementos na versão do projeto, já o arquivo SUPPORT.md disponibiliza vias de contato para questões que não se encaixem no escopo do repositório. Por fim, os arquivos que definem os modelos de interação com o repositório: bug_report.md para comunicar bugs e problemas; feature_request.md para solicitar novas funcionalidades e; PULL_REQUEST_TEMPLATE.md para descrever as alterações de código que foram feitas na solicitação de integração das mesmas.

5 Conclusões e Trabalhos Futuros

Esta pesquisa apresentou o aplicativo INTRACS (*Inertial Tracker Computing System)*, um sistema multiplataforma que coleta dados inerciais de múltiplos sensores por comunicação sem fio, em tempo real, e ao mesmo tempo os processa com algoritmos personalizados chamados de métodos de processamento, em que o usuário do aplicativo pode selecionar o método que preferir. Também foi apresentado um protótipo de dispositivo inercial para a coleta e envio dos dados inerciais, sendo capaz de suportar até 32 unidades inerciais. O protótipo foi montado com componentes de relativamente baixo custo sendo de fácil aquisição e montagem, além de fornecer a possibilidade de atualização da tecnologia de comunicação isoladamente.

O aplicativo foi desenvolvido e estruturado com uma adaptação da arquitetura limpa, proposta por Martin (2017), tendo em vista fornecer um projeto com alta longevidade e manutenibilidade para futuros contribuidores. O projeto foi disponibilizado em um repositório público no GitHub[1] em um formato de código aberto (*OSS - Open Source Software*) e conta com uma licença MIT que fornece gratuidade e liberdade. Ademais, o repositório conta com diversos documentos de apoio em inglês para facilitar a entrada e engajamento de novos contribuidores.

A principal vantagem deste projeto é estar totalmente disponível em código aberto e, ao mesmo tempo, estar estruturado em uma arquitetura que permite o incremento e extensão de suas funcionalidades de maneira orgânica e natural. Em particular, é possível contribuir com métodos de processamento sem a necessidade de conhecer a estrutura toda do projeto, contribuindo apenas com o algoritmo do método em uma classe que segue os padrões estabelecidos.

5.1 Trabalhos futuros

O projeto serve como base para muitas aplicações futuras. A funcionalidade de coletar e processar dados inerciais em tempo real é utilizada em diversas áreas de conhecimento. Ciências da saúde por exemplo, pode fazer uso deste projeto em pesquisas de análise de parâmetros cinemáticos com uso de sensores inerciais. Similarmente, para engenharias e ciência exatas e da terra, com pesquisas relacionadas a robótica ou internet das coisas. Em suma, qualquer trabalho que tem por interesse o uso de dados inerciais processados em tempo real pode se beneficiar deste projeto.

[1] <https://github.com/brunotacca/INTRACS>

O aplicativo foi desenvolvido totalmente focado em sua estrutura interna e na sua funcionalidade principal, logo existem alguns pontos de melhoria que podem ser aplicados. A interface gráfica por exemplo não foi elaborada, o esquema de cores, botões, transições entre telas e animações são extremamente básicos ou não existentes. Existem também, algumas funcionalidades que podem ser adicionadas ao aplicativo como: a exportação dos dados inerciais para arquivos .csv; a internacionalização do aplicativo que no momento se encontra apenas em inglês e; a configuração de perfis e usuários diferentes, para que um mesmo conjunto (*smartphone* e dispositivo) possa ser usado por diferentes pessoas em uma mesma instituição.

Também é possível melhorar a visualização dos dados que estão sendo coletados e processados. Atualmente estes dados são estruturados em um formato tabular bem simples, e os valores são exibidos diretamente sem nenhum fator temporal. Logo, é possível a adição de um gráfico de valor-tempo como o exibido na Figura 18 por exemplo. Além disso, como a estrutura da arquitetura fornece um desacoplamento em vários níveis, também é possível a integração com APIs de outros sistemas, como o OpenSim.

No momento, o aplicativo só permite a seleção de um método de processamento, porém, é possível alterar a arquitetura para permitir a seleção de múltiplos métodos. Com isso, existem muitos métodos de processamento que podem ser implementados, as diferentes versões do Filtro de Kalman por exemplo poderiam ser usadas como métodos de redução de ruído. Ademais, trabalhos que sugerem algoritmos de processamento em cima de sensores inerciais como os de Qiu et al. (2019), Bötzel et al. (2018) e Das, Hooda e Kumar (2019) também poderiam ser adicionados.

Apesar do *hardware* do protótipo ser de baixo custo em comparação com as alternativas existentes, as peças possuem um tamanho reduzido favorecendo a portabilidade e tiveram que ser importadas, com isso acredita-se que seja possível a redução do custo com a criação de um protótipo com peças que podem ser encontradas no mercado local, como o Arduino Pro Mini, Arduino Fio ou Arduino Nano. Embora exista uma possível redução da portabilidade atual do protótipo, a redução do custo total do protótipo multissensor pode ser beneficial para outros estudos.

Referências

ABERDOUR, M. Achieving quality in open-source software. *IEEE Software*, v. 24, n. 1, p. 58–64, jan 2007. ISSN 07407459. Citado na página 87.

ABYARJOO, F. et al. Implementing a sensor fusion algorithm for 3d orientation detection with inertial/magnetic sensors. In: SOBH, T.; ELLEITHY, K. (Ed.). *Innovations and Advances in Computing, Informatics, Systems Sciences, Networking and Engineering*. Cham: Springer International Publishing, 2015. p. 305–310. ISBN 978-3-319-06773-5. Citado 3 vezes nas páginas 10, 42 e 43.

AGOSTINI, V. et al. Wearable sensors for gait analysis. In: *2015 IEEE International Symposium on Medical Measurements and Applications, MeMeA 2015 - Proceedings*. [S.l.]: IEEE, 2015. p. 146–150. ISBN 9781479964765. Citado 2 vezes nas páginas 15 e 37.

ALCARAZ, J. C. et al. Machine learning as digital therapy assessment for mobile gait rehabilitation. In: *IEEE International Workshop on Machine Learning for Signal Processing, MLSP*. [S.l.]: IEEE, 2018. v. 2018-September, p. 1–6. ISBN 9781538654774. ISSN 21610371. Citado na página 16.

AMINIAN, K. et al. Spatio-temporal parameters of gait measured by an ambulatory system using miniature gyroscopes. *Journal of Biomechanics*, v. 35, n. 5, p. 689 – 699, 2002. ISSN 0021-9290. Citado na página 40.

ANICHE, M. et al. A validated set of smells in model-view-controller architectures. *Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016*, Institute of Electrical and Electronics Engineers Inc., p. 233–243, jan 2017. Citado na página 66.

ANTINK, C. H.; BRÜSER, C.; LEONHARDT, S. Multimodal sensor fusion of cardiac signals via blind deconvolution: A source-filter approach. In: *Computing in Cardiology 2014*. [s.n.], 2014. v. 41, n. January, p. 805–808. ISBN 9781479943463. ISSN 2325887X. Disponível em: <https://ieeexplore.ieee.org/document/7043165>. Citado na página 44.

ASHTON, J. The design of commercial hearing aids. *Journal of the British Institution of Radio Engineers*, v. 11, n. 2, p. 51–59, feb 1951. ISSN 2054-054X. Citado na página 15.

BARRETT, S. F. *Arduino I: Getting Started*. [S.l.]: Morgan & Claypool Publishers, 2020. v. 15. 1–222 p. ISSN 19323174. ISBN 978-1681738208. Citado na página 46.

BARRETT, S. F. *Arduino III: Internet of Things*. [S.l.]: Morgan & Claypool Publishers, 2021. v. 16. 1–237 p. ISSN 19323174. ISBN 978-1636390833. Citado na página 46.

BEEBY, S. et al. *MEMS Mechanical Sensors*. [S.l.]: Artech House, 2013. v. 53. 1689–1699 p. ISSN 1098-6596. ISBN 9788578110796. Citado 2 vezes nas páginas 36 e 41.

BITZER, J. D.; SCHRÖDER, P. J. The Impact of Entry and Competition by Open Source Software on Innovation Activity. *The Economics of Open Source Software Development*, Elsevier, p. 219–246, jan 2006. Citado na página 87.

BÖTZEL, K. et al. Quantification of gait parameters with inertial sensors and inverse kinematics. *Journal of Biomechanics*, Elsevier Ltd, v. 72, p. 207–214, apr 2018. ISSN 18732380. Citado 2 vezes nas páginas 45 e 91.

BOUKHARY, S.; COLMENARES, E. A clean approach to flutter development through the flutter clean architecture package. *Proceedings - 6th Annual Conference on Computational Science and Computational Intelligence, CSCI 2019*, Institute of Electrical and Electronics Engineers Inc., p. 1115–1120, dec 2019. Citado na página 47.

BOUTAAYAMOU, M. et al. Validated extraction of gait events from 3D accelerometer recordings. In: *2012 International Conference on 3D Imaging, IC3D 2012 - Proceedings*. [S.l.]: IEEE, 2012. p. 1–4. ISBN 9781479915804. Citado na página 39.

BROWN, P. et al. Refactoring the whitby intelligent tutoring system for clean architecture. *Theory and Practice of Logic Programming*, v. 21, p. 1–17, 11 2021. Citado na página 47.

BUENO, C. E. d. O. *Desenvolvimento de um Aplicativo Utilizando o Framework Flutter e Arquitetura Limpa*. [S.l.], 2021. Disponível em: <https://repositorio.pucgoias.edu.br/jspui/handle/123456789/1861>. Citado na página 48.

BULIć, P.; KOJEK, G.; BIASIZZO, A. Data transmission efficiency in bluetooth low energy versions. *Sensors*, v. 19, n. 17, 2019. ISSN 1424-8220. Citado 2 vezes nas páginas 86 e 87.

CHEN, S. et al. Toward Pervasive Gait Analysis With Wearable Sensors: A Systematic Review. *IEEE Journal of Biomedical and Health Informatics*, v. 20, n. 6, p. 1521–1537, nov 2016. ISSN 21682194. Citado na página 37.

COCKBURN, A. *Hexagonal architecture*. 2005. <https://alistair.cockburn.us/hexagonal-architecture/>. Acesso em 2021/06/10. Citado 2 vezes nas páginas 28 e 65.

DAS, R.; HOODA, N.; KUMAR, N. A Novel Approach for Real-Time Gait Events Detection Using Developed Wireless Foot Sensor Module. *IEEE Sensors Letters*, v. 3, n. 6, 2019. ISSN 24751472. Citado na página 91.

DEHZANGI, O.; TAHERISADR, M.; CHANGALVALA, R. IMU-based gait recognition using convolutional neural networks and multi-sensor fusion. *Sensors (Switzerland)*, v. 17, n. 12, p. 2735, nov 2017. ISSN 14248220. Citado 2 vezes nas páginas 16 e 40.

DELLIAN, E. Newton on mass and force: a comment on Max Jammer's Concepts of Mass (1961; 2000). *Physics Essays*, Physics Essays Publication, v. 16, n. 2, p. 264–277, jun 2003. ISSN 08361398. Citado na página 36.

DELP, S. L. et al. OpenSim: Open-source software to create and analyze dynamic simulations of movement. *IEEE Transactions on Biomedical Engineering*, v. 54, n. 11, p. 1940–1950, nov 2007. ISSN 00189294. Citado 3 vezes nas páginas 16, 46 e 47.

DOHENY, E. P.; FORAN, T. G.; GREENE, B. R. A single gyroscope method for spatial gait analysis. In: *2010 Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC'10*. [S.l.]: IEEE, 2010. p. 1300–1303. ISBN 9781424441235. Citado na página 40.

FOWLER, M. *Refactoring: improving the design of existing code*. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2018. 431 p. ISBN 9780134757681. Citado na página 19.

FOWLER, M. et al. *Patterns of Enterprise Application*. [S.l.]: Addison-Wesley Professional, 2002. 560 p. ISBN 0321127420. Citado 2 vezes nas páginas 19 e 28.

FUNG, M. L.; CHEN, M. Z.; CHEN, Y. H. Sensor fusion: A review of methods and applications. In: *Proceedings of the 29th Chinese Control and Decision Conference, CCDC 2017*. [S.l.]: IEEE, 2017. p. 3853–3860. ISBN 9781509046560. Citado 2 vezes nas páginas 43 e 44.

GAMMA, E. et al. *Design Patterns. Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley Professional, 1996. 395 p. ISBN 020163361-2. Citado 10 vezes nas páginas 9, 19, 20, 23, 24, 25, 35, 36, 83 e 84.

GARLAN, D.; SHAW, M. An Introduction to Software Architecture. *World Scientific Publishing Company*, 1994. Citado na página 28.

GONZALEZ, R. C. et al. Real-time gait event detection for normal subjects from lower trunk accelerations. *Gait & Posture*, v. 31, n. 3, p. 322 – 325, 2010. ISSN 0966-6362. Citado na página 40.

GOULERMAS, J. Y. et al. An instance-based algorithm with auxiliary similarity information for the estimation of gait kinematics from wearable sensors. *IEEE Transactions on Neural Networks*, v. 19, n. 9, p. 1574–1582, sep 2008. ISSN 10459227. Citado 2 vezes nas páginas 15 e 16.

GOUWANDA, D.; SENANAYAKE, S. M. Identification of gait asymmetry using wireless gyroscopes. In: *IEEE Asia-Pacific Conference on Circuits and Systems, Proceedings, APCCAS*. [S.l.]: IEEE, 2010. p. 608–611. ISBN 9781424474561. Citado na página 40.

GRAVINA, R. et al. Multi-sensor fusion in body sensor networks: State-of-the-art and research challenges. *Information Fusion*, Elsevier B.V., v. 35, p. 1339–1351, may 2017. ISSN 15662535. Citado 2 vezes nas páginas 44 e 45.

HERSHBERGER, D. Wearable communication devices. In: *Case Studies in Medical Instrument Design 1991*. [S.l.]: IEEE, 1991. p. 200–203. ISBN 0780306279. Citado 2 vezes nas páginas 15 e 36.

HUANG, Y. et al. Novel Foot Progression Angle Algorithm Estimation via Foot-Worn, Magneto-Inertial Sensing. *IEEE Transactions on Biomedical Engineering*, v. 63, n. 11, p. 2278–2285, nov 2016. ISSN 15582531. Citado na página 42.

HUMPHERYS, J.; REDD, P.; WEST, J. A fresh look at the kalman filter. *SIAM Review*, Society for Industrial and Applied Mathematics, v. 54, n. 4, p. 801–823, nov 2012. ISSN 00361445. Citado na página 44.

JARCHI, D. et al. A Review on Accelerometry-Based Gait Analysis and Emerging Clinical Applications. *IEEE Reviews in Biomedical Engineering*, v. 11, p. 177–194, 2018. ISSN 19411189. Citado 2 vezes nas páginas 37 e 39.

JUNG, P. G.; LIM, G.; KONG, K. A mobile motion capture system based on inertial sensors and smart shoes. In: *Proceedings - IEEE International Conference on Robotics and Automation*. [S.l.]: IEEE, 2013. p. 692–697. ISBN 9781467356411. ISSN 10504729. Citado na página 44.

KALMAN, R. E. A new approach to linear filtering and prediction problems. *Journal of Fluids Engineering, Transactions of the ASME*, 1960. ISSN 1528901X. Citado na página 45.

KAWANO, K. et al. Analyzing 3D knee kinematics using accelerometers, gyroscopes and magnetometers. In: *2007 IEEE International Conference on System of Systems Engineering, SOSE*. [S.l.]: IEEE, 2007. p. 1–6. ISBN 1424411602. Citado na página 42.

KEMPE, V. *Inertial MEMS: Principles and practice*. Cambridge: Cambridge University Press, 2011. v. 9780521766586. 1–479 p. ISBN 9780511933899. Citado 6 vezes nas páginas 36, 37, 38, 39, 40 e 42.

KHALEGHI, B. et al. Multisensor data fusion: A review of the state-of-the-art. *Information Fusion*, Elsevier, v. 14, n. 1, p. 28–44, jan 2013. ISSN 15662535. Citado 3 vezes nas páginas 43, 44 e 45.

KIMURA, E. N. R. *Desenvolvimento de um sistema de gestão de armazém para dispositivos móveis em Flutter*. [S.l.], 2019. Disponível em: <https://bibliotecadigital.ipb.pt/handle/10198/22955>. Citado na página 48.

KIZAKEVICH, P. N.; JOCHEM, W. J.; JONES, A. W. A personal cardiac bioimpedance monitoring system. In: *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. [S.l.: s.n.], 1988. p. 48–50 vol.1. Citado na página 15.

KORK, S. K. et al. Biometric database for human gait recognition using wearable sensors and a smartphone. In: *BioSMART 2017 - Proceedings: 2nd International Conference on Bio-Engineering for Smart Technologies*. [S.l.]: IEEE, 2017. p. 1–4. ISBN 9781538607053. Citado na página 16.

KUN, L. et al. Ambulatory estimation of knee-joint kinematics in anatomical coordinate system using accelerometers and magnetometers. *IEEE Transactions on Biomedical Engineering*, v. 58, n. 2, p. 435–442, feb 2011. ISSN 00189294. Citado na página 42.

LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. [S.l.]: Prentice Hall PTR, 2004. v. 17. 960 p. ISBN 0131489062. Citado 7 vezes nas páginas 19, 20, 21, 22, 23, 26 e 66.

LAU, H.; TONG, K. The reliability of using accelerometer and gyroscope for gait event identification on persons with dropped foot. *Gait & Posture*, v. 27, n. 2, p. 248 – 257, 2008. ISSN 0966-6362. Citado na página 40.

LAVIKAINEN, J. et al. Open-Source Software Library For Real-Time Inertial Measurement Unit Data-Based Inverse Kinematics Using OpenSim. *Research Square Platform LLC*, aug 2021. Disponível em: <https://doi.org/10.21203/rs.3.rs-775793/v2>. Citado 2 vezes nas páginas 16 e 46.

LÉTOURNEAU, D. et al. *OpenIMU - Data Analyser for Inertial Measurement Units and Actimetry Data (v0.6.5)*. 2021. Disponível em: <https://github.com/introlab/OpenIMU>. Acesso em: 7 out. 2021. Citado 3 vezes nas páginas 16, 46 e 47.

LI, Q. et al. Kalman filter and its application. In: *Proceedings - 8th International Conference on Intelligent Networks and Intelligent Systems, ICINIS 2015*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2016. p. 74–77. ISBN 9781479988228. Citado na página 44.

LISKOV, B. Keynote address - data abstraction and hierarchy. *ACM SIGPLAN Notices*, ACM PUB27 New York, NY, USA, v. 23, n. 5, p. 17–34, 1987. Disponível em: <https://dl.acm.org/doi/abs/10.1145/62139.62141>. Citado na página 26.

LIU, T. et al. A mobile force plate and three-dimensional motion analysis system for three-dimensional gait assessment. *IEEE Sensors Journal*, v. 12, n. 5, p. 1461–1467, may 2012. ISSN 1530437X. Citado na página 16.

LLINAS, J.; HALL, D. L. Introduction to multi-sensor data fusion. In: *Proceedings - IEEE International Symposium on Circuits and Systems*. [S.l.: s.n.], 1998. v. 6, p. 537–540. ISSN 02714310. Citado na página 43.

MACMANUS, R. *Trackers*. [S.l.]: David Bateman, 2015. ISBN 9781775481294. Citado 2 vezes nas páginas 15 e 36.

MAKNI, A.; LEFEBVRE, G. Attitude Estimation for Posture Detection in eHealth Services. In: *2018 IEEE 31st International Symposium on Computer-Based Medical Systems (CBMS)*. [S.l.]: IEEE, 2018. p. 310–315. ISBN 978-1-5386-6060-7. Citado 2 vezes nas páginas 10 e 42.

MARTIN, R. C. Design Principles and Design Patterns. *objectmentor*, 2000. Citado 3 vezes nas páginas 19, 20 e 25.

MARTIN, R. C. *Agile software development: principles, patterns, and practices*. [S.l.]: Prentice Hall PTR, 2003. Citado 2 vezes nas páginas 26 e 27.

MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. [S.l.]: Pearson, 2017. Citado 22 vezes nas páginas 9, 17, 24, 26, 27, 28, 29, 30, 31, 32, 33, 35, 47, 48, 54, 65, 66, 67, 68, 69, 70 e 90.

Martínez Z, J. C. et al. Utilización de Arquitecturas Limpias para Trabajo con Buenas Prácticas en la Construcción de Aplicaciones Java. *Revista Innovación Digital y Desarrollo Sostenible - IDS*, Institucion Universitaria Digital de Antioquia, v. 1, n. 2, p. 133 – 140, feb 2021. ISSN 2711-3760. Citado na página 47.

MARTINI, A.; BOSCH, J. The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles. In: *Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015*. [S.l.]: Institute of Electrical and Electronics Engineers Inc., 2015. p. 1–10. ISBN 9781479919222. Citado na página 28.

MENDES JR., J. J. A. et al. Sensor fusion and smart sensor in sports and biomedical applications. *Sensors*, v. 16, n. 10, 2016. ISSN 1424-8220. Citado na página 44.

MILANO, P. D. *Design and simulation of capacitive accelerometers*. 2018. Disponível em: <https://www.mems.polimi.it/index.php/accelerometers>. Acesso em: 6 jan. 2022. Citado 2 vezes nas páginas 9 e 38.

NEDELKOVSKI, D. *What is MEMS? Accelerometer, Gyroscope & Magnetometer with Arduino*. HowToMechatronics, 2016. Disponível em: <https://howtomechatronics.com/>. Acesso em: 4 nov. 2020. Citado 2 vezes nas páginas 9 e 37.

NEDELKOVSKI, D. *What is Hall Effect and How Hall Effect Sensors Work*. HowToMechatronics, 2017. Disponível em: <https://howtomechatronics.com/>. Acesso em: 8 nov. 2020. Citado 2 vezes nas páginas 9 e 41.

OLSEN, H. B.; BEKEY, G. A. Identification of Robot Dynamics. *IFAC Proceedings Volumes*, Elsevier, v. 19, n. 14, p. 1004–1010, dec 1986. ISSN 1474-6670. Citado na página 15.

OPENSENSE. OpenSim Project, 2021. Disponível em: <https://simtk.org/projects/opensense>. Acesso em: 7 out. 2021. Citado na página 47.

PADHYE, R.; MANI, S.; SINHA, V. S. A study of external community contribution to Open-source projects on GitHub. *11th Working Conference on Mining Software Repositories, MSR 2014 - Proceedings*, Association for Computing Machinery, p. 332–335, may 2014. Citado na página 87.

PALERMO, J. *The Onion Architecture: part 1*. 2008. <https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/>. Acesso em 2021/06/10. Citado na página 28.

PATEL, S. et al. A review of wearable sensors and systems with application in rehabilitation. v. 9, n. 1, 2012. ISSN 17430003. Citado 2 vezes nas páginas 37 e 44.

PRADHAN, P.; DWIVEDI, A. K.; RATH, S. K. Impact of Design Patterns on Quantitative Assessment of Quality Parameters. *Proceedings - 2015 2nd IEEE International Conference on Advances in Computing and Communication Engineering, ICACCE 2015*, Institute of Electrical and Electronics Engineers Inc., p. 577–582, oct 2015. Citado na página 19.

QIU, S. et al. Body Sensor Network-Based Robust Gait Analysis: Toward Clinical and at Home Use. *IEEE Sensors Journal*, v. 19, n. 19, p. 8393–8401, oct 2019. ISSN 15581748. Citado na página 91.

ROSA, J.; SILVA, H.; MATIAS, R. A web-based framework using a Model-View-Controller architecture for human motion analysis. In: . [S.l.]: Institute of Electrical and Electronics Engineers (IEEE), 2015. p. 1–2. Citado na página 46.

SHANDILYA, S.; MEENA, M. K.; KUMAR, N. A low cost wireless sensor development for assessing ground reaction force in gait analysis. In: *12th IEEE International Conference Electronics, Energy, Environment, Communication, Computer, Control: (E3-C3), INDICON 2015*. [S.l.]: IEEE, 2016. p. 1–3. ISBN 9781467373999. Citado 2 vezes nas páginas 15 e 16.

SIIRTOLA, P. et al. Efficient accelerometer-based swimming exercise tracking. In: *IEEE SSCI 2011: Symposium Series on Computational Intelligence - CIDM 2011: 2011 IEEE Symposium on Computational Intelligence and Data Mining*. [S.l.]: IEEE, 2011. p. 156–161. ISBN 9781424499274. Citado na página 44.

SREEKUMAR, R. S.; SIVABALAN, R. V. A survival study of object oriented principles on software project development. *Global Conference on Communication Technologies, GCCT 2015*, Institute of Electrical and Electronics Engineers Inc., p. 307–310, nov 2015. Citado na página 19.

STANEV, D. et al. Real-Time Musculoskeletal Kinematics and Dynamics Analysis Using Marker- and IMU-Based Solutions in Rehabilitation. *Sensors 2021, Vol. 21, Page 1804*, Multidisciplinary Digital Publishing Institute, v. 21, n. 5, p. 1804, mar 2021. Citado na página 46.

STEINMACHER, I. et al. Social barriers faced by newcomers placing their first contribution in open source software projects. *CSCW 2015 - Proceedings of the 2015 ACM International Conference on Computer-Supported Cooperative Work and Social Computing*, Association for Computing Machinery, Inc, p. 1379–1392, feb 2015. Citado na página 87.

SURESH, P. et al. A state of the art review on the Internet of Things (IoT) history, technology and fields of deployment. In: *2014 International Conference on Science Engineering and Management Research, ICSEMR 2014*. [S.l.]: IEEE, 2014. p. 1–8. ISBN 9781479976133. Citado 2 vezes nas páginas 15 e 36.

TEIGLAND, R. et al. Balancing on a tightrope: Managing the boundaries of a firm-sponsored OSS community and its impact on innovation and absorptive capacity. *Information and Organization*, Pergamon, v. 24, n. 1, p. 25–47, jan 2014. ISSN 1471-7727. Citado na página 87.

VAJS, I. A. et al. Open-source application for real-time gait analysis using inertial sensors. *2020 28th Telecommunications Forum, TELFOR 2020 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., nov 2020. Citado 2 vezes nas páginas 16 e 46.

VIR, P.; MING, S.; YONG, F. An Empirical Investigation of Code Contribution , Communication Participation , and Release Strategy in Open Source Software Development : A Conditional Hazard Model Approach. 2007. Citado na página 87.

WAGNER, J.; TRIERENBERG, A. The origin of the gyroscope: The machine of bohnenberger. *Bulletin of the Scientific Instrument Society*, v. 107, p. 10–17, 12 2010. Citado na página 15.

WANG, J. S. et al. Walking pattern classification and walking distance estimation algorithms using gait phase information. *IEEE Transactions on Biomedical Engineering*, v. 59, n. 10, p. 2884–2892, 2012. ISSN 00189294. Citado na página 16.

WATSON, J. MEMS Gyroscope Provides Precision Inertial Sensing in Harsh, High Temperature Environments. *Analog Devices*, p. 1 – 4, 2016. Citado 4 vezes nas páginas 9, 39, 40 e 45.

WEI, Y.-g. et al. Research and application of access control technique in 3d virtual reality system opensim. In: *2013 Sixth International Symposium on Computational Intelligence and Design*. [S.l.: s.n.], 2013. v. 2, p. 65–68. Citado na página 47.

WILLEMSEN, A. T. M.; BLOEMHOF, F.; BOOM, H. B. Automatic Stance-Swing Phase Detection from Accelerometer Data for Peroneal Nerve Stimulation. *IEEE Transactions on Biomedical Engineering*, v. 37, n. 12, p. 1201–1208, 1990. ISSN 15582531. Citado na página 15.

XU, D. et al. A Wearable Ultra-low-cost Gait Analysis System Based on Foot Pressure Detection. In: *2018 IEEE International Conference on Cyborg and Bionic Systems, CBS 2018*. [S.l.]: IEEE, 2019. p. 80–83. ISBN 9781538673553. Citado 2 vezes nas páginas 15 e 16.

YONG, M. S. et al. The diversity-innovation paradox in open-source software. *Proceedings - 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, MSR 2021*, Institute of Electrical and Electronics Engineers Inc., p. 627–629, may 2021. Citado na página 87.

YU, J. et al. Human gait analysis based on opensim. In: *2020 International Conference on Advanced Mechatronic Systems (ICAMechS)*. [S.l.: s.n.], 2020. p. 278–281. Citado na página 47.

ZAMAN, R. et al. Hybrid predictive model for lifting by integrating skeletal motion prediction with an opensim musculoskeletal model. *IEEE Transactions on Biomedical Engineering*, p. 1–1, 2021. Citado na página 47.

ZEBIN, T.; SCULLY, P.; OZANYAN, K. B. Inertial sensing for gait analysis and the scope for sensor fusion. In: *2015 IEEE SENSORS - Proceedings*. [S.l.]: IEEE, 2015. p. 1–4. ISBN 9781479982028. Citado 2 vezes nas páginas 16 e 41.

ZHAO, H. et al. IMU-based gait analysis for rehabilitation assessment of patients with gait disorders. In: *2017 4th International Conference on Systems and Informatics, ICSAI 2017*. [S.l.]: IEEE, 2017. v. 2018-January, p. 622–626. ISBN 9781538611074. Citado 2 vezes nas páginas 37 e 40.