UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

CARLOS SOLON SOARES GUIMARÃES JUNIOR

PROPOSTA DE UM *FRAMEWORK* BASEADO EM ARQUITETURA ORIENTADA A SERVIÇOS PARA A ROBÓTICA

Porto Alegre
2015

CARLOS SOLON SOARES GUIMARÃES JUNIOR

PROPOSTA DE UM *FRAMEWORK* BASEADO EM ARQUITETURA ORIENTADA A SERVIÇOS PARA A ROBÓTICA

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Controle e Automação

ORIENTADOR: Prof. Dr. Renato Ventura Bayan Henriques

CO-ORIENTADOR: Prof. Dr. Carlos Eduardo Pereira

Porto Alegre
2015

CARLOS SOLON SOARES GUIMARÃES JUNIOR

PROPOSTA DE UM *FRAMEWORK* BASEADO EM ARQUITETURA ORIENTADA A SERVIÇOS PARA A ROBÓTICA

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Renato Ventura Bayan Henriques, UFRGS
Doutor pela Universidade Federal de Minas Gerais – Belo Horizonte, Brasil

Banca Examinadora:

Prof. Dr. Luís Fernando Alves Pereira, UFRGS
Doutor pelo Instituto Tecnológico de Aeronáutica – São José dos Campos, Brasil

Prof. Dr. Rodrigo Da Silva Guerra, UFSM
Doutor pela Universidade de Osaka - Osaka, Japão

Prof. Dr. Jose Barata Oliveira, UNINOVA
Doutor pela Universidade Nova de Lisboa - Lisboa, Portugal

Coordenador do PPGEE: _____
Prof. Dr. Alexandre Sanfelice Bazanella

Porto Alegre, outubro de 2015.

DEDICATÓRIA

Dedico este trabalho primeiramente a Deus, por ser essencial em minha vida, autor de meu destino, ao meu pai Carlos Solon Soares Guimarães, minha mãe Maria Iolema Santiago Guimarães e aos meus irmãos. A todos os professores do curso, que foram tão importantes na minha vida acadêmica e no desenvolvimento desta dissertação, em especial ao Prof. Renato Ventura Bayan Henriques e o Prof. Carlos Eduardo Pereira, pela paciência nas orientações e incentivos que tornaram possíveis a conclusão deste trabalho.

RESUMO

Sistemas embarcados, em especial aqueles utilizados em robótica, apresentam, em sua estrutura, uma multiplicidade de dispositivos que resultam em uma arquitetura bastante heterogênea e bem distribuída. Para auxiliar na resolução dessa complexidade inerente, este trabalho resgata os conceitos de *frameworks*, buscando na sua integração e modelo conceitual, desenvolver um conjunto de ferramentas que gerencia a mediação entre sistemas embarcados e demais aplicações de *software*, fornecendo bibliotecas e componentes reutilizáveis para aplicações na robótica. Propõe-se utilizar *framework* e *middleware* de sistemas *open-source* para integração entre a plataforma de *software* e *hardware*. Um dos objetivos do projeto é criar um *framework* multi-plataforma com diferentes tipos de serviços para o de desenvolvimento de aplicações no campo da robótica. O projeto tem como estudo de caso sistemas embarcados aplicados em robótica móvel e tecnologia assistiva.

Palavras-chave: Engenharia de *Software*, Arquitetura Orientada a Serviços, *Frameworks*, *Middleware*, Sistema Embarcado, Robótica.

ABSTRACT

Embedded systems, especially those used in robotics, present in its structure, a plurality of devices that result in a very heterogeneous and well distributed architecture. To help resolve this inherent complexity, the work rescues the concepts of frameworks, seeking their integration and conceptual model, develop a set of tools that manage to mediate between embedded systems and other of software applications, providing reusable libraries and components for applications in robotics. It is proposed to use framework and middleware systems open source for integration between the platform software and hardware. One of the project objectives is to create a multi-platform framework with different types of services, considering the programming and compatibility with hardware for the development of applications in the field of robotics. The project's case study developing embedded applied in mobile and Assistive Technology robotics.

Keywords: Software Engineering, Service Oriented Architecture, Frameworks, Middleware, Embedded System, Robotics.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

BPM	Business Process Management
CC	Corrente Contínua
CCC	Central de Comutação e Controle
CIR	Centro Instantâneo de Rotação
EAI	Enterprise Application Integration)
ERB	Estação Rádio Base
ESB	Enterprise Service Bus
GSM	Global System for Mobile Communications
ICE	Internet Comunications Engine
IMEI	International Mobile Equipment Identity
JVM	Java Virtual Machine
ROS	Robot Operating System
SART	Structured Analysis Real-Time
SAW	Surgical Assistant Workstation
SIM	Subscriber Identity Module
TA	Tecnologia assistiva
TDMA	Time Division Multiple Access
UML	Unified Modeling Language
CBD	Desenvolvimento Baseado em Componentes
OO	Orientado à Objetos
SOA	Service-Oriented Architecture

LISTA DE SÍMBOLOS

θ	Ângulo que define a orientação do robô;
${}^0\dot{\xi}_c$	Pose do robô móvel;
${}^iR_j$	Matriz de rotação do sistema de coordenadas j em relação ao sistema de coordenadas i;
RC	Raio de curvatura instantâneo da trajetória do robô;
v_i	Velocidades das rodas, onde i representa as velocidades tangenciais da roda esquerda e à direita;
ω_i	Velocidades angulares, onde i representa as velocidades a roda esquerda e à direita;
y	Saída controlada;
y_R	Sinal de referência;
x_0	Estado inicial do sistema;
t_0	Instante em que o controle se inicia;
r	Raio da roda;
V	Velocidade linear;
R	Distância radial do robô a partir da posição inicial;
l	Distância entre as rodas;
n	Eixo da roda de redução;
C_e	Resolução do encoder;
k_i	Constantes do modelo;

SUMÁRIO

1 INTRODUÇÃO

O ramo da robótica costuma atrair a atenção de leigos e profissionais, explorando um campo onde a imaginação trabalha ao máximo. A robótica de serviço em específico tem levado esses sistemas para mais perto da população e trazendo desafios aos desenvolvedores destas tecnologias. Para construir uma máquina autônoma há uma série de dificuldades. Para mover entre dois pontos é preciso o controle de motores, mapeamento e sensoriamento da região, estudo da cinemática do robô, planejamento da trajetória e monitoramento do movimento. Algumas plataformas de robótica, possuem manuais e um *firmware* para facilitar a programação. Porém cada plataforma funciona de forma diferente e seus códigos seguem diferentes práticas de programação. Atualmente existe muito projetos abertos e de fácil utilização para quem busca desenvolver na área, tendo-se muitas vezes adaptar e implementar tudo desde as funções mais básicas. Parte disso acontece pelas diferenças físicas entre as máquinas e devido a se desenvolver projetos não pensando em reuso de *software* e em boas práticas de programação. As soluções existentes funcionam apenas na plataforma no qual foram desenvolvidas e com interfaces diferentes, muitas vezes impedindo que o desenvolvedor reutilize seu código em outra plataforma. Como são poucas as bibliotecas e ferramentas projetadas para funcionarem em várias plataformas, cada sistema fica acoplado às bibliotecas de suas respectivas plataformas. Este problema diminui a reusabilidade e quase anula a chance de portabilidade. Para desenvolvermos sistemas com produtividade e qualidade diversos aspectos do processo de desenvolvimento devem ser observados. No que concerne ao código desenvolvido, dois itens merecem atenção especial quando desejamos obter uma arquitetura sustentável: a coesão e o acoplamento. O acoplamento refere-se ao quanto uma unidade funcional depende de outra para funcionar. Uma unidade funcional pode ser um método, função ou mesmo uma classe. Quanto maior a dependência entre as unidades funcionais, mais fortemente acopladas elas estão. Uma das formas de se medir o acoplamento de um método, por exemplo, é a quantidade de parâmetros de entrada e suas respectivas complexidades. Quanto mais parâmetros e mais complexos eles forem, maior o acoplamento do método. A coesão está ligada à responsabilidade única da unidade funcional. Demonstra coerência e unidade conceitual no relacionamento com os outros componentes da unidade funcional. Ou seja, um método coeso realiza uma única função conceitual, servindo a apenas um propósito específico (ERL, 2007). Dadas as definições acima, podemos concluir que o desejável ao se desenvolver um módulo, método ou função é que o mesmo seja altamente coeso e com o mais baixo acoplamento possível. Métodos com muitas linhas de códigos são possíveis candidatos a *refactoring*, dada a possibilidade de baixa coesão. O mesmo pode ser aplicado para métodos com muitos parâmetros (RINCON, 2014).

Escrever um código reutilizável e extensível não é algo trivial. Para garantir que o mesmo código possa ser replicado em diferentes contextos com nenhuma ou mínima de

alteração é preciso de uma arquitetura bem projetada e uma correta modularização das tarefas do sistema. Uma vez que é conhecido o escopo com que se irá trabalhar, um planejamento da melhor forma de estruturá-lo é necessário para garantir que todas as funcionalidades sejam separadas e suas comunicações o menos acopladas possível. Isso leva ao estudo de padrões arquiteturais, encapsulamento e padrões de projeto, que nos ajudam a realizar essa tarefa.

Uma linha de pesquisa em constante crescimento na robótica são os *frameworks*. *Frameworks* são estruturas de classes que constituem implementações incompletas que, estendidas, permitem produzir diferentes artefatos de *software*. Uma das vantagens desta abordagem é a promoção de reuso de código e projeto, que pode diminuir o tempo e o esforço exigidos na produção de *software*. A abordagem de *frameworks* pode se valer de padrões para a obtenção de estruturas de classes e *Component-Based Development* ou simplesmente Desenvolvimento Baseado em Componentes (DBC) bem organizadas e mais aptas a modificações e extensões. O desenvolvimento orientado a componentes pretende organizar a estrutura de um *software* como uma interligação de artefatos de *software* independentes, os componentes. O reuso de componentes previamente desenvolvidos, permite a redução de tempo e esforço para a obtenção de um *framework* para robótica baseado em arquiteturas orientadas a serviços (*Service-Oriented Architecture - SOA)*. Uma arquitetura SOA tem como seu componente fundamental o conceito de serviços. O conceito de serviço é definido de diversas formas, como mostram Silva (2000a) e Maxwell (2011).

Portanto, o conceito do que é uma arquitetura orientada a serviços ainda não é consensual, uma vez que serviço não é um termo bem definido e se confunde com os conceitos de componente e de contrato. Através do trabalho de Silva (2000a) e Fugita (1993) foi possível identificar algumas características relevantes que todas aplicações e *frameworks* que se dizem orientados a objetos (OO), componentes e serviços, possuem. O uso do padrão SOA está evoluindo e está cada vez mais presente em aplicações nos mais diversos segmentos, sejam eles em nível de dispositivos, na implementação de camadas de negócios ou mesmo no setor industrial, como apresentado em Giacomolli (2014). É um conceito de arquitetura que suporta acoplamento mínimo entre componentes, possibilitando ganhos em flexibilidade e interoperabilidade. Por conseguinte, qualquer tipo de aplicação pode ser representada como um conjunto complexo de serviços. Com a utilização de SOA, um recurso ou componente é identificado como um serviço. As funcionalidades agregadas a um serviço são publicadas e disponibilizadas através de uma interface padrão ou classe abstrata, o que possibilita a troca de informações ou requisição da execução de alguma tarefa entre os componentes. A reutilização de artefatos de *software* em larga escala é um dos argumentos a favor da abordagem de orientação a serviços. A expressão artefato de *software* é usada aqui de forma genérica, não se referindo necessariamente a código (podendo abranger os produtos da análise e do projeto) (SILVA, 2000).

Atualmente, existe uma infra-estrutura tecnológica pronta, que suporta CBD e SOA sem muito esforço, mas não garante, apenas permite a aplicação das abordagens citadas. Uma comparação de CBD versus SOA é muito abrangente e, portanto, envolve questões de várias dimensões. Sendo assim, este trabalho focará nos serviços que refletem os processos do projeto. Isto porque, é apenas nesse âmbito que o SOA pode cumprir sua promessa de alinhar serviços com negócio. A relevância dessa pesquisa está na fusão de diferentes conhecimentos teóricos das distintas áreas das engenharia e ciências que, integradas, proporcionaram a implementação de um *framework* aplicado a robótica.

1.1 Objetivo

Observa-se em comum nas abordagens de *frameworks* OO e CBD a possibilidade de reutilização de artefatos de *software* de alta granularidade, e a caracterização de reuso de projeto e código. A adoção destas abordagens no desenvolvimento de *software* - em conjunto ou isoladamente - pode diminuir significativamente o tempo para o desenvolvimento de novas aplicações orientadas a serviços, bem como a quantidade de código a desenvolver, na medida em que parte das responsabilidades das novas aplicações são atribuídas a artefatos de *software* reutilizados. O presente trabalho se preocupa com a integração das abordagens de *middleware*, independência de plataforma, encapsulamento, reusabilidade, escalabilidade, simplicidade de integração, simplicidade de estender as funcionalidades existentes, para desenvolver um conjunto de bibliotecas e componentes, que gerenciam a mediação entre as demais aplicações do sistema, fornecendo bibliotecas reutilizáveis e ferramentas, para o desenvolvimento de um *framework* baseado em SOA para robótica, que têm em comum a reutilização, seja de projeto ou de código, isto é, o aproveitamento de experiências de desenvolvimentos de *hardwares* e *softwares open-source*, para criar um ambiente que disponibilize serviços e integração de diferentes aplicações. A integração se concentra principalmente em pacotes de classes e componentes, que permitam o aplicativo do *software* simular, controlar e interagir com os diferentes sistemas integrados. Como objetivos do trabalho, estão a integração e especificação do modelo conceitual da arquitetura orientada a serviço do *framework*, implementação de uma coleção de pacotes de classes e componentes, para auxiliar na percepção, controle e desenvolvimento de dispositivos, fornecendo serviços para projetos de robótica. Para validação do artefato de *software* pretendido, dois estudos de casos foram aplicados no *framework*, um baseado em robótica móvel (EDUBOT, 2014); (GUIMARÃES; TAMAYO; HENRIQUES, 2014) e outro em tecnologia assistiva (GUIMARÃES; HENRIQUES; PEREIRA, 2012); (GUIMARÃES; HENRIQUES; PEREIRA, 2013); (GUIMARÃES; PEREIRA; HENRIQUES, 2014).

1.2 Organização do Trabalho

No Capítulo 2 são apresentados os fundamentos teóricos, que descrevem resumidamente alguns conceitos sobre Engenharia de *Software* e os diferentes paradigmas de *frameworks*. O Capítulo 3 apresenta alguns *frameworks Open-source* para robótica. No Capítulo 4 são apresentadas as análises e projetos dos estudos de caso do trabalho. O Capítulo 5 aborda a metodologia de desenvolvimento utilizada para o *framework*. No Capítulo 6 é apresentada a proposta do *framework* pretendido. O Capítulo 7 apresenta a implementação e resultados e por fim, o Capítulo 8 são feitas as considerações finais sobre o trabalho desenvolvido e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Um dos principais objetivos da Engenharia de *Software* é o reuso. Através da reutilização de *software* obtém-se o aumento da qualidade e redução do esforço de desenvolvimento. A orientação a objetos fornece funcionalidades para que classes possam ser reutilizadas, bem como métodos por meio de seus mecanismos de herança e polimorfismo. Componentes de *software* definem unidades reutilizáveis que oferecem serviços através de interfaces bem definidas. Além destas formas de reutilização, a tecnologia de *frameworks* possibilita que uma família de produtos seja gerada a partir de uma única estrutura que captura os conceitos mais gerais da família de aplicações para criar um *framework* orientado a serviços (MAXWELL, 2014, 2013).

Este capítulo apresenta os principais conceitos sobre *frameworks* encontrados na literatura. Primeiramente, são revistas algumas definições sobre *frameworks*. Logo após, são apresentados os conceitos da Engenharia de *software* de serviços e sua relação com os paradigmas OO e CBD. E por último são apresentados alguns dos conceitos referentes aos estudos de casos do projeto.

2.1 Sistemas Embarcados

Existem muitas definições de sistemas embarcados, mas todas estas definições podem ser combinadas em um conceito simples. Este sistema computacional é normalmente menos poderoso que os sistemas computacionais de propósito geral, embora em alguns casos existam sistemas embarcados que sejam complexos, desempenhando várias funções diferentes. Quase todos dispositivos eletrônicos modernos utilizam processadores para ajudar a controlar fábricas, gerenciar sistemas e habilitar a comunicação entre pessoas e produtos. Normalmente em sistemas embarcados é utilizado um processador de baixo consumo de potência com uma quantidade limitada de memória. Alguns destes sistemas embarcados utilizam sistemas operacionais otimizados, que possuem uma capacidade limitada de funções de operação. Contudo, a escolha dos componentes do sistema sempre deve ser baseada nas características da aplicação para a qual o sistema embarcado está sendo proposto. Muitos sistemas embarcados são concebidos partindo do pressuposto que devem ser utilizados por um longo período de tempo sem necessitarem manutenção. Estes tipos de componentes sofrem desgaste natural com o tempo e periodicamente necessitam ser trocados e/ou reparados.

Um projeto de *Software* Embarcado inicia-se primeiramente estudando todas as possibilidades e desafios que serão enfrentados durante tal projeto. Basicamente seria entender as suas características, que vão desde a quantidade da memória do *software* (o que faz toda a diferença em um projeto) até os conceitos de programação em tempo real. Outra característica importante é o custo total desse desenvolvimento, pois, como a cada

dia os *softwares* se tornam mais complexos, aumenta-se, assim, os seus custos, sendo importante, então, analisá-los. Com o aumento do custo, aumenta-se também a pressão para que o projeto seja concluído o mais rápido possível, podendo haver alguns problemas no final do projeto. Falar de Sistemas Embarcados abrange uma área grande de diferentes conceitos, envolvendo tipos de *hardwares* e *softwares*, sendo a base física da tecnologia o *hardware* e a lógica o *software*. Os principais componentes de *hardware* de sistema embarcado são os processadores e *chips*, algumas tecnologias de conectividade, como Acesso Múltiplo por Divisão de Código (CDMA), Acesso Múltiplo por Divisão de Tempo (TDMA) e Sistema Global para Comunicações Móveis (GSM), *bluetooth* e uma das mais usadas, a *wireless*.

O *firmware* é um *software* interpolado num dispositivo de *hardware* que permite a leitura e execução de *software*, porém não permite modificação. O termo *firmware* foi originado para indicar um substituinte de *hardware* em microcontroladores. Em outras palavras, o *firmware* é um controlador de entrada e saída de baixo nível que gerencia dispositivos de *hardware*. No computador, ele permite a comunicação entre *software* e *hardware*. A linguagem de programação dos *firmwares* é, primordialmente, a linguagem de máquina, cada arquitetura de computador tem a sua própria linguagem de máquina e, portanto, a sua própria linguagem de montagem (*Assembly*). A linguagem de máquina, que é um mero padrão de bits, torna-se legível pela substituição dos valores em bruto por símbolos chamados mnemónicos. Essas linguagens de montagem diferem no número e tipo de operações que suportam. Também têm diferentes tamanhos e números de registradores, e diferentes representações dos tipos de dados armazenados. Enquanto todos os computadores de utilização genérica são capazes de desempenhar essencialmente as mesmas funções, o modo como o fazem é diferente. Além disso, podem existir conjuntos múltiplos de *mnemónicas*, ou sintaxes de linguagem de montagem, para um único conjunto de instruções. Nestes casos, o conjunto mais popular é aquele que é utilizado pelo fabricante na sua documentação. Porém, hoje alguns microcontroladores interpretam, também, funções e classes de linguagens como C/C++ e Java, linguagens de programação de alto nível para implementação de sistemas, permitindo uma descrição com um alto nível de abstração tornando os programas mais fáceis de serem lidos e escritos.

Ser desenvolvedor de um projeto de *software* embarcado não é uma tarefa trivial, pois se deve ter um cuidado muito grande no desenvolvimento do projeto em relação às demandas de desempenho, espaço e potência consumida. Sistemas Embarcados são sistemas computacionais especialistas. Estes sistemas são constituídos por um conjunto *hardware*, *software* e periféricos, sendo responsáveis por uma função específica ou um conjunto restrito de funções específicas (WEHRMEISTER, 2005).

2.2 Engenharia de *Software*

Uma primeira definição de engenharia de *software* foi proposta por *Fritz Bauer* na primeira grande conferência dedicada ao assunto: "O estabelecimento e uso de sólidos princípios de engenharia para que se possa obter economicamente um *software* que funcione eficientemente com máquinas reais". A Engenharia de *Software* atua na aplicação de abordagens sistemáticas ao desenvolvimento e manutenção de *software*. Os objetivos principais da Engenharia de *Software* são a melhora da qualidade do *software* e o aumento da produtividade da atividade de desenvolvimento de *software*.

Na literatura, (PRESSMAN, 2010; SOMMERVILLE, 2011) pode-se encontrar diversas definições da Engenharia de *Software*:

- "O estabelecimento e uso de sólidos princípios de engenharia para que se possa obter economicamente um *software* que seja confiável e que funcione eficientemente em máquinas reais".

- "Engenharia de *Software* é um ramo da engenharia cujo foco é o desenvolvimento de *softwares* de alta qualidade e dentro de custos adequados".

- "Aplicação de uma abordagem sistemática, disciplinada e qualificável, para o desenvolvimento, operação e manutenção do *software*".

A engenharia de *software* é uma derivação da engenharia de sistemas e de *hardware*. Ela abrange um conjunto de três elementos fundamentais - métodos, ferramentas e procedimentos - que possibilita ao gerente o controle do processo de desenvolvimento do *software* e oferece ao profissional uma base para a construção produtiva de *software* de alta qualidade. Os métodos de engenharia de *software* proporcionam os detalhes de "como fazer"para construir o *software*. Os métodos envolvem um amplo conjunto de tarefas que incluem: planejamento, estimativa de projeto, análise de requisitos de *software* e de sistemas, projeto da estrutura de dados, arquitetura de programa e algoritmo de processamento, codificação, teste e manutenção. Os métodos da engenharia de *software* muitas vezes introduzem uma notação gráfica ou orientada a linguagem especial e introduzem um conjunto de critérios para a qualidade do *software*. Um dos principais objetivos da Engenharia de *Software* é o reuso. Através da reutilização de *software* obtém-se o aumento da qualidade e redução do esforço de desenvolvimento.

De modo mais objetivo, pode-se dizer que a Engenharia de *Software* busca prover a tecnologia necessária para produzir *software* de alta qualidade a um baixo custo. Os dois fatores motivadores são essencialmente a qualidade e o custo. A qualidade de um produto de *software* é um parâmetro cuja quantificação não é trivial, apesar dos esforços desenvolvidos nesta direção. Por outro lado, o fator custo pode ser facilmente quantificado desde que os procedimentos de contabilidade tenham sido corretamente efetuados (MAXWELL, 2014, 2013).

2.3 Classificação e Definição de *Frameworks*

Frameworks podem ser classificados de diversas formas. Inicialmente, são classificados em dois grupos principais: *frameworks* orientados a objetos e *frameworks* de componentes (SILVA, 2000) (MAXWELL, 2014, 2013). O primeiro define uma estrutura para o desenvolvimento de aplicações orientados a objetos (OO) enquanto que o segundo define uma infra-estrutura de execução onde componentes (DBC) podem ser conectados. *Frameworks* OO e CBD podem ainda ser classificados quanto ao seu uso como *frameworks* caixa branca, que exige um bom conhecimento da estrutura interna do *framework* por parte do usuário do *framework*, caixa preta, que é menos flexível que o *framework* caixa branca, mas possibilita o seu uso sem que o desenvolvedor precise conhecer a estrutura interna do *framework*, e caixa cinza, que tenta combinar as vantagens dos *frameworks* caixa branca e preta.

Além disso, um *framework* agiliza o processo de desenvolvimento de *software* que lida com determinado tipo de problema, provendo recursos que fornecem flexibilidade para os desenvolvedores adequarem-no às suas necessidades, reusando a solução da parte comum dos problemas. Também pode ser definido como um projeto reusável de todo ou de parte de um sistema, fornecendo um conjunto de classes abstratas e a forma com

que suas sub-classes interagem. Um *framework* é um esqueleto de um sistema, que é instanciado e especializado para gerar uma família de aplicações.

A literatura é repleta de definições de *frameworks*. A seguir, as principais são descritas segundo os trabalhos seguidos:

- "Um *framework* é um conjunto de classes que constitui um projeto abstrato para a solução de uma família de problemas."
- "Um *framework* é uma arquitetura desenvolvida com o objetivo de atingir a máxima reutilização, representada como um conjunto de classes abstratas e concretas, com grande potencial de especialização."
- "Um *framework* é um conjunto de objetos que colaboram com o objetivo de atender a um conjunto de responsabilidades para uma aplicação específica ou um domínio de aplicação."
- "Um *framework* é definido como um *software* parcialmente completo projetado para ser instanciado. O *framework* define uma arquitetura para uma família de subsistemas e oferece os construtores básicos para criá los."

Apesar de diferentes, as definições encontradas na literatura não são contraditórias. A definição adotada nesta dissertação é a de (SILVA, 2000; FUGITA, 2009) e (MAXWELL, 2014, 2013).

2.3.1 *Frameworks* Orientado a Objetos

A abordagem de *framework* Orientado a Objetos (OO) utiliza o paradigma de orientação a objetos para produzir uma descrição de um domínio para ser reutilizada. Um *framework* OO é uma estrutura de classes inter-relacionadas, que corresponde a uma implementação incompleta para um conjunto de aplicações de um domínio. Esta estrutura de classes deve ser adaptada para a geração de aplicações específicas, gerando um esqueleto de implementação de uma aplicação ou de um subsistema de aplicação, em um domínio de problema particular. É composto de classes abstratas e concretas e provê um modelo de interação ou colaboração entre as instâncias de classes definidas pelo *framework* OO. Um *framework* OO é utilizado através de configuração ou conexão de classes concretas e derivação de novas classes concretas a partir das classes abstratas do *framework* OO. A diferença fundamental entre *framework* OO e a reutilização de classes de uma biblioteca, é que neste caso são usados artefatos de *software* isolados, cabendo ao desenvolvedor estabelecer sua interligação, e no caso do *framework* OO, é procedida a reutilização de um conjunto de classes inter-relacionadas estabelecido no projeto do *framework* OO. As Figuras 1, 2 e 3 abaixo, ilustram esta diferença (a parte sombreada representa classes e associações que são reutilizadas).

Dois aspectos caracterizam um *framework* OO:

- **Os *frameworks* OO fornecem infraestrutura e projeto:** *frameworks* OO portam infraestrutura de projeto disponibilizada ao desenvolvedor da aplicação, que reduz a quantidade de código a ser desenvolvida, testada e depurada. As interconexões pre-estabelecidas definem a arquitetura da aplicação, liberando o desenvolvedor desta responsabilidade. O código escrito pelo desenvolvedor visa estender ou particularizar o comportamento do *framework* OO, de forma a moldá-lo a uma necessidade específica.

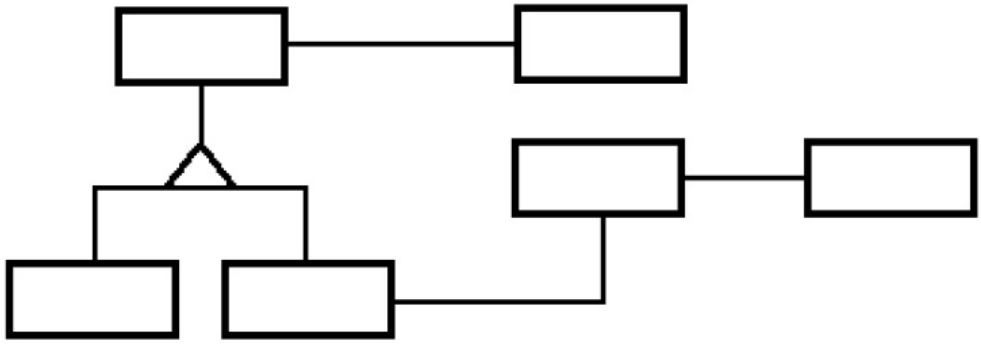


Figura 1: Aplicação desenvolvida totalmente (SILVA, 2000)

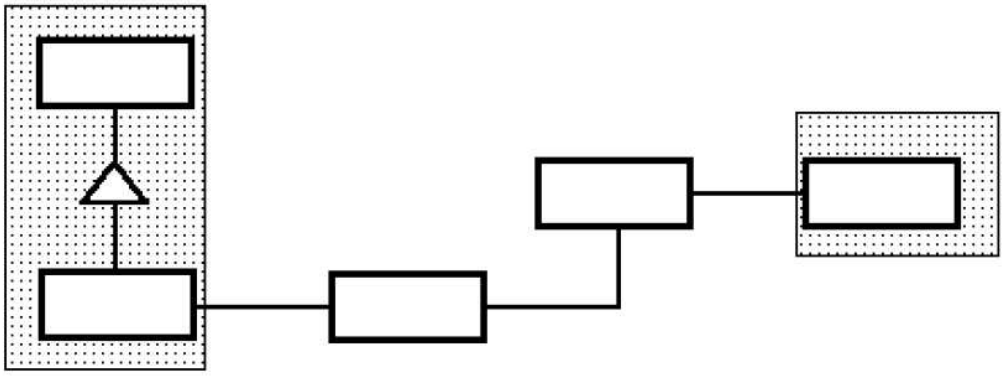


Figura 2: Aplicação desenvolvida reutilizando classes de biblioteca (SILVA, 2000)

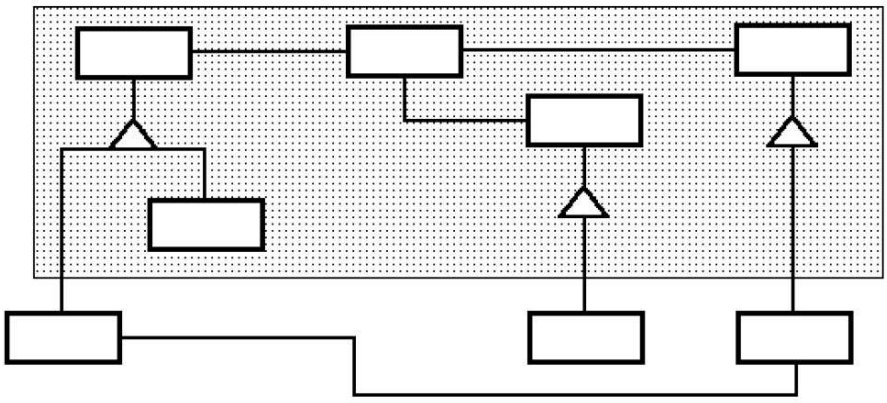


Figura 3: Aplicação desenvolvida reutilizando um *framework* OO (SILVA, 2000)

- **Os *frameworks* OO "chamam", não são "chamados":** um papel do *framework* OO é fornecer o fluxo de controle da aplicação. Assim, em tempo de execução, as instâncias das classes desenvolvidas esperam ser chamadas pelas instâncias das classes do *framework* OO.

Um *framework* OO se destina a gerar diferentes aplicações para um domínio. Precisa, portanto, conter uma descrição dos conceitos deste domínio. As classes abstratas de um *framework* OO são os repositórios dos conceitos gerais do domínio de aplicação. No contexto de um *framework* OO, um método de uma classe abstrata pode ser deixado propositalmente incompleto para que sua definição seja acabada na geração de uma aplicação. Apenas atributos a serem utilizados por todas as aplicações de um domínio são incluídos em classes abstratas.

No processo de desenvolvimento de um *framework* OO, deve-se produzir uma estrutura de classes com a capacidade de adaptar-se a um conjunto de aplicações diferentes. Para construir um *framework* OO, é fundamental que se disponha de modelagens de um conjunto significativo de aplicações do domínio. Este conjunto pode se referir a aplicações previamente desenvolvidas, ou a aplicações que se deseja produzir a partir do *framework* OO. A ótica de diferentes aplicações é o que dá ao desenvolvedor a capacidade de diferenciar conceitos gerais de conceitos específicos.

Em termos práticos, dotar um *framework* OO de generalidade, alterabilidade e extensibilidade requer uma cuidadosa identificação das partes que devem ser mantidas flexíveis e a seleção de soluções de projeto de modo a produzir uma arquitetura bem estruturada. Isto passa pela observação de princípios de projeto OO, como o uso de herança para reutilização de interfaces (ao invés do uso de herança para reutilização de código); reutilização de código através de composição de objetos; preocupação em promover polimorfismo, na definição das classes e métodos, de modo a possibilitar acoplamento dinâmico. No contexto de *frameworks* OO, o uso adequado de herança implica na concentração das generalidades do domínio em classes abstratas, no topo da hierarquia de classes. Isto promove uso adequado de herança, pois a principal finalidade destas classes abstratas é definir as interfaces a serem herdadas pelas classes concretas das aplicações.

Pode-se afirmar que o desenvolvimento de um *framework* OO é mais complexo que o desenvolvimento de aplicações específicas do mesmo domínio, devido:

- À necessidade de considerar os requisitos de um conjunto significativo de aplicações, de modo a dotar a estrutura de classes do *framework* OO de generalidade, em relação ao domínio tratado;

- À necessidade de ciclos de evolução voltados a dotar a estrutura de classes do *framework* OO de alterabilidade e extensibilidade.

A pesquisa sobre identificação de padrões fornece estruturas de projeto semiprontas que podem ser reutilizadas, contribuindo no desenvolvimento de *frameworks* OO, no sentido de produzir uma organização de classes bem estruturada. Padrões promovem o uso adequado de herança e o desenvolvimento de projetos em que o acoplamento entre classes é minimizado.

O processo de desenvolvimento de *frameworks* OO pode envolver a tarefa de combinar *frameworks* OO para uso conjunto. A Figura 4 ilustra o uso de composição de objetos para a combinação de objetos responsáveis pela lógica de controle.

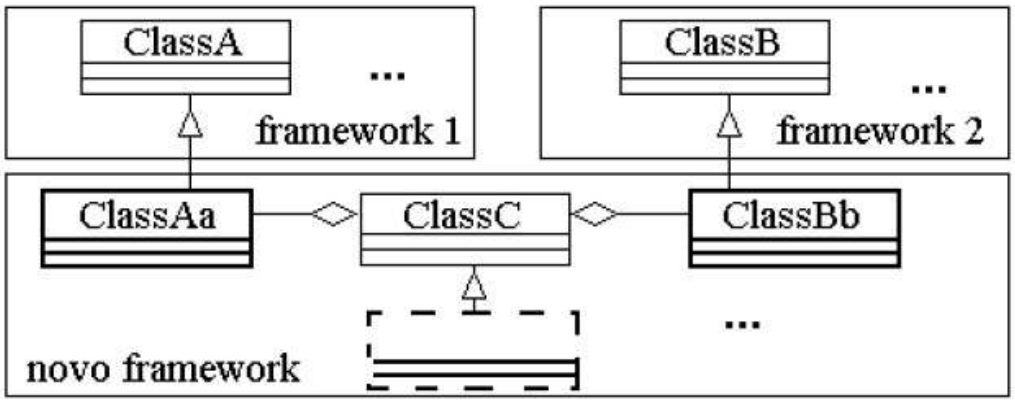


Figura 4: Combinação de *frameworks* OO (SILVA, 2000)

2.4 Engenharia de *Software* Baseado em Componentes

A Engenharia de *Software* baseado no desenvolvimento baseado em componentes ou CBD é uma abordagem que surgiu na comunidade de engenharia de *software* na última década. Destina-se a mudar a ênfase na construção do sistema de programação tradicional para compor sistemas de *software* com uma mistura de componentes padrões e componentes customizados. A utilização desta abordagem permite a reutilização de componentes de *software* em diversas aplicações, ao invés de se construir um sistema inteiramente novo, o que diminui significativamente a quantidade de código gerado. Além da diminuição do esforço empregado, outros benefícios são evidentes, pois um componente amplamente usado torna-se confiável, robusto, eficiente e bem conhecido quanto as suas funcionalidades, interfaces e limitações. Tendo-se em vista um projeto complexo que envolva diferentes funcionalidades, pode-se mais facilmente construí-lo utilizando blocos funcionais, a partir do reúso de componentes prontos. Assim, o projetista não necessita conhecer a implementação específica de cada funcionalidade, focando apenas na de seu interesse. Além disso, este sistema torna-se modular, o que ajuda nas dependências de controle e aumenta a exigibilidade do sistema para futuras alterações e manutenções. Os testes no sistema também ficam mais fáceis, uma vez que pode-se testar cada componente separadamente. Contundo, desenvolvimento de *software* baseado em componentes e a reutilização de código ainda não são práticas amplamente utilizadas na robótica.

Hoje em dia a maioria das pesquisas e desenvolvimento de *software* ainda são baseadas em arquiteturas de *software* personalizadas, construídas a partir do zero. Portanto a maioria das aplicações na robótica são sistemas desenvolvidos com uma finalidade específica, os quais acumulam grande quantidade de *software* que implementam sistemas completos. No entanto, isso não favorece o reúso de *software*, pois este torna-se específico para um determinado *hardware*, sistema operacional ou meio de comunicação, além do que toda funcionalidade e conhecimento está dentro do código, e não exposta de forma clara e organizada em uma interface (FUGITA, 2009); (MAXWELL, 2014, 2013). Afim de superar tais dificuldades, a engenharia *software* baseada em componentes possui três principais funções:

- Desenvolver *software* a partir de partes pré produzidas;
- Reutilizar tais partes em diferentes aplicações;

- Ser capaz de oferecer fácil manutenção e customização a estas partes para desenvolver novas funções e funcionalidades;

2.4.1 *Frameworks* Orientado a Componentes

Para acessar e interconectar componentes, são utilizadas suas portas. Uma porta é um meio identificável de conexão, por onde um componente oferece seus serviços ou acessa os serviços dos outros. As portas são ligadas através de conectores, implementados através de chamada de métodos, propagação de eventos, fluxo de dados, transferência de arquivos, etc. Os tipos de conectores variam para cada tecnologia e possibilitam a conexão em tempo de codificação, compilação, inicialização ou execução. A interface é o contrato de utilização do componente. Respeitando-se os contratos, pode-se alterar a implementação interna do componente ou substituí-lo por outro, sem modificar seus clientes. A interface define as maneiras de utilizar o componente, separando a especificação da implementação. Um componente apresenta múltiplas interfaces correspondendo aos conjuntos de serviços que visam diferentes necessidades dos clientes. Normalmente, o componente possui pelo menos uma interface relativa aos serviços disponibilizados (interface de negócio) e outra à conexão com a infra-estrutura de execução (interface de sistema), onde são tratados serviços técnicos, como os relacionados ao ciclo de vida, à instalação e à persistência.

A forma usual de descrever um componente consiste na descrição de sua interface. Porém, os mecanismos de descrição de interface existentes, em geral, são pobres para descrever componentes porque produzem apenas uma visão externa incapaz de descrever suas funcionalidades e como estes interagem, Silva (2000b) especifica os diagramas de componentes conforme sua abstração. Na visão do desenvolvimento orientado a componentes do autor, comunicação unidirecional é um caso particular, pois a interação de um componente com o meio externo pode ser bidirecional, ou seja, em geral o componente pode fornecer serviços mas também requerê-los. É possível que um componente precise interagir com mais de um componente simultaneamente. Neste caso a interface deve dispor de mecanismos de acesso bidirecionais e que possibilitem a conexão de um componente a mais de um componente. Cada um destes pontos de acesso é chamado de canal de comunicação. Assim, no trabalho de Silva (2000b), um componente possui uma interface, composta de um ou mais canais de comunicação, através dos quais o componente se comunica com o meio externo. A Figura 5 ilustra um artefato de *software* constituído pela interligação de um conjunto de componentes, através dos canais de comunicação de suas interfaces.

Dificilmente componentes são reusáveis tal qual foram desenvolvidos. Normalmente precisam ser adaptados para se moldarem aos requisitos do sistema a que serão acoplados. Duas abordagens têm sido usadas para adaptar um componente: alteração e empacotamento (*wrapping*). O empacotamento, ao invés de modificar o componente, cria uma visão externa diferente para ele. Outra alternativa para interconectar componentes originalmente incompatíveis consiste em criar um componente intermediário para mediar a comunicação. Este componente intermediário é genericamente chamado cola (*glue*).

O empacotamento consiste em produzir uma visão externa para um componente, isto é, uma interface, diferente de sua interface original com vista a adaptá-lo a requisitos específicos. A Figura 6 ilustra esta situação. Apresenta dois componentes, Componente1 e Componente2, com interfaces incompatíveis, isto é, não podem ser conectados para operação conjunta. Na parte inferior da figura o bloco que representa o Componente2 é inserido em uma estrutura de empacotamento com uma interface diferente da interface

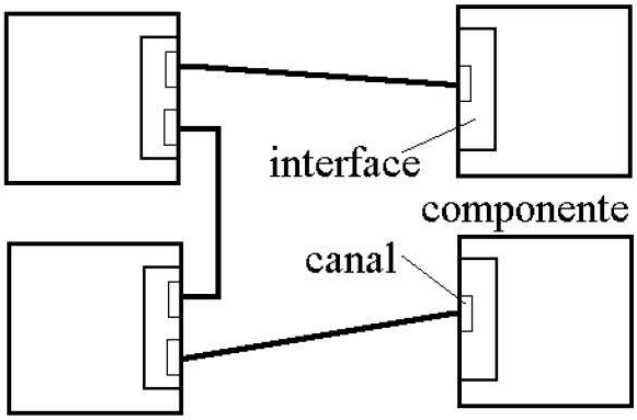


Figura 5: Interligação de componentes através de seus canais de comunicação (SILVA, 2000)

definida neste componente. O resultado é que a interface da estrutura de empacotamento é compatível com a interface do componente Componente1, possibilitando a interligação de suas interfaces. Com isto, obtém-se a situação inicialmente impossível: a operação conjunta dos componentes Componente1 e Componente2, com interfaces incompatíveis.

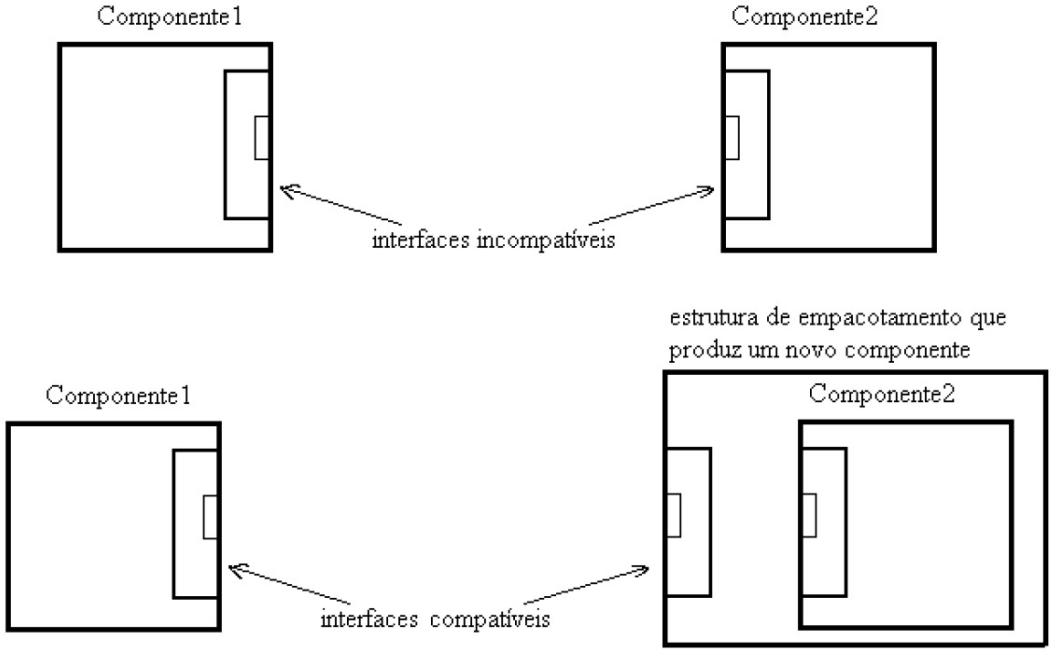


Figura 6: Adaptação de componente através de empacotamento (*wrapping*) (SILVA, 2000)

O recurso do empacotamento pode ser usado com diferentes finalidades. Uma primeira situação é que a única incompatibilidade seja a estrutura de interface, isto é, componentes comportamental e funcionalmente compatíveis com interfaces não conectáveis. Isto pode ser causado por assinaturas de métodos diferentes em um ambiente homogêneo (diferenças em nome de método, ordem de parâmetros, previsão de retorno), ou por heterogeneidade dos componentes (que pode incluir diferença de linguagem de programação, de plataforma de execução e de localização física).

A colagem de componentes (*glueing*) trata o mesmo problema do empacotamento, isto

é, viabilizar a operação conjunta de componentes originalmente incompatíveis. Como no caso anterior, esta incompatibilidade pode estar associada a sintaxe das interfaces, heterogeneidade ou necessidade de extensões ou alterações funcionais. A diferença neste caso é que o tratamento dado ao problema é a inclusão de um novo elemento, a cola (*glue*), entre os componentes incompatíveis, possibilitando sua operação conjunta. A Figura 7 ilustra a compatibilização de componentes através de colagem.

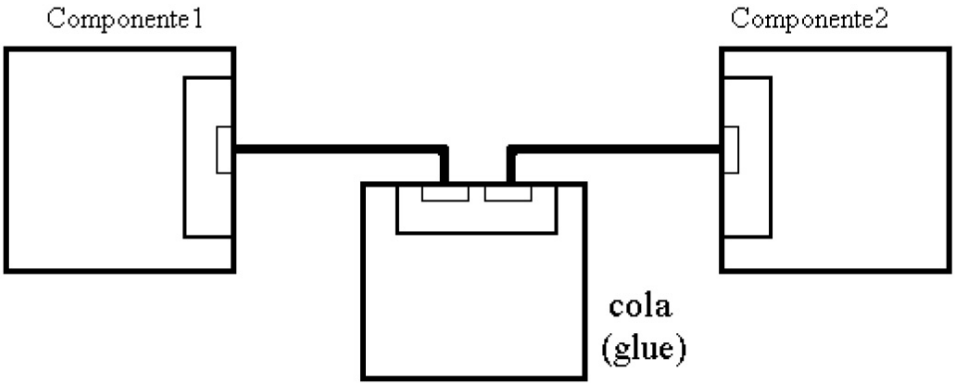


Figura 7: Adaptação de componente através de colagem (*glueing*) (SILVA, 2000)

O elemento cola nada mais é senão um terceiro componente, cuja interface possibilita sua conexão aos componentes Componente1 e Componente2 e cuja funcionalidade consiste em compatibilizar a operação conjunta destes componentes.

As interfaces são classificadas em fornecidas (*provided interfaces*) e requeridas (*required interfaces*). Um componente possui uma interface fornecida ao implementar todas as operações definidas naquela interface e uma interface requerida ao usar pelo menos uma operação definida na interface. Na UML 2.0, interfaces fornecidas são representadas por uma circunferência fechada, enquanto as interfaces requeridas são semicircunferência. Conforme ilustrado na Figura 8, componentes se conectam por meio da interface requerida de um com a interface fornecida de outro. Para conectar componentes com conectores incompatíveis, desenvolve-se um código adicional chamado de adaptador, que faz as conversões e operações necessárias para compatibilizar interfaces.

O modelo de componentes (*component model*) define vários aspectos da construção e da interação dos componentes, entre eles, a forma de implementar as interfaces e os conectores. Vários modelos apoiam-se na orientação a objetos para a implementação de interfaces e mensagens, entretanto, esta tecnologia não provê suporte à representação de interfaces requeridas e aspectos não-funcionais. O modelo de componentes define também o padrão de nomeação dos componentes, de composição, de versionamento e de empacotamento. O empacotamento possibilita que um componente seja instalado como uma unidade, contendo arquivos, módulos, código executável, código fonte, código de validação, etc.

Um exemplo de arquitetura para projetos que envolvam o reuso de componentes é *Service Oriented Architecture*, SOA , pois nela dividi-se as funcionalidades do sistema nos chamados serviços. Esses serviços representam para os requisitos o mesmo que os componentes representam para a arquitetura da aplicação. Sendo assim, pode-se dizer que um sistema que será desenvolvido utilizando uma arquitetura SOA pode ser visto como uma composição de funcionalidades e essas funcionalidades podem ser desenvolvidas baseadas em componentes. O *Gartner Group* descreve SOA da seguinte maneira: "Mais do que uma tecnologia, SOA também influencia regras e processos de negócios,

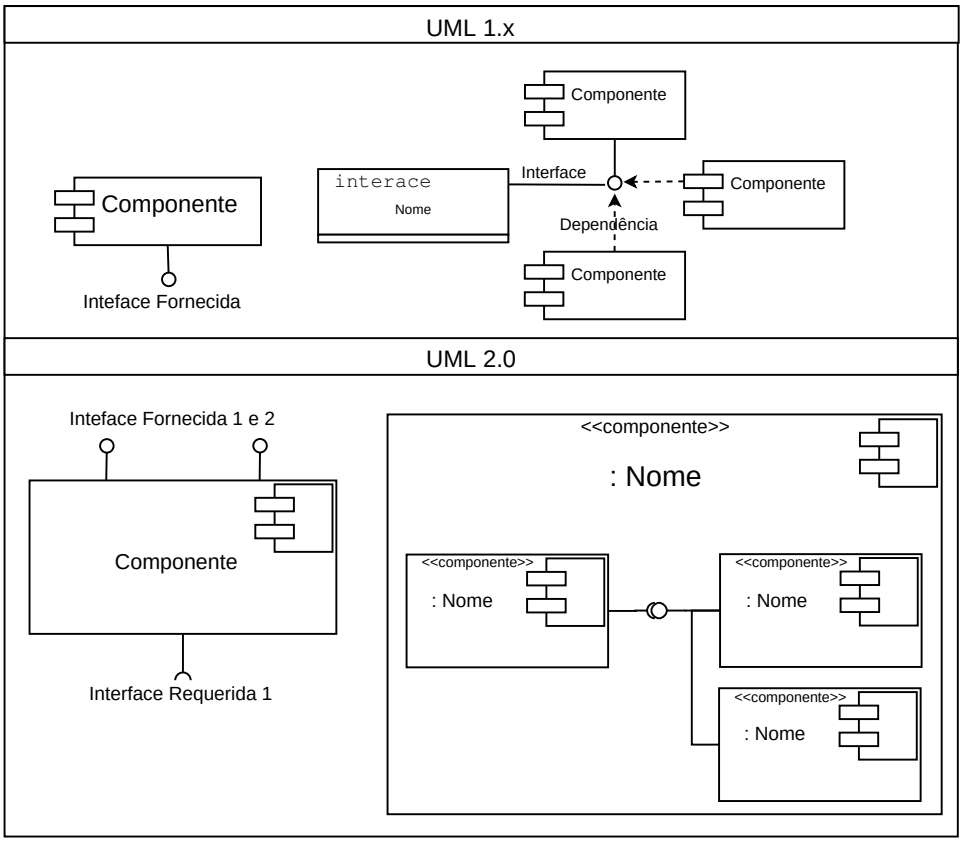


Figura 8: Padrões de diagramas de componentes segundo a UML

além de muitas vezes implicar reengenharia de *software* simultaneamente". Apesar das facilidades apresentadas em se trabalhar com SOA e componentes, não se pode restringir um ao outro. O desenvolvimento de componentes também pode ser usado quando são empregados outros estilos de arquiteturas.

2.5 *Frameworks* Caixa-branca, Caixa-preta e Caixa-cinza

Rincon (2014) descreve *frameworks* OO e componentes como podendo ser caixa-preta, caixa-branca ou caixa-cinza. Um *framework* caixa-preta só revela suas interfaces e como usá-la. Neste tipo de abordagem não há nenhum tipo de informação sobre seu comportamento ou subclasses. Essa abordagem é raramente usada para *frameworks* e mal aconselhada pelo autor, se encaixando melhor para projeto de componentes. Ainda assim, o sistema caixa-preta não é todo fechado a extensão. Novos comportamentos podem ser adicionados através de *plugins* que estendem das interfaces e unidos via composição. Porém, este tipo de abordagem é mais difícil de desenvolver (MAXWELL, 2014, 2013).

Já a abordagem caixa-branca revela sua estrutura, permite extensão de suas classes e personalização de seu comportamento. Ela é muito mais usada para *frameworks* e aconselhada por Rincon (2014). Já Fugita (1993) diz que, apesar de ser largamente utilizada, a estrutura caixa-branca exige que o programador conheça profundamente a estrutura interna do *framework* e que o uso de herança é muito mais comum em caixa-branca, enquanto no caixa-preta usa-se mais composição e delegação. Em comparação com o modelo caixa-preta, este é mais fácil de se desenvolver, porém mais difícil de estender.

O padrão caixa-cinza revela alguns detalhes de seu funcionamento e permite exten-

são e colaboração com alguns de seus pacotes, porém mantém algumas partes totalmente fechadas em seus componentes. *Frameworks* caixa-cinza tem flexibilidade suficiente e extensibilidade, e ainda tem a habilidade de esconder informações desnecessárias do desenvolvedor da aplicação (RINCON, 2014).

2.6 Engenharia de *Software* de Serviço

A discussão sobre o tema Engenharia de *Software* de Serviço foi incentivada pela repercussão a respeito da utilização da Arquitetura Orientada à Serviço, que apresenta muitas vantagens (e desafios) em sua aplicação, porém, ainda necessita de metodologia que norteie o desenvolvimento de sistemas baseados neste paradigma de arquitetura, uma vez que a Engenharia de *Software* Tradicional apresenta-se muito limitada para esse tipo de arquitetura.

Com sistemas de Arquitetura Orientada a Serviços (SOA) operando em ambiente de execução distribuído e heterogêneo, os engenheiros de tais sistemas são confinados pelos limites da engenharia de *software* tradicional. SOA está rapidamente emergindo como principal paradigma de computação distribuída para o desenvolvimento, integração e manutenção de aplicações corporativas. Muitas organizações estão agora em seu uso precoce de SOA, e assumem que eles podem simplesmente aplicar princípios e técnicas dos paradigmas pré-existentes da engenharia de *software*. Aplicações habilitadas SOA operam em ambientes de execução heterogêneos, distribuídos, não-deterministas, imprevisíveis e altamente dinâmicos, portanto, engenheiros SOA rapidamente encontram os limites de tais paradigmas da engenharia de *software* tradicional, que não fornecem qualquer conselho de estilo específico.

Diante desse contexto, tornou-se cada vez mais evidente a necessidade de sistemas mais flexíveis, aderentes e de fácil e rápida adaptação aos processos de negócio. Surge então um novo paradigma de arquitetura de *software*, baseado no conceito de Serviço, e que tem como principais características o baixo acoplamento, dinamismo e adaptabilidade, a reutilização de serviços, interoperabilidade e a independência de tecnologia: Arquitetura Orientada a Serviço.

Começa-se então a discutir a Engenharia de *Software* de Serviço como uma disciplina emergente que envolve parte da Engenharia de *Software* Tradicional e introduz outros novos conceitos para o desenvolvimento de sistemas baseados em SOA.

Engenharia de *Software* de Serviço é a ciência e aplicação de conceitos, modelos, métodos e ferramentas para projetar, desenvolver (fonte), implantar, testar, fornecer e manter sistemas de *software* alinhados ao negócio e baseados em SOA de forma disciplinada, reprodutível e repetíveis.

Diante de tantas vantagens proporcionadas pela Arquitetura Orientada a Serviço, surge a necessidade de revisitar os métodos e ferramentas utilizadas para o desenvolvimento de sistemas baseados em serviços, uma vez que a Engenharia de *Software* Tradicional não consegue atender às peculiaridades de tais sistemas: passa-se então a discutir à respeito da Engenharia de *Software* de Serviços. A Engenharia de *Software* de Serviço tem seus primeiros relatos em meados de 2009 em um seminário (KUROIWA, 2011), e é definida a seguir:

> *Software service engineering is the science and application of concepts, models, methods, and tools to design, develop (source), deploy, test, provision, and maintain usiness-aligned and SOA-based software systems in a disciplined, reproducible, and repeatable manner.*

Os principais desafios listados para a evolução da Engenharia de *Software* de Serviço são: Alinhamento Negócio-TI, adaptabilidade; Novos modelos e abstrações para representar e lidar com a dinâmica SOA; Como lidar com a heterogeneidade; O mapeamento de requisitos; Componibilidade e Testes (MAXWELL, 2014, 2013).

2.6.1 Arquitetura Orientada a Serviços

Uma arquitetura de *software* é um conceito abstrato que dá margem a uma série de definições. A definição usada pelo *ANSI/IEEE* afirma que uma arquitetura de *software* trata basicamente de como os componentes fundamentais de um sistema se relacionam intrinsecamente e extrinsecamente (MAXWELL, 2014, 2013).

Basicamente, Arquitetura Orientada a Serviços (SOA) é um paradigma de construção e integração de *software* que estrutura aplicações em elementos modulares chamados serviços. O serviço, a unidade fundamental de uma arquitetura SOA, é um elemento computacional que tem como propósito desempenhar uma função específica e que pode ser utilizado por um cliente. Segundo os conceitos básicos de SOA, um serviço é composto por uma interface e uma implementação. Geralmente, um serviço consiste em uma função de negócio desempenhada por um módulo de *software* (a implementação) e encapsulada por uma *interface* bem definida e acessível àqueles que desejam utilizar o serviço, ou seja, os potenciais clientes. Os clientes do serviço não têm acesso aos detalhes de como ele foi construído, mas apenas aos detalhes expostos em sua *interface*. A interface define as funções desempenhadas pelo serviço e eventuais precondições para utilizá-lo, mas não revela como estas funções são realizadas. Esta forma de encapsulamento é conhecida como caixa-preta e é um princípio característico de paradigmas como orientação a objetos e componentes de *software*. Porém, diferentemente destes, serviços representam funções completas de negócio e são projetados de modo a serem usados não somente no âmbito de um programa ou sistema, mas no âmbito da organização ou até entre organizações (PAPAZOGLOU, 2003).

Desta maneira, uma arquitetura SOA possibilita uma infra-estrutura para computação distribuída, por meio de serviços que podem ser fornecidos e consumidos dentro de uma organização e entre organizações, por meio de redes de comunicação como a Internet. Uma arquitetura SOA básica é caracterizada pelas interações entre três tipos de agentes de *software*: os Provedores de Serviço, os Consumidores (ou Clientes) de Serviço e o Registro de Serviço (FUGITA, 2009). As interações entre estes agentes podem ser visualizadas na Figura 9.

Os serviços são oferecidos pelos Provedores de Serviço, organizações responsáveis por desenvolver suas implementações, fornecer suas descrições e prestar suporte técnico e de negócio. De modo geral, os Provedores disponibilizam módulos de *software* (as implementações dos serviços) que podem ser acessados através de uma rede e publicam suas descrições em um Registro de Serviços, agente que abriga informações sobre as funções oferecidas, os requisitos para se utilizar o serviço e orientações sobre como realizar a interação. É o Registro de Serviços que torna essas informações disponíveis para serem consultadas por clientes em potencial. Por sua vez, os Consumidores de Serviço são os agentes que necessitam solicitar a execução de um Serviço. Os Consumidores buscam nos Registros a descrição de um serviço que satisfaça às suas necessidades e, ao encontrá-la, utilizam esta descrição para ligar-se ao Provedor e realizar a invocação do serviço. Note que os papéis de Provedor e Consumidor são lógicos, de modo que um mesmo agente pode exibir características de ambos dependendo do contexto (PAPAZOGLOU, 2003).

A palavra serviço pode ser definida como a execução de trabalho ou desempenho de

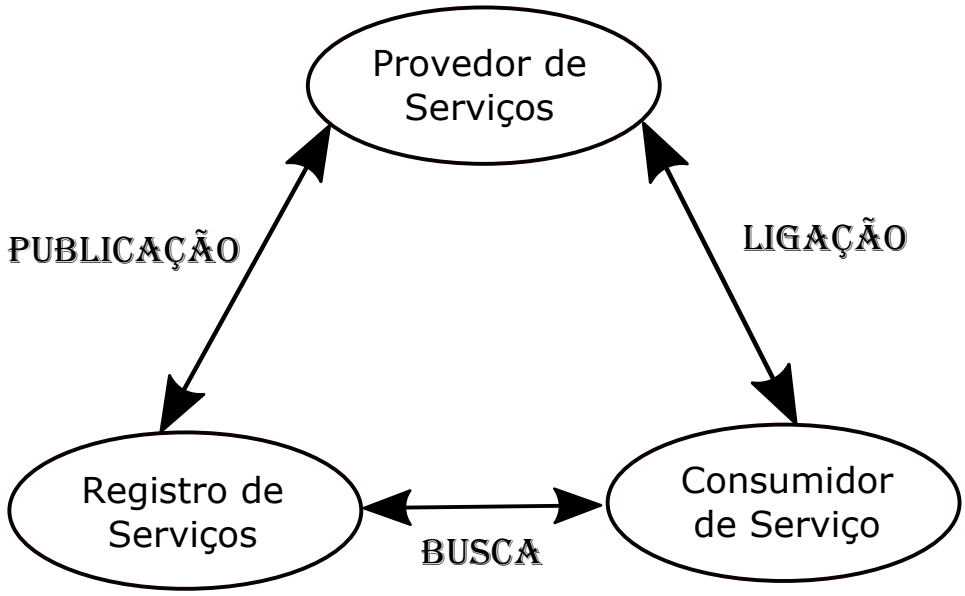


Figura 9: Agentes de uma arquitetura SOA (FUGITA, 2009)

funções, ordenados por um requisitante. No contexto específico de sistemas de *software*, serviço ainda está ligado a conceitos como a capacidade de realizar trabalho para outro, a especificação do trabalho oferecido e a oferta de realizar trabalho para outro. No contexto de SOA, um serviço tem significado semelhante, já que consiste em um *software* capaz de realizar uma função específica quando solicitado por seus consumidores.

Os diversos autores da literatura sobre o assunto citam e descrevem seus próprios conjuntos de características de um serviço, mas não há um consenso sobre o que define exatamente um serviço, pois cada um foca em aspectos diferentes do paradigma. Por isso, cada uma destas características deve ser analisada para chegar-se a um entendimento do que vem a ser orientação a serviços (MAXWELL, 2014, 2013).

A orientação a serviços pode trazer uma abstração entre a lógica de negócio e a infra-estrutura de TI, gerando um desacoplamento entre os modelos de processos de negócios e a arquitetura de sistemas de informação. Em uma arquitetura orientada a serviços dando suporte tecnológico aos processos de negócio de uma empresa, a camada de serviços localiza-se exatamente entre a camada de processos e a infra-estrutura de aplicações de Tecnologia da Informação (TI) (ERL, 2007). A camada de serviços cria uma abstração entre os processos de negócio e as aplicações, conforme mostra a Figura 10

Baseado no princípio de composição, é possível dividir a camada de serviços em mais três camadas de abstração que determinam o tipo e a granularidade dos serviços. Assim, temos: a camada de serviços de aplicação, a camada de serviços de negócio e a camada de orquestração de serviços. Na Figura 10, pode ser visualizada a hierarquia das camadas de serviços.

Frequentemente, relaciona-se SOA à integração de sistemas, o que não condiz com a realidade. Basta observar que EAI (*Enterprise Application Integration*) surgiu com o objetivo principal de resolver os problemas da integração ponto a ponto, também conhecido como *"spaghetti integration"*, fornecendo um meio uniforme de integração, SOA tem ambições muito maiores. Além disso, EAI possui características algumas vezes conflitantes com SOA, como o fato de ser centrado em dados e não em processos e de não se direcionar pelo negócio. O que motiva a criação de um serviço não é a integração ponto a ponto. Entretanto, SOA dá grande ênfase às interfaces que, quando bem definidas, facilitam a

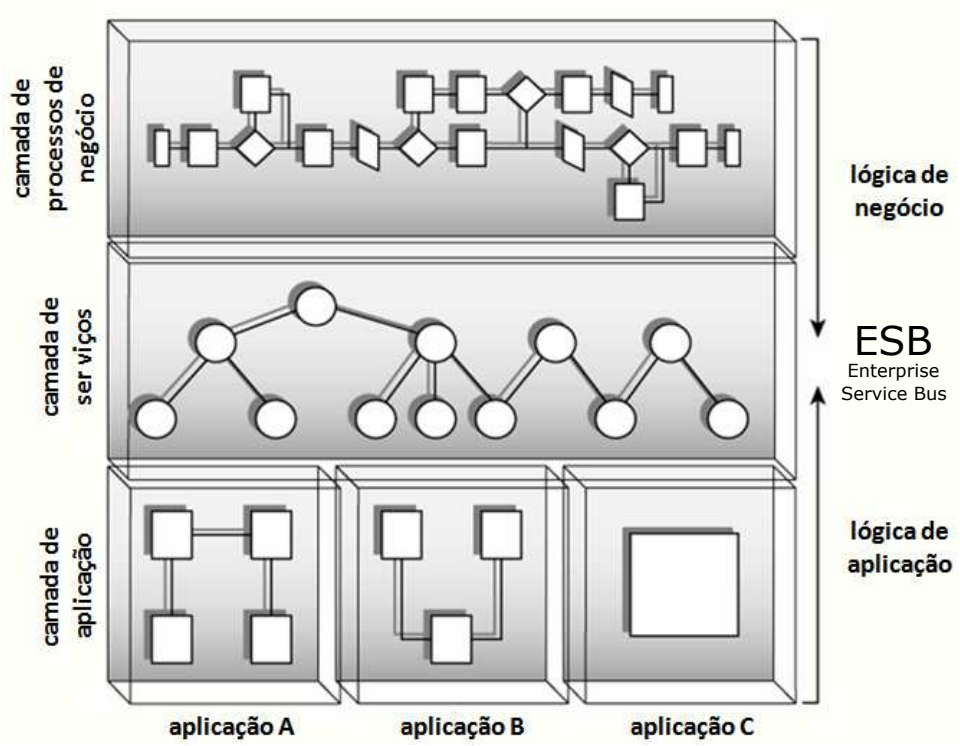


Figura 10: Camadas de uma arquitetura SOA (FUGITA, 2009)

integração e, até mesmo, o reuso de código. Para atender as necessidades de integração no contexto de SOA, evitando integração ponto a ponto, temos o *Enterprise Service Bus* (ESB). O ESB é um conceito, com algumas características de EAI, que fornece uma camada para a lógica de integração de serviços. Existem ferramentas para ESB que disponibilizam uma infra-estrutura flexível de conectividade para integração de aplicações e serviços, suportando. *Enterprise Service Bus* (ESB) SOA. O ESB tem como características: roteamento de mensagens entre serviços; conversão de protocolos de transporte entre requisitante e serviços; transformação do conteúdo de mensagens entre o requisitante e o serviço; Mensagens Síncronas e Assíncronas.

O foco nas interfaces é o que dá ao SOA a habilidade de obter acoplamento fraco, composição, reuso e vários outros objetivos do projeto (*design goals*). A essência de SOA está mais relacionada à habilidade de atender o negócio rapidamente. Para tanto, obter um nível de granularidade para os serviços identificáveis nos processos de negócio é uma abordagem fundamental. Isto contribui principalmente em questões como a orquestração no contexto de BPM (*Business Process Management*) e a justificativa de investimentos em TI, uma vez que permite a visibilidade da associação do serviço com o negócio. Podemos resumir BPM como a combinação de pessoas, tecnologia e processos. BPM é uma prática para melhorar a eficiência das organizações, automatizando os processos de negócios.

Os serviços de aplicação possuem menor granularidade e representam os serviços de infra-estrutura de uma arquitetura SOA, oferecendo funções específicas de tecnologia. O propósito dos serviços de aplicação é prover funções reusáveis e processar dados contidos em sistemas legados e novas aplicações desenvolvidas. Os serviços de negócio são os elementos fundamentais da arquitetura SOA, pois são aqueles que representam a lógica de negócio da organização. São formados pela composição de diversos serviços de aplicação para implementar a lógica de negócio. Podem representar tarefas de processo ou entidades de negócio. Por fim, a camada de orquestração de serviços realiza a ligação

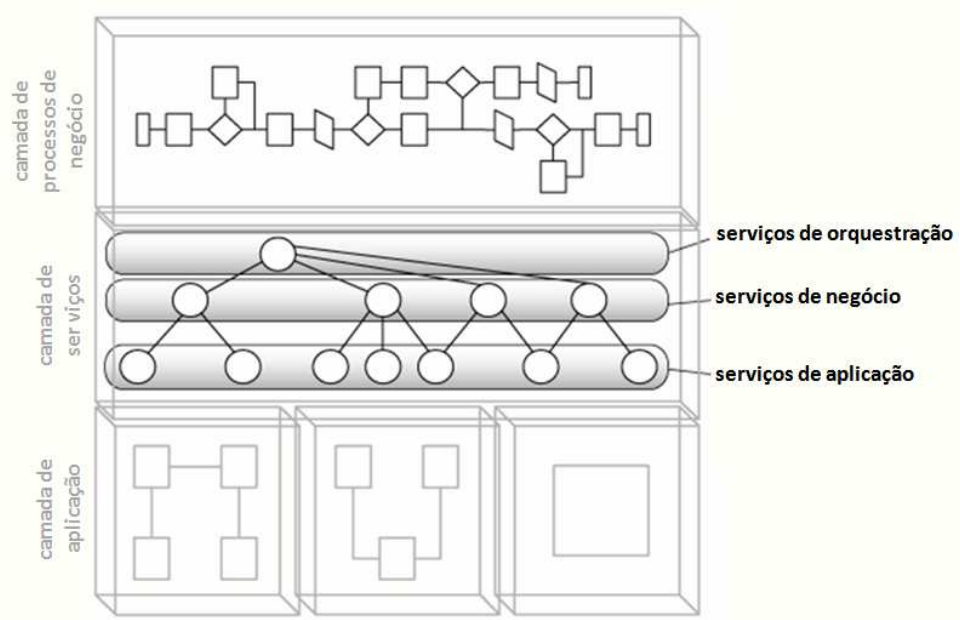


Figura 11: Camadas de serviços (ERL, 2007) (FUGITA, 2009)

entre os modelos de processos de negócio e os serviços da arquitetura SOA. O conceito de orquestração baseia-se na construção de processos de negócio a partir da composição de diversos serviços, por exemplo, utilizando uma linguagem de orquestração como o *Web Services Business Process Execution Language* (OASIS, 2013). Trata-se de uma linguagem para se especificar o fluxo de trabalho e lógica de negócio envolvidos nas chamadas de *Web Services*. Nesta camada estão os serviços de processo, que possuem alta granularidade e representam processos de negócio completos, encapsulando a lógica e regras de negócio envolvidas (FUGITA, 2009; MAXWELL, 2014, 2013).

Depois de encontrados os componentes e serviços, a fim de se atender a uma aplicação específica, são necessários que se efetue a orquestração, ou seja, que tais serviços sejam chamados em uma ordem lógica, e que sejam cumpridas todas as pré-condições das interfaces envolvidas (Lógica de Processo).

2.7 Relação entre os paradigmas OO, CBD e SOA

Muitas comparações já foram feitas entre desenvolvimento orientado a objetos (OO) e o desenvolvimento baseado em componentes (CBD). Componentes e objetos, possuem características similares, como a necessidade de interfaces bem definidas e a disponibilização de operações aos seus clientes. Essas semelhanças dificultam ainda mais o entendimento de OO e CBD. Algo muito parecido vem acontecendo no caso de CBD e SOA. Existem conceitos de CBD e SOA similares, por exemplo, encapsulamento, contratos e reuso, que podem dificultar a comparação dessas abordagens. Para uma melhor comparação é necessário compreender os conceitos essenciais envolvidos, quais sejam, processos de negócio, componentes realizadores dos serviços, entre outros.

Atualmente, existe uma infra-estrutura tecnológica pronta, que suporta o desenvolvimento OO, CBD e SOA sem muito esforço, mas não garante, apenas permite a aplicação das abordagens citadas. Uma comparação de CBD e SOA é muito abrangente e, portanto, envolve questões de várias dimensões.

Soluções como CBD e SOA se destinam, na maioria das vezes, a superar limitações

humanas, como lidar com grande quantidade de informações complexas. Para lidar com tal complexidade, a abstração é uma ferramenta chave, e está intimamente relacionada com o conceito de encapsulamento. Focando nesse aspecto essencial, Serafim (2009) descreve em seu artigo os níveis de encapsulamento. Baseando-se nesses níveis, é possível partir de simples linhas de código e chegar ao SOA, passando por CBD, mesmo considerando que essas abordagens foram concebidas em contextos diferentes. Para uma visão geral desses dois conceitos, observe o esquema da Figura 12.

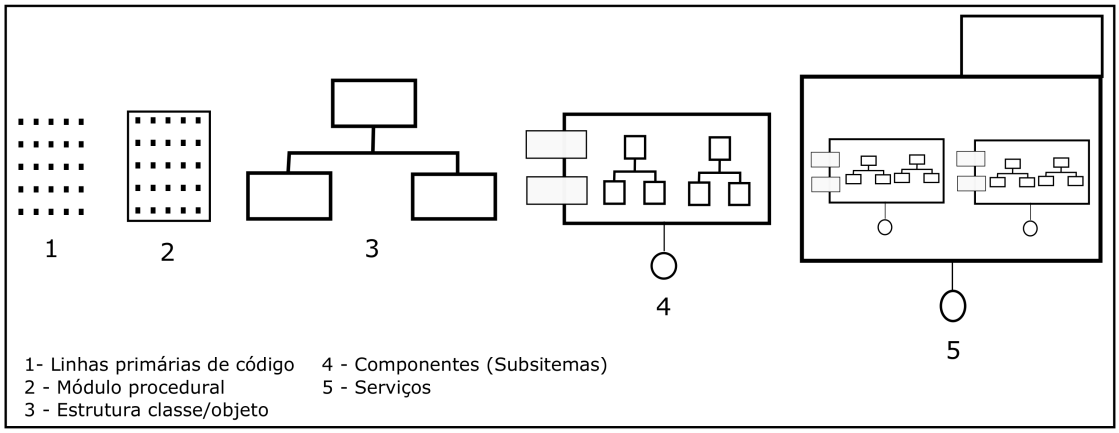


Figura 12: Esquema dos níveis de encapsulamento com serviços (SERAFIM, 2009)

No início, eram apenas linhas de código e vários comandos de salto de instruções (GOTO). Sua sintaxe é, em geral: goto destino, onde destino pode ser um *label* (rótulo ou nome de um endereço) ou um número, que representa um determinado endereço. As instruções passam a ser executadas no endereço apontado por destino. Muitas vezes, face à necessidade de se fazer uma alteração, era mais fácil reedificar o programa inteiro em função da desorganização que existia. Além disso, para realizar qualquer manutenção, era preciso memorizar todo o código envolvido. Com a adoção de sub-rotinas (primeiro nível de encapsulamento), foi possível reaproveitar as linhas de código. Estas linhas, encapsuladas através de uma sub-rotina, podiam ser chamadas quantas vezes fossem necessárias. Além disso, usava-se a decomposição funcional, ou seja, um problema era dividido em problemas menores (sub-rotinas) sucessivamente, e depois, as operações eram agrupadas em módulos lógicos. Infelizmente, a maioria das hierarquias de sub-rotinas geradas com essa abordagem não são suficientemente independentes para garantir reuso ou alterações controladas, pois, existe um acoplamento forte e sem controle. Assim, apenas esse nível de encapsulamento deixava a manutenção cara em ambientes com aplicações numerosas ou complexas. Neste caso, uma alteração no código poderia afetar de forma imprevisível todo o sistema, sendo necessário testá-lo novamente por completo (SERAFIM, 2009).

Os componentes são a melhor abordagem para implementar serviços, embora se deva entender que um aplicativo baseado em componente bem projetado não é necessariamente uma abordagem SOA. Conforme mostrado, temos que considerar muitas questões para melhor atender as mudanças no negócio, que é o objetivo. Uma abordagem orientada a serviços implica em uma camada de arquitetura de aplicativo adicional. A Figura 13 demonstra como as camadas podem ser aplicadas à arquitetura.

- **Camada de Serviços:** A camada de serviços é caracterizada por processos que realizam funções individuais de um negócio ou aplicação. Serviços são compostos de contrato, interface, lógica de negócios e são identificados no processo de negócio.

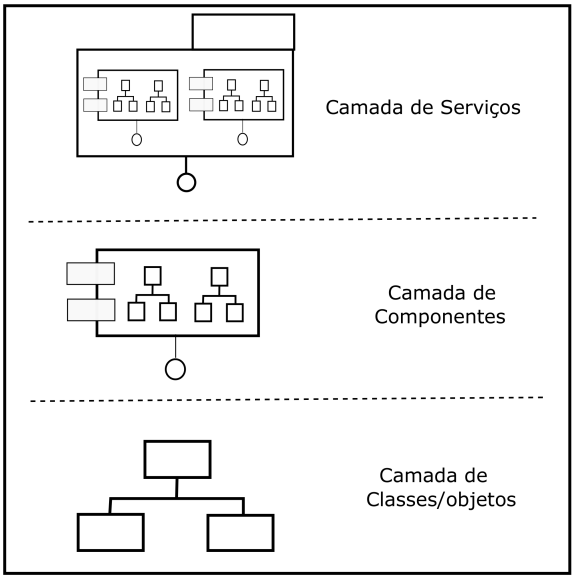


Figura 13: Camadas de uma implementação SOA (SERAFIM, 2009)

- **Camada de Componentes de negócio (Subsistemas):** Um componente de negócio é uma parte de um sistema que encapsula as regras de negócio e as expõe através de interfaces bem definidas. Componentes são independentes, substituíveis e modulares, eles ajudam a gerenciar a complexidade e encorajam a reutilização. Estes são identificados decompondo-se o modelo de classe conceitual (de análise).

- **Camada de Classes e Objetos:** Na camada de Classes e objetos é onde estão as classes, atributos e relacionamentos de um objeto. Os objetos colaboram entre si para realizar as regras de negócio de um componente específico. Por fim, o exemplo da Figura 6 mostra que SOA absorve o projeto baseado em componentes, ou seja, o serviço vai ser implementado por um ou mais componentes.

A partir disso, podemos analisar que o paradigma de orientação a serviços não significa uma ruptura com relação a outros paradigmas de desenvolvimento de *software*, pois representa uma evolução derivada da orientação a objetos e do desenvolvimento baseado em componentes e pode inclusive ser aplicado juntamente com eles. A seguir, é descrita a relação da orientação a serviços com os outros paradigmas e suas diferenças.

- **Orientação a objetos:** Conforme citado anteriormente, o paradigma de orientação a serviços tem como ponto em comum com o paradigma de orientação a objetos o fato de ambas serem maneiras de se construir *software* com base na separação de assuntos (ERL, 2007). Princípios como abstração, encapsulamento e composição de serviços foram formulados a partir de conceitos de orientação a objetos. Além disso, orientação a objetos é comumente utilizada para implementar a lógica de aplicação encapsulada em um serviço. Entretanto, os dois paradigmas possuem algumas diferenças que serão discutidas a seguir.

  A orientação a serviços prega o baixo acoplamento entre suas unidades (os serviços). Apesar de objetos possuírem rotinas desacopladas e até reusáveis, as classes por definição possuem relacionamentos entre si (agregação, herança), o que gera certo grau de dependência entre os objetos. Na orientação a serviços, muito das informações necessárias para o processamento está contido nas mensagens, de forma

que os serviços guardem o mínimo possível de informação de estado. A orientação a serviços encoraja a amarração da lógica de processamento com dados, mantendo mais informação contida nos objetos e criando uma dependência de estado.

- **Orientação a componentes:** Apesar de serem relacionados, os conceitos de componente e de serviço guardam diferenças entre si e não devem ser confundidos. Alguns princípios da orientação a serviço, como a abstração de interface e implementação e a transparência de localização, são oriundos de boas práticas da orientação a componentes. Porém, o serviço difere dos componentes ao não se limitar a uma determinada plataforma, linguagem ou tecnologia de implementação.

  Algumas fontes afirmam que serviços podem ser implementados utilizando-se componentes, o que pode levar à falsa ideia de que componentes e serviços podem ser mapeados de um para um. Existem dois pontos a serem considerados que desfazem esse tipo de conclusão. O primeiro é a granularidade de componentes e serviços. As funções oferecidas por componentes em geral possuem granularidade baixa, não possuindo valor direto para os processos de negócio. Serviços devem possuir granularidade um pouco mais alta, pois representam atividades de negócio e provêm funções de utilidade direta para o negócio. Portanto, a implementação de um serviço geralmente é construída pela composição de mais de um componente de baixa granularidade, resultando em algo de granularidade um pouco mais alta.

  O segundo é o fato de, apesar de possuírem interface e implementação separados assim como os componentes, os serviços operam sob um tipo de contrato que estabelece um acordo e cria expectativas com base em suas características semânticas. Tais características, pelo fato de serem complexas e de natureza humana, não podem ser representadas por um simples conjunto de assinaturas de funções, como as descrições de interface de arcabouços de componentes atuais. Mesmo a tecnologia de *Web Services*, que é frequentemente empregada na implementação de SOA, ainda não oferece uma maneira consolidada de representar estas características semânticas.

O paradigma de orientação a serviços tende a ser cada vez mais adotado pelas organizações industriais, tendo em vista benefícios que se espera que ele traga. Porém, esta transição impõe alguns desafios a serem considerados. Um novo método de análise e projeto orientado a serviços deve buscar resolver de modo direto a complexidade de análise e projeto, e indiretamente tratar requisitos de desempenho, segurança e governança. Na elaboração deste método, é especialmente importante focar nos princípios de orientação a serviços e em conceitos como granularidade e camadas de serviços (MAXWELL, 2014, 2013; FUGITA, 2009).

3 METODOLOGIA

Como se pode inferir a partir da análise das propostas existentes em relação aos requisitos do projeto para o desenvolvimento do *framework*, torna-se necessário consolidar os métodos analisados em um método que unifique as boas práticas de cada um e que atenda aos requisitos de análise e projeto orientados a serviços. Este capítulo descreve o Método escolhido de Análise e Projeto Orientado a Serviços ERL (20015), uma proposta de método elaborada a partir do estudo e análise dos métodos existentes e dos princípios de orientação a serviço.

Thomas Erl é um autor de publicações sobre SOA e membro de organizações de padronização nas áreas de SOA. Em suas publicações (ERL, 2007), o autor descreve um método para o desenvolvimento de uma arquitetura SOA e seus serviços.

Erl descreve o desenvolvimento de uma arquitetura SOA por meio de um ciclo de vida de desenvolvimento de serviços, focando principalmente nas atividades de análise e projeto, que são as etapas em que os serviços são identificados e especificados. O autor destaca o uso dos princípios de orientação a serviços durante estas atividades, para garantir que os serviços desenvolvidos sejam efetivos dentro da arquitetura SOA. Em determinados pontos, tanto da análise quanto do projeto, a aderência aos princípios de orientação a serviços é verificada.

3.1 Ciclo de Vida

O método procura oferecer flexibilidade em sua aplicação por meio de iterações durante a fase de Análise e de Projeto. O ciclo de vida descrito inicia-se com a Modelagem de Negócio, passa pelas fases de Análise e Projeto (abrangidas pelo método), seguindo adiante com atividades de Construção, como Implementação e Testes, conforme pode ser visualizado na Figura 14.

A fase de Análise Orientada a Serviços é o estágio inicial em que é determinado o escopo da arquitetura SOA em desenvolvimento. As camadas de serviços (orquestração, negócio e aplicação) são mapeadas e serviços preliminares são modelados na forma de serviços candidatos.

Na fase seguinte, Projeto Orientado a Serviços, os serviços que farão parte da arquitetura SOA já estão definidos, sendo necessário então determinar como eles serão construídos. A atividade de projeto é altamente baseada em padrões e influenciada pelos princípios de orientação a serviços. Quatro diferentes estratégias podem ser utilizadas para se projetar um serviço: serviços baseados em entidades, serviços de aplicação, serviços baseados em tarefas e serviços de processo.

A fase de Desenvolvimento de Serviços é o momento em que os serviços são efetivamente construídos. Nesta fase são considerados os aspectos técnicos e especificidades da

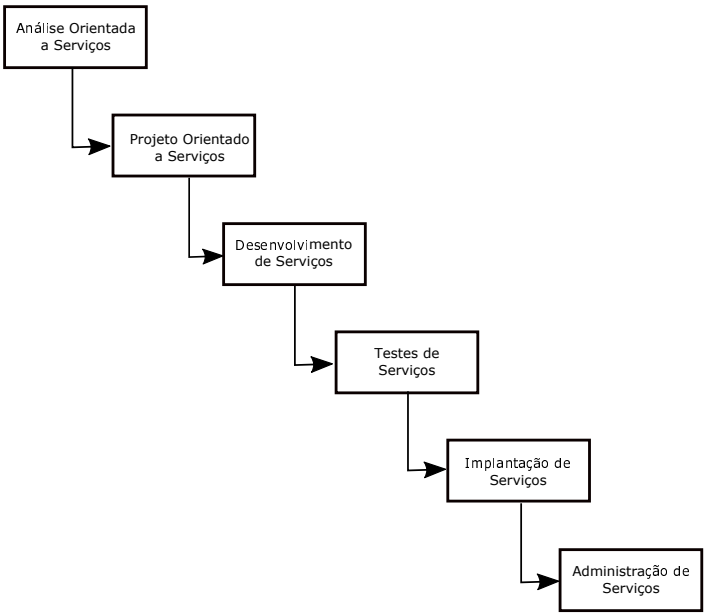


Figura 14: Ciclo de vida de desenvolvimento SOA (ERL, 2007)

tecnologia de implementação, como linguagens de programação, ambiente de desenvolvimento e plataformas.

Pelo fato de poderem ser reusados e participar de composições de maneira não previstas, os serviços devem ser rigorosamente testados. Na fase de Teste de Serviços, deve ser verificada uma série de requisitos dos serviços, como aderência aos padrões, comunicação com diversos tipos de consumidores, aspectos de QoS e tratamento de exceções e compensação.

A fase de Implantação de Serviços trata de instalar e configurar os componentes distribuídos que implementam os serviços, as interfaces de serviços e produtos de *middleware* associados em ambiente de produção. Deve-se definir a alocação física de cada componente, avaliar a capacidade da infra-estrutura satisfazer aos requisitos de desempenho esperados, estabelecer as configurações de segurança e verificar as integrações com sistemas e aplicações.

Finalmente, a fase de Administração de Serviços acompanha a execução dos serviços, monitorando seu uso e desempenho, realizando o controle de versão dos artefatos relacionados e dando manutenção para que os serviços operem de modo satisfatório (FUGITA, 2009).

3.2 Atividades de Análise

As atividades de análise no método proposto por Erl (ERL, 2007) correspondem à fase de Análise Orientada a Serviços do ciclo de vida. O objetivo principal desta fase é determinar quais os serviços que devem ser construídos e o escopo de cada um deles, ou seja, qual a lógica deve ser encapsulada por cada um.

Os passos para a realização da Análise Orientada a Serviços são exibidos na Figura 15, e são descritos a seguir.

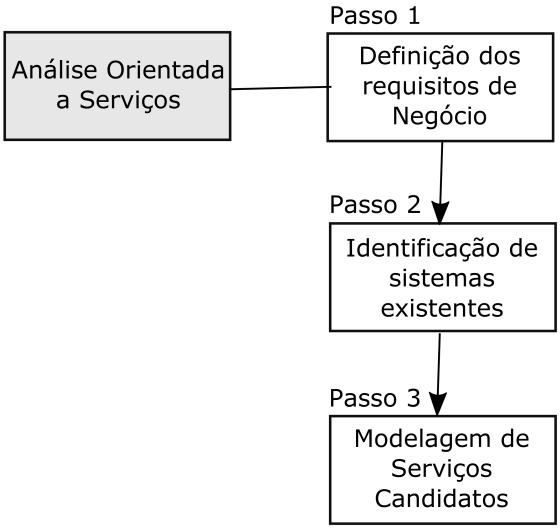


Figura 15: Análise Orientada a Serviços (ERL, 2005)

3.2.1 Definição dos requisitos de negócio

Nesta atividade, os requisitos de negócio relacionados à solução orientada a serviços sendo construída são levantados e documentados. Esta documentação de requisitos servirá como ponto de partida para a Análise Orientada a Serviços. Os requisitos de negócio devem estar definidos o suficiente para que um processo de automação possa ser especificado e utilizado no desenvolvimento da solução SOA (FUGITA, 2009).

3.2.2 Identificação de sistemas existentes

O objetivo específico desta atividade é identificar se existe alguma lógica de aplicação já existente que satisfaça total ou parcialmente a algum dos requisitos levantados na atividade anterior. Neste momento ainda não é necessário concentrar-se em como os serviços interagirão com as aplicações legadas, mas apenas realizar um levantamento de alto nível para prever quais sistemas podem ser afetados. Este tipo de informação será útil para a identificação dos serviços candidatos na etapa de modelagem (FUGITA, 2009).

3.2.3 Modelagem de serviços candidatos

A atividade de Modelagem de Serviços é o principal passo da fase de Análise Orientada a Serviços proposta por Erl. Nesta fase, são identificadas operações candidatas que devem ser agrupadas segundo seu contexto lógico, dando origem a serviços candidatos. Eventualmente, os serviços candidatos podem ser combinados em modelos de composição para formar a solução orientada a serviços. A Modelagem de Serviços compreende uma série de sub-tarefas, apresentadas na Figura 16 .

- No primeiro passo, ocorre a decomposição do processo de negócio documentado em uma série de passos de baixa granularidade, sendo que este nível de granularidade obtido pode diferir do nível original.

- Em seguida, são identificadas as operações candidatas dos serviços de negócio. Basicamente, os candidatos a operações serão os passos do processo identificados na etapa anterior, exceto aquelas atividades que são tarefas humanas que não deverão ser automatizadas e passos que já são executados por aplicações legadas que não podem ser encapsuladas por serviços.

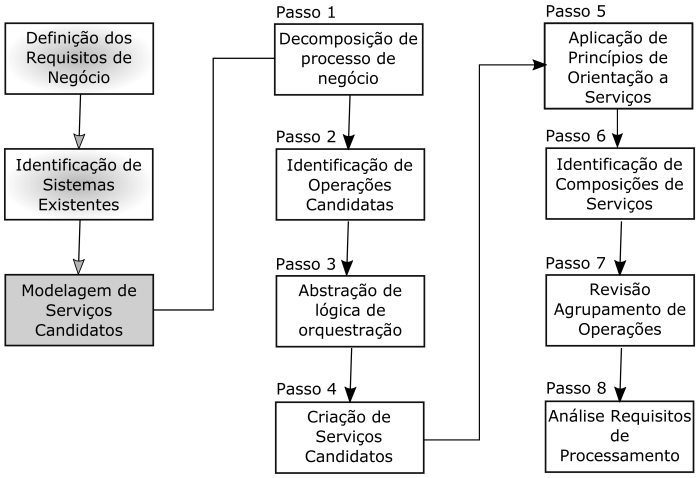


Figura 16: Modelagem de serviços candidatos (ERL, 2005)

- Caso a arquitetura SOA em questão possua uma camada de orquestração de serviços, então se deve definir que partes da lógica de aplicação serão abstraídas pela lógica de orquestração. Isto é, deve-se definir qual a lógica estará representada no processo ao invés de ser executada por um serviço de negócio. Regras de negócio, lógica condicional, lógica de exceção e lógica sequencial são alguns exemplos de lógica que podem ser abstraídos pela camada de orquestração.

- As operações candidatas devem então ser analisadas para se criar os candidatos a serviços de negócio. As operações deverão assim ser agrupadas de acordo com o contexto lógico correspondente. Por exemplo, serviços baseados em tarefas seriam agrupamentos de acordo com processo ou serviços baseados em entidades seriam agrupamentos de operações segundo os relacionamentos entre entidades. Operações podem ser adicionadas aos serviços para aumentar sua reusabilidade.

- Definidos os serviços candidatos, aplicam-se os princípios de orientação a serviços para garantir que eles realmente atendam aos requisitos de uma arquitetura SOA. As operações candidatas devem ter seus escopos e lógica revisados para que sejam realmente reusáveis e autônomas (sem dependências entre si).

- Para se identificar composições de serviços candidatos, devem ser analisados diversos cenários de execução do processo de negócio. Destas análises, podem-se inferir possíveis composições de serviços e concluir se os agrupamentos de operações são adequados. Nos cenários, devem ser considerados os casos de falha ou exceção.

- Baseados nos resultados do passo anterior, revisam-se os agrupamentos de operações candidatas em serviços.

- Por último, opcionalmente, podem-se revisar os requisitos de processamento dos serviços candidatos. Tal revisão tem o propósito de determinar qual a lógica necessária para executar as operações candidatas e se tal lógica já existe ou deve ser desenvolvida.

Para documentar os serviços em seu método de análise de projeto, Erl propõe uma notação semelhante a uma notação de classe da UML, que exibe um serviço com suas operações. Esta notação é utilizada em diagramas que exibem composições de serviços. A notação utilizada por Erl pode ser vista na Figura 17.

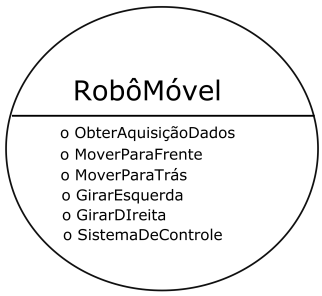


Figura 17: Notação ERL para representar um serviço (ERL, 2005)

3.3 Atividades de Projeto

O método descrito por Erl tem suas atividades de projeto especificadas na fase de Projeto Orientado a Serviços do ciclo de vida. Os principais objetivos desta fase são definir as interfaces dos serviços candidatos modelados na Análise, garantir a adequação aos princípios de orientação a serviços e definir quais padrões serão suportados e utilizados na implementação dos serviços. Como resultados, tem as especificações dos serviços nas camadas de negócio, aplicação e orquestração. Os passos do Projeto Orientado a Serviços podem ser visualizados na Figura 18 e são descritos nas seções seguintes (FUGITA, 2009).

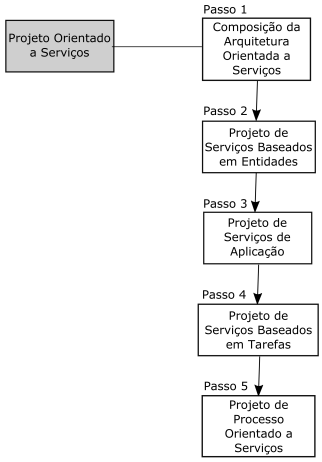


Figura 18: Projeto orientado a serviços (ERL, 2007)

3.3.1 Composição da arquitetura orientada a serviços

Nesta primeira atividade, são determinadas quais camadas de serviços serão utilizadas na arquitetura sendo construída. Devem-se definir também os padrões que serão utilizados na especificação e implementação dos serviços. Por exemplo, se for utilizada a tecnologia *Web Services*, deve-se estabelecer quais especificações de XML e *Web Services* serão utilizadas e quais extensões serão necessários (FUGITA, 2009).

3.3.2 Projeto de serviços baseados em entidades

Serviços de negócio baseados em entidades representam entidades de dados definidas no modelo de negócio da organização. Este tipo de serviço é completamente independente de processo de negócio, podendo ser reusado por qualquer solução que necessite acessar

dados relacionados a uma entidade particular. Serviços baseados em entidades fazem parte da camada de serviços de negócio.

A atividade de projeto de serviços baseados em entidade tem como propósito especificar as interfaces e a lógica encapsulada por eles. Para a especificação das interfaces, Erl propõe o uso de documentos WSDL, que podem ser utilizados posteriormente para implementação com Web Services. Os esquemas de dados das entidades tratadas pelo serviço são definidos e representados na forma de esquemas XSD e comporão as mensagens trocadas pelas operações do serviço.

Princípios de orientação a serviços devem ser também aplicados aos serviços. Serviços baseados em entidades são intrinsecamente autônomos pelo fato de cada um ser responsável por manipular uma entidade ou grupo de entidades específico. Também não preservam informação de estado, uma vez que possuem pouca lógica de fluxo de trabalho encapsulada. Deve-se atentar para a reusabilidade do serviço, possivelmente adicionando operações que podem ser úteis em outros contextos além do considerado para uma solução em particular (FUGITA, 2009).

3.3.3 Projeto de serviços de aplicação

Os serviços da camada de aplicação são a principal força de trabalho da arquitetura SOA, executando funções demandadas pelas camadas de orquestração e de serviços de negócio. São abstrações orientadas a serviço do ambiente tecnológico da organização, não sendo por isso necessário considerar aspectos de negócio em seu projeto. A atividade de projeto de serviços de aplicação inicia-se com a análise das operações candidatas propostas, para verificar se possuem o nível de granularidade e reusabilidade adequados. Devem então ser definidos os dados de entrada e saída para cada uma das operações, possivelmente na forma de esquemas XSD. Finalmente, a definição da interface WSDL deve ser completada com as operações correspondentes. Como serviços de aplicação devem poder ser utilizados por diversos tipos de serviços de negócio, os dados de entrada e saída definidos para as operações devem ser definidos de modo simples e genérico. Ao aplicarem-se os princípios de orientação a serviços, deve-se atentar para que eles se mantenham autônomos sem dependências entre sua lógica de negócio. Por fim, devem ser identificadas as possíveis restrições de ordem técnica que se aplicam ao serviço de aplicação, como componentes, *Application Programming Interface* (API's) e adaptadores necessários para a realização de conexões, restrições de segurança e possíveis requisitos de SLA (FUGITA, 2009).

3.3.4 Projeto de serviços baseados em tarefas

Serviços baseados em tarefas pertencem à camada de serviços de negócio e geralmente têm seu projeto simplificado por possuírem pouca necessidade de serem reusáveis. Tipicamente encapsulam lógica de fluxo de trabalho e realizam a orquestração de serviços de aplicação. A lógica de fluxo de trabalho deve ser especificada, podendo ser representada por diagramas de atividades. Esta lógica especificada pode auxiliar na descoberta de novas operações necessárias. As operações candidatas têm suas entradas e saídas mapeadas e são representadas por meio de documentos XSD e WSDL (FUGITA, 2009).

3.3.5 Projeto de processo orientado a serviços

Os processos de negócio orientados a serviços correspondem à camada de orquestração da arquitetura SOA. O processo corresponde ao fluxo de execução de serviços de

negócio, com suas regras e lógica, descrevendo o que é chamado de orquestração de serviços. No método proposto por Erl, a linguagem WS-BPEL é utilizada para representar as definições dos processos de negócio, porém esse processo pode ser adaptado para outras linguagens. Nesse caso, Enquanto o projeto de serviços baseados em tarefas e em entidades concentrava-se em definir as interfaces dos serviços e as mensagens das operações, o projeto de processos de negócio busca criar um fluxo WS-BPEL que implemente a lógica de negócio relegada à camada de orquestração durante a Análise Orientada a Serviços. Tal lógica que foi deixada de fora dos serviços especificados é abstraída em um processo de negócio separado. O primeiro passo do projeto de processos recebe como entrada a descrição da lógica de fluxo de trabalho a ser abstraída pela orquestração, o serviço de processo candidato e os serviços de negócio projetados nas atividades de projeto anteriores. Devem-se levantar quais os tipos de dados e informações serão necessários para executar a lógica e invocar os serviços de negócio. Com base neste levantamento, devem-se especificar os esquemas de dados das mensagens utilizadas no processo e a interface do serviço de processo. Neste ponto muitos esquemas definidos para os serviços de negócio poderão ser reaproveitados. Definidas a interface e as mensagens, parte-se para o desenvolvimento da especificação WS-BPEL do processo. Neste passo, será especificado como os serviços serão orquestrados. Devem-se definir quais serviços serão invocados, criar as regras de negócio envolvidas nas invocações e o tratamento dos dados trocados com os serviços. Com os processos WS-BPEL especificados, deve-se analisar a lógica de fluxo de trabalho criada frente aos cenários de execução discutidos anteriormente. Neste passo, pode ocorrer um refinamento do processo e dos serviços de negócio, dando origem a algumas alterações necessárias de acordo com os cenários (FUGITA, 2009).

3.3.6 Artefatos

A documentação dos requisitos de negócio é um artefato utilizado na fase inicial de Análise Orientada a Serviços e especifica os requisitos relacionados à solução em desenvolvimento. Estes requisitos podem ser obtidos por meio de técnicas convencionais de levantamento utilizadas em métodos de desenvolvimento de *software* distribuído. A documentação dos serviços candidatos é o resultado da Análise Orientada a Serviços e consiste em uma descrição em alto nível dos serviços criados a partir do agrupamento de operações candidatas identificadas. Este artefato descreve a lógica de negócio executada por cada operação candidata do serviço. Durante o Projeto Orientado a Serviços, as mensagens trocadas pelas operações são definidas na forma de esquemas XSD e a descrição das interfaces é realizada por meio da linguagem de definição de interfaces WSDL. Os serviços de processo desenvolvidos para a camada de orquestração são especificados utilizando-se a linguagem de definição de processos WS-BPEL (FUGITA, 2009).

3.3.7 Análise do Método

O método proposto por Erl consiste em uma abordagem puramente top-down para identificação e especificação de serviços. Na atividade de modelagem de serviços candidatos, Erl descreve de forma detalhada os passos a serem executados para identificar operações candidatas e descrever os serviços candidatos resultantes. O autor coloca a modelagem de processo como parte da atividade de definição de requisitos de negócio, separando-a das atividades de identificação e descrição dos serviços que ocorrem durante a modelagem de serviços candidatos. Apesar de não serem especificados os papéis associados às atividades, esta separação permite que cada atividade seja executada por pessoas com conhecimentos específicos. Apesar de os recursos existentes serem identificados

durante a Análise Orientada a Serviços, o reuso destes recursos não é considerado para compor os serviços descritos na modelagem de serviços candidatos. Tal identificação servirá para definir se um passo do processo será executado por um serviço ou por uma aplicação legada, sendo que no segundo caso não é considerado como uma execução de serviço. Consequentemente, não há descrita no método nenhuma atividade para definição de como cada serviço será realizado. O conceito de serviços candidatos é utilizado pelo método de Erl como transição da Análise para o Projeto. Os serviços são considerados candidatos desde sua modelagem até o momento em que têm sua especificação definida. Trata-se de uma proposta bastante alinhada ao conceito de BPM, que é aplicado tanto durante a decomposição de processos de negócio em serviços quanto no projeto de processo de negócio orientado a serviço, atividade em que é aplicada a orquestração de processos em WS-BPEL. Na fase de Projeto, o método leva em consideração as camadas de serviços, ao descrever atividades de projeto voltadas para cada tipo de serviço: entidade, tarefa e aplicação. Cada uma destas atividades tem como objetivo especificar serviços focando nas particularidades de um determinado tipo de serviço. Erl dá bastante ênfase aos princípios de orientação a serviços, incluindo atividades específicas para validação tanto na Análise quanto no Projeto (ERL, 2007). Na atividade de modelagem de serviços candidatos, realiza-se uma avaliação para verificar se os serviços candidatos modelados possuem características de reusabilidade e autonomia. Já durante as atividades de projeto de serviços, são verificados os princípios de reusabilidade, autonomia, independência de estado e baixo acoplamento, dependendo do tipo de serviço sendo projetado. Como o método adota a tecnologia *Web Services* para a realização dos serviços, são utilizados os padrões XSD e WSDL para especificar os contratos de serviços (FUGITA, 2009).

4 ESTUDOS DE CASOS: ROBÓTICA MÓVEL E TECNOLOGIA ASSISTIVA

Neste capítulo será apresentada a abordagem prática realizada para estudo do processo de desenvolvimento baseado em todos os paradigmas vistos nas seções anteriores. Visando o desenvolvimento de uma biblioteca de componentes, bem especificada e devidamente documentada para a proposta de um *framework* para projetos de sistemas embarcados e robótica. Para validação do *framework* dois estudos de casos foram desenvolvidos como serviços: Robótica Móvel e Tecnologia Assistiva.

4.1 Metodologias no desenvolvimento dos sistemas embarcados

A utilização de sistemas computacionais vem crescendo na sociedade, com isso, aplicações de tempo real tornam-se frequentes. As aplicações de tempo real variam em relação à complexidade e às necessidades de garantia no atendimento dos requisitos temporais. As aplicações embarcadas possuem características que os diferem dos demais sistemas. Características como: sistemas dedicados, sistemas reativos, confiabilidade, restrições de tempo real, tamanho do código, desempenho, baixo consumo de potência e energia, físicas (tamanho e peso), devem ser consideradas no desenvolvimento de um sistema embarcado. Para lidar com essas restrições na construção de um sistema embarcado são utilizadas as linguagens de modelagem de alto nível de abstração. Sendo assim, o projeto de um sistema embarcado parte de uma visão abstrata e ao longo do desenvolvimento são feitos refinamentos até o produto finalizado.

O processo de desenvolvimento do projeto embarcado utilizará paradigma de ciclo de vida em cascata (*waterfall*), estruturado em cascatas de fases, como ilustra a Figura 19 onde o final de uma fase implica no início de outra. Este tipo de comportamento ressalta a qualidade de um modelo rígido e linear para o desenvolvimento de *software* embarcado de tempo real no sentido que uma fase começa após a outra numa direção linear. Para que o *software* embarcado tenha uma maior flexibilidade, caso seja necessário, a arquitetura do *software* embarcado poderá fazer uso do modelo em cascata revisto, o qual prevê a possibilidade de, a qualquer tarefa do ciclo, regressar a uma tarefa anterior de forma a contemplar alterações funcionais ou técnicas, caso a coordenação dos *softwares* embarcados venha requerer.

De acordo com este fluxo, as etapas de projeto compreendem:

- **Análise de Requisitos:** definição dos requisitos do sistema,
- **Especificação:** detalhamento das funcionalidades do sistema,

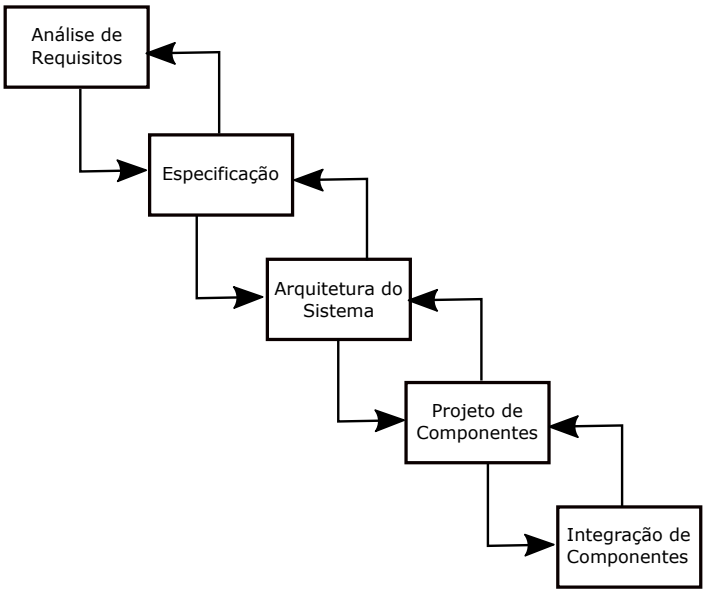


Figura 19: Níveis de abstração para o processo de projeto de sistemas embarcados. Fonte: (LEE; SESHIA, 2011).

- **Arquitetura do Sistema:** detalhamento interno do sistema, bem como os componentes que compõem o sistema,
- **Projeto de Componentes:** projetar os componentes do sistema,
- **Integração do Sistema:** integração dos componentes desenvolvidos para o sistema e a validação dos mesmos.

Existem vários modelos de ciclo de vida, no projeto dos sistemas embarcados dos estudos de caso são utilizadas duas estratégias de desenvolvimento: a abordagem *top-down*, em que o projeto inicia com uma visão mais abstrata e através de refinamentos sucessivos obtêm-se o sistema propriamente dito e a abordagem *bottom-up*, o qual o projeto é iniciado com uma descrição em nível de componentes e a partir destes componentes constrói-se o sistema completo permitindo tomar decisões quanto ao custo dos componentes e qual será a melhor arquitetura para o sistema. Sendo um misto das duas abordagens para que as decisões críticas possam ser tomadas a tempo evitando o retrabalho (LEE; SESHIA, 2011).

4.1.1 Especificação do *Hardware*

Para o *hardware* dos estudos de caso do projeto, é proposto o desenvolvimento de uma plataforma de *hardware*, para uma melhor descrição geral do sistema foi desenvolvido um diagrama de blocos, Figura 20, que é um diagrama cujo objetivo é a representação gráfica do processo e modelo dos projetos embarcados. Através de figuras geométricas e ligações, descrevem-se as relações entre cada subsistema e o fluxo de informação. O protótipo do *hardware* pode ser composto de sensores, módulos e atuadores que são configurados levando em consideração a morfologia mecânica e as relações geométricas de cada projeto, e assim poder determinar a melhor adaptação possível dos componentes.

O tratamento das informações captadas e transmitidas pelos módulos e sensores são processados através de um microcontrolador, dotado de um algoritmo de controle para ativar os atuadores. Os módulos e sensores estimulam os atuadores conforme a lógica de

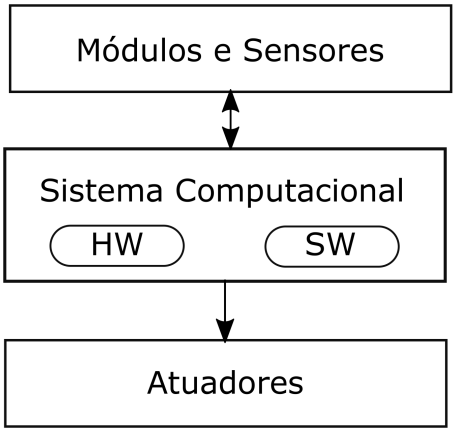


Figura 20: Diagrama de bloco da arquitetura dos estudos de caso do projeto. Fonte: (LEE; SESHIA, 2011).

programação e controle dos estudos de caso. O protótipo é um trabalho de *hardware* e *software* baseados em plataformas flexíveis *open-source* para o desenvolvimento. Para a construção do *hardware* dos estudos de caso do projeto vem sendo utilizado inicialmente plataformas *open-hardware*. A reutilização dos circuitos proporciona a criação de novas plataformas. A Figura 21, mostra o projeto do circuito e *layout* parcialmente modelado em *software* CAD (EAGLE, 2014).

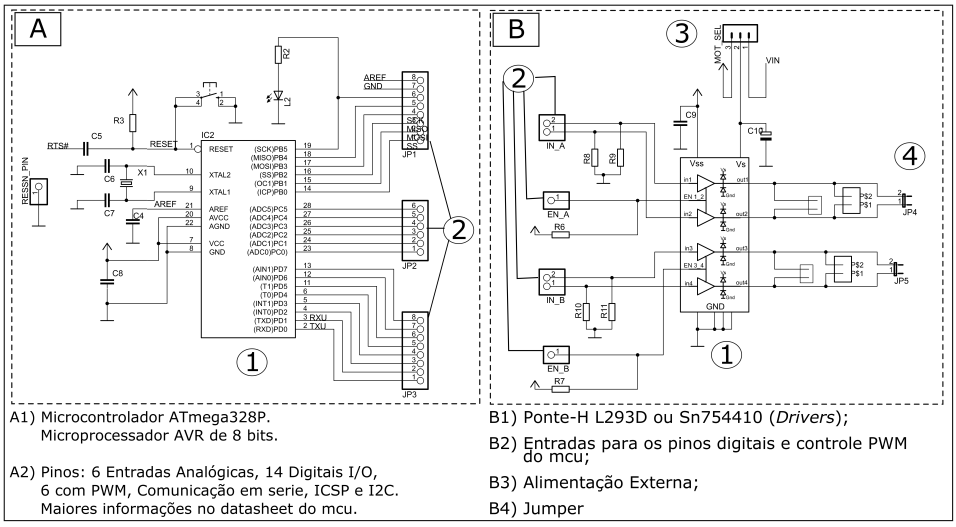


Figura 21: Microcontrolador ATmega328P e driver L293D. Fonte: (GUIBOT, 2014).

Testes e validações de alguns dos componentes de *harware* dos estudos de caso, utilizam inicialmente o microcontrolador ATmega328P (ATMEL, 2015). O microcontrolador é conectado com um *driver* ponte-h para o acionamento dos motores independentes do robô móvel ou dos motores de vibração tátil da tecnologia assistiva. A ponte-h utiliza duas pontes completas independentes (*full-bridge drive*), cujo objetivo é ativar os motores de Corrente Contínua (CC) dos projetos, uma vez que no projeto são utilizados motores CC, que exigem tensão maior do que a saída do microcontrolador pode fornecer. O projeto propõe uma plataforma de *hardware* como uma opção de baixo custo, para controlar o sentido de rotação dos motores e, por consequência, o sentido da corrente que circula entre os polos dos motores, visto que os motores CC alteram seus sentidos de rotação

quando inverte-se sua polaridade. O *driver* é controlado por três entradas, onde a primeira entrada é de habilitação (EnA) que ativa ou desativa a ponte-h e as outras duas (In1 e In2) determinam a circulação interna da corrente.

A escolha dos componentes se deu por uma combinação de fatores: baixo custo, tempo de desenvolvimento, facilidade de encontrar no mercado e programação do microcontrolador em três níveis (AVR *Assembler*, AVR GCC e *Wiring* C/C++). A comunicação entre a interface de controle do *framework* e o dispositivo se dá através de uma conexão serial padrão. O ATmega328P permite comunicação serial no padrão *Universal asynchronous receiver/transmitter* (UART) TTL (5 V), disponível nos pinos digitais 0 (RX) e 1 (TX), a Figura 22, apresenta o *driver FTDI USB para conexão serial*.

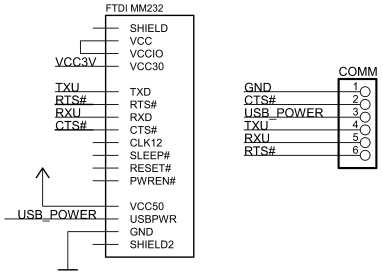


Figura 22: Conversor USB-TTL FTDI. Fonte: (GUIBOT, 2014).

Um conversor USB-TTL *Future Technology Devices International* (FTDI), dá suporte para os *drivers* relacionados à conversão TTL para sinais USB responsável pela comunicação serial. Um chip FTDI MM232 encaminha esta comunicação serial através da USB e os drivers FTDI fornecem uma porta virtual para o *software* no computador.

O circuito completo pode ser visualizado na Figura 23. O circuito é um projeto de *hardware open-source*, projeto Motoruino, uma placa baseada na plataforma Arduino, ele é projetado para trabalhar com motores, servos e sensores.

O projeto é fácil de usar, entender e estender, projeto é compatível com Arduino e *Shields*, o circuito possui um microcontrolador Atmega328P para o processamento e um *driver* Ponte-H (L293D), que permite controlar dois motores de corrente contínua. O L293D, suporta correntes de saída de 600mA por canal, isso é, você pode ligar até dois motores de 600mA cada. A voltagem suportada é de 4.5 à 36 volts. Isso permite controlar diversos tipos de motores respeitando-se, é claro, a corrente máxima suportada pelo chip. Recomenda-se utilizar motores com menos de 600 mA, apesar do CI suportar picos de 1.2A. Também é recomendado utilizar um dissipador de calor caso o CI comece a esquentar (GUIBOT, 2014).

4.1.2 Especificação do *Software*

Para conhecer melhor a API de programação *Wiring* C/C++ escolhido para implementação das bibliotecas do *firmware* do microcontrolador, utilizado para os estudos de caso, foi elaborado uma pesquisa dos principais conceitos da estrutura que envolvem a plataforma de *software* e *hardware* do *framework Wiring*.

Para começar precisamos começar contextualizando o *framework Processing*, uma linguagem de programação e ambiente de desenvolvimento integrado (IDE) *open-source*, construído para as comunidades de artes eletrônicas e *design* visual, com o objetivo de ensinar os fundamentos da programação de computador em um contexto visual. O projeto foi iniciado em 2001 por Casey Reas e Benjamin Fry, no Grupo de Estética e Computação no *MIT Media Lab*. Um dos objetivos declarados do *Processing* é fazer com que

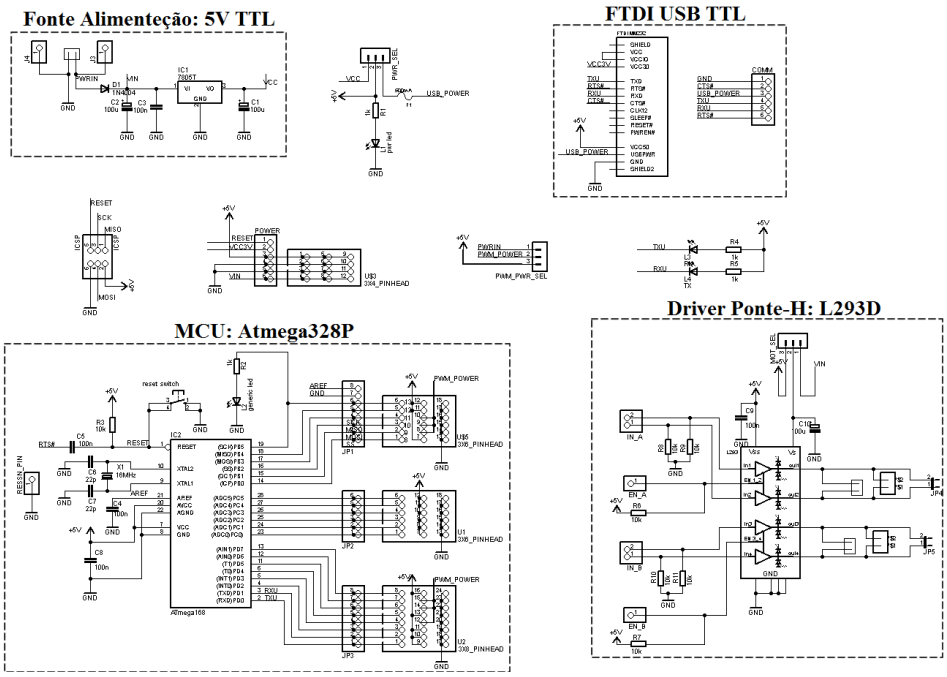


Figura 23: Circuito para controlar motores CC *open-source*

.

não-programadores comecem com a programação, através da gratificação instantânea de *feedback* visual. A linguagem baseia-se na linguagem Java, mas usa um modelo de programação sintaxe simplificada e gráficos.

Posteriormente ao *Processing*, veio a plataforma de *hardware Wiring*, uma plataforma eletrônica de prototipagem composto de uma linguagem de programação, um ambiente de desenvolvimento integrado (IDE), e um microcontrolador *single-board open-source*. Ele foi desenvolvido a partir de 2003 por Hernando Barragán. Barragán começou o projeto no *Interaction Design Institute Ivrea*. O projeto está atualmente desenvolvido na Escola de Arquitetura e Design na Universidade de Los Andes, em Bogotá, Colômbia. *Wiring* foi construído sobre o projeto *Processing*. A documentação foi criado pensando na comunidade onde especialistas, desenvolvedores intermediários e iniciantes de todo o mundo compartilham ideias, conhecimentos e sua experiências coletivas. O foco principal do projeto é criar um quadro multi-plataforma para todos diferentes tipos de microcontroladores. A plataforma *Wiring* fornece um ambiente de desenvolvimento integrado (IDE), que inclui suporte para linguagens de programação C, C++ e Java. Wiring suporta quase 80-90% dos microcontroladores Atmel da serie AVR 8 Bit e outros microcontroladores como PIC32, ARM Cortex M3 e MSP430. Considerando a programação e a compatibilidade com diversos hardware isso abre mais portas para o de desenvolvimento de projetos.

E assim surgiu o Arduino em 2005, uma plataforma de *hardware* e *software open-source*, que é um misto entre *Processing* e *Wiring*, sendo um fork desta segunda, uma plataforma de prototipagem rápida para eletrônica. O projeto é baseado em uma família de desenhos de placa de microcontroladores fabricados principalmente por *SmartProjects* na Itália, e também por vários outros fornecedores, usando vários microcontroladores 8 bits Atmel AVR ou 32 bits Atmel ARM processadores. Esses sistemas fornecem conjuntos de pinos I/O digitais e analógicos que podem ser conectados a várias placas de expansão ("escudos") e outros circuitos. As placas possuem interfaces de comunicação serial, incluindo USB em alguns modelos, para carregamento de programas de computadores

pessoais. Para a programação dos microcontroladores, a plataforma Arduino fornece um ambiente de desenvolvimento integrado (IDE), que inclui suporte para linguagens de programação C, C++ e Java.

A IDE Arduino parece praticamente o mesmo que a IDE Wiring e Processing, e funciona da mesma maneira, você pode trabalhar com *sketch* (esboços) com ambas as IDE's utilizando suas API's e bibliotecas. Atualmente estão surgindo muitas outras plataformas de *hardware* e *software* para diferentes arquiteturas de microcontroladores, onde a maioria de suas ferramentas de *hardware* e *software*, são baseados em *Processing*, *Wiring* e *Arduino*. A Figura 24, apresenta o diagrama de blocos parcial de como funciona a estrutura do *framework* para plataformas de *hardware*. Por exemplo, para gerar a API de programação de alto nível utilizado em *Wiring* C/C++, são encapsulados os registradores de baixo e médio nível dos microcontroladores em estruturas de programação em C e C++. Os arquivos implementados para API podem ser visualizados acessando os diretórios do *software* Arduino no computador *host* onde o mesmo foi instalado ou baixado, assim devem ser acessados os seguintes diretórios: arduino-vX\hardware\arduino\avr\cores\arduino.

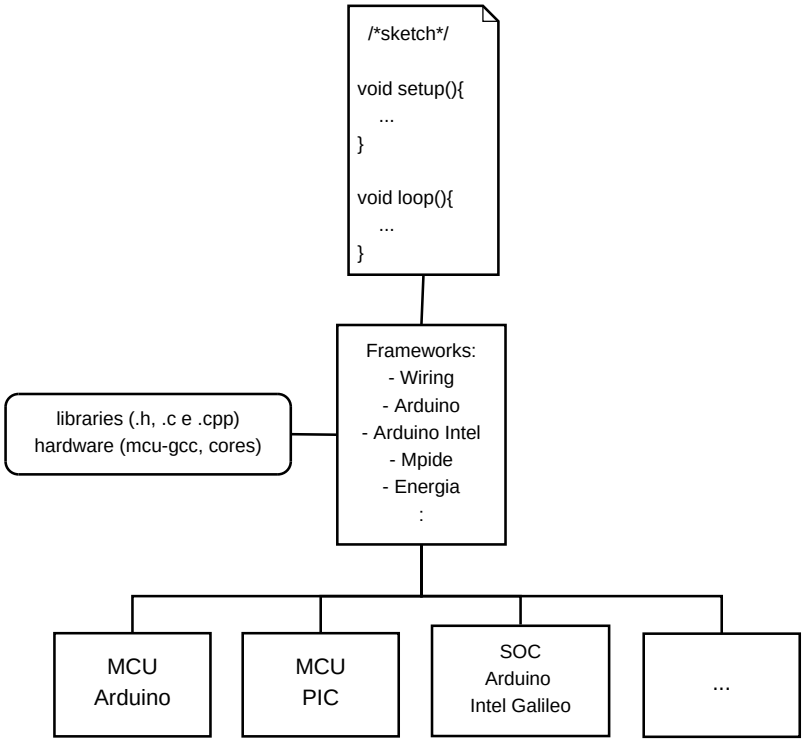


Figura 24: Diagrama de blocos parcial para *Frameworks* baseados na API *Wiring*.

O software é uma IDE, que é executado em um computador *host* onde é feita a programação, conhecida como *sketch*, na qual será feita *upload* para a placa de prototipagem, através de uma comunicação serial. O *sketch* feito pelo projetista dirá à placa o que deve ser executado durante o seu funcionamento. Para um melhor entendimento são apresentados na Figura **??**, imagens dos três *frameworks* descritos até agora. Cada um possui um ambiente de desenvolvimento integrado ou IDE, programa do computador que reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar este processo.

Porém, não são as únicas IDE's existentes para programação dos *hardwares*, existem outras variedades de ferramentas que podem ser utilizados. Conforme a arquitetura utilizada, cada fabricante possui *framework* próprio para dar suporte aos seus micropro-

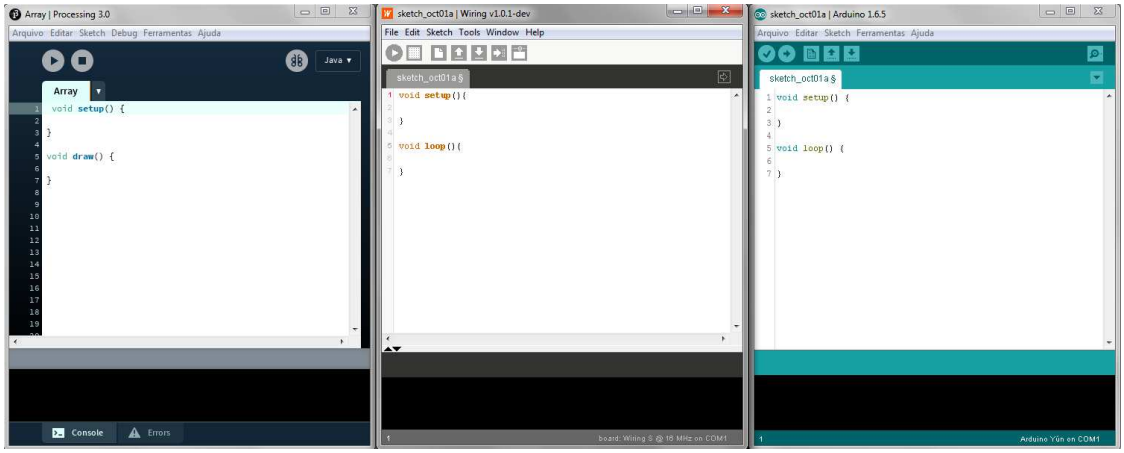


Figura 25: *IDE's dos Frameworks citados*.

cessadores, como por exemplo *Atmel Studio* para Atmel ou MPLAB para PIC, e assim por diante. Para programar no alto nível utilizando API *Wiring*, é necessário utilizar IDE's que deem suporte para as bibliotecas de programação, hoje muitas ferramentas estão suportando a API de programação Wiring, como as IDE's Eclipse Arduino ou CodeBlocks Arduino, entre uma variedade enorme de possibilidades. Neste trabalho, para os estudos de caso, o principal objetivo é o desenvolvimento da programação em C++ das bibliotecas dos componentes eletrônicos dos projetos, utilizando a API *Wiring*. A IDE de programação utilizada foram as IDE's Atmel Studio, Eclipse e Arduino.

Para demonstrar a eficiência e flexibilidade da API *Wiring* para microcontroladores vamos dispor de dois códigos de programação. Os dois códigos implementam a mesma lógica, porém cada um implementa a linguagem de programação em C baseado no GNU Compiler Collection (GCC) do microprocessador. Um pequeno exemplo de pisca LED (*Light Emitting Diode*) para microcontroladores Atmel-Atmega328P e Texas-MSP430G2553 são apresentados.

O microcontrolador ATmega328P faz parte da popular família de microcontroladores de 8 bits CMOS baseado na arquitetura AVR lançada pela ATMEL. Este microcontrolador possui altíssima performance podendo executar instruções com um ciclo de clock, fazendo com que o mesmo alcance 1 MIPS/MHz (1 Milhão de Instruções por Segundo por Mega Hertz), possibilitando ao programador otimizar o projeto combinando consumo de potência versus velocidade de processamento. O ATmega328P é utilizado nas placas Arduino e oferece performance que permite executar desde um simples programa que faz piscar um LED até um controle de um robô ou ainda um programa de controle de acesso controlado por rede. A seguir o código para piscar um LED utilizando a API de programação nativa do processador AVR.

```
1  /*
2   @author Carlos Solon
3   ------------------------*/
4  // Pisca Led - Atmega328P
5
6  #include <avr/io.h>
7  #include <util/delay.h>
8
9  enum {
10  BLINK_DELAY_MS = 1000,
11 };
```

```
12
13 int main (void)
14 {
15  /* seta pino 5 da PORTB para output*/
16  DDRB |= _BV(DDB5);
17
18 /* DDRB |= 1<<PB5; //Definir o pino como saída
19    DDRD &= ~(1<<PD2); //Definir o pino como entrada*/
20
21  while(1) {
22   /* seta pino 5 como alto para ligar o led*/
23   PORTB |= _BV(PORTB5);
24   _delay_ms(BLINK_DELAY_MS);
25
26   /* seta pino 5 como baixo para desligar o led*/
27   PORTB &= ~_BV(PORTB5);
28   _delay_ms(BLINK_DELAY_MS);
29  }
30
31  return 0;
32 }
33  }
```

A família *Texas Instruments* MSP430 de microcontroladores contém ultra-baixo consumo de energia, consiste em vários dispositivos com diferentes conjuntos de periféricos direcionados para várias aplicações. A arquitetura, combinada com cinco modos de baixo consumo de energia, é otimizado para alcançar vida útil da bateria em aplicações de medição portáteis. O dispositivo possui um poderoso CPU RISC de 16 bits, registos de 16 bits, e geradores constantes que contribuem para a eficiência máxima de código. O oscilador controlado digitalmente (DCO) permite *wake-up* a partir de modos de baixo consumo para o modo ativo em menos de 1 mS. Além disso, os membros da família MSP430 tem um conversor de 10 bits analógico-para-digital (A/D). As aplicações típicas incluem sistemas de sensores de baixo custo que captura os sinais analógicos, convertê-los em valores digitais, e depois processar os dados para exibição ou para transmissão para um sistema *host*. A seguir o código para piscar um LED utilizando a API de programação nativa do processador MSP430.

```
1 /*
2  @author Carlos Solon
3  ------------------------*/
4 // Pisca Led - MSP430G2553
5
6 #include <msp430.h>
7
8 int main( void )
9 {
10     WDTCTL = WDTPW + WDTHOLD; // Desabilita timer do  watchdog
11
12     P1DIR = 0b00000001; // P1.0 é uma saída, P1.1-7 são entradas
13     P2DIR = 0b00000000; // P2.0-7 são entradas
14
15     // Configure o Módulo Básico do relógio
16     // Isso define o relógio do MSP430 calibrado 1 MHz,
17     DCOCTL  = CALDCO_1MHZ;
18     BCSCTL1 = CALBC1_1MHZ;
19
20     // loop para sempre
```

```
21      while(1)
22      {
23          P1OUT = 0b00000001;       // ligado - P1.0
24          __delay_cycles(100000); // 0.1 segundo delay (100000 ciclos)
25          P1OUT = 0b00000000;       // desligado - P1.0
26          __delay_cycles(100000);
27      }
28
29      return 0;
30  }
31
32   }
```

Da para notar que o código do Pisca LED do microcontrolador Atmega328P e MSP430, possuem códigos distintos, porém executam a mesma ação, que nesse caso é acender e apagar o LED, cada microcontrolador é projetado com um microprocessador, logo cada um terá seu próprio conjunto de instruções e bibliotecas. Isso traz pouca flexibilidade para o desenvolvimento de sistemas embarcados. Não permitindo a extensão do código do projeto para diferentes microprocessadores. Isso pode ser resolvido se ambos os microprocessadores utilizarem a API de desenvolvimento *Wiring*. Agora, como exemplo, será mostrado como utilizar o mesmo código para o projeto Pisca LED utilizando a API Wiring para ambos os microcontroladores Atmega328P e MSP430. Isso é possível porque cada microcontrolador implementa a API *Wiring* baseado na arquitetura e instruções dos registradores de cada microprocessador, criando assim um padrão de linguagem de programação de microcontroladores de alto nível.

```
1  /*
2   @author Carlos Solon
3   ------------------------*/
4  // Pisca Led - Atmega328P e MSP430
5
6  void setup() {
7    // inicializa pino digital 13 como output.
8    pinMode(7, OUTPUT);
9  }
10
11 // loop para sempre
12 void loop() {
13   digitalWrite(13, HIGH);    //  Liga o LED
14   delay(1000);               // Espera 1 segundo
15   digitalWrite(13, LOW);     // apaga o LED
16   delay(1000);               // Espera 1 Segundo
17 }
18  }
```

A seguir, é apresentado na Figura 26, três IDE's de microcontroladores distintos que implementam a API *Wiring*. Da esquerda para direita temos a IDE Energia (Texas), IDE Arduino (Atmel) e IDE Mpide (PIC). É possivel visualizar que ambos os microcontroladores, cada qual com seu microprocessador, agora partilham do mesmo código para o projeto Pisca LED.

Levando-se em consideração a flexibilidade e portabilidade de utilizar essas plataformas de hardware e software *open-source*, as implementações dos códigos e bibliotecas dos estudos de caso do projeto serão desenvolvidos baseados na API *Wiring*, assim é possível desenvolver os módulos dos componentes eletrônicos do projeto, com baixo acoplamento.

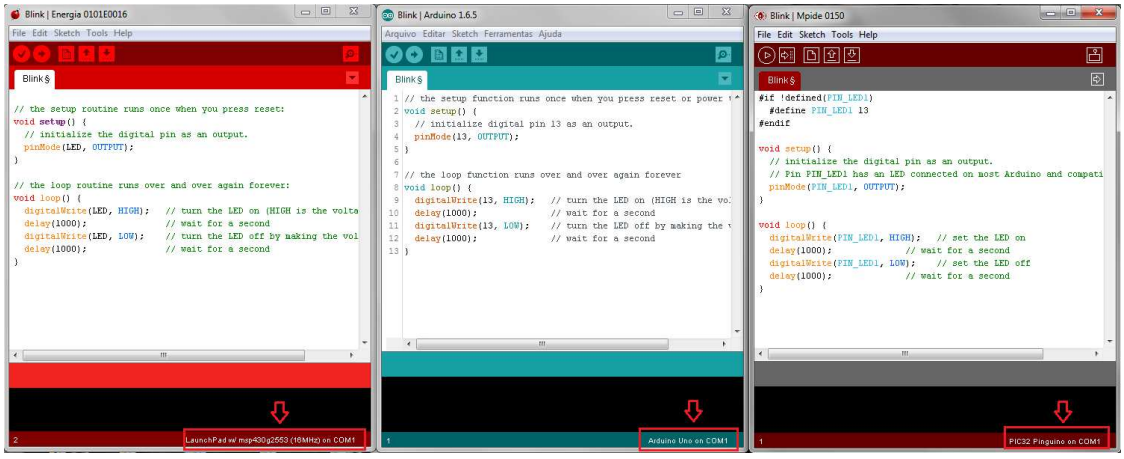


Figura 26: *IDE's dos Frameworks citados*.

4.2 Introdução à proposta EduBOT

Busca-se com o emprego das tecnologias na educação, uma melhor qualidade do ensino e ambientes de aprendizagem mais ricos e motivadores para os discentes. Dentre o amplo espectro de ideias e propostas, no que se refere aos artefatos computacionais, é notório observar que as soluções exploram, em sua maioria, apenas a vertente de *software*. No entanto, a demanda por novos aparatos de *hardware* vem crescendo sendo evidenciada, sobretudo, pelos esforços da comunidade acadêmica em propor a inserção da robótica com fins pedagógicos. No contexto educacional, a utilização da robótica pode ampliar significativamente a gama de atividades que podem ser desenvolvidas e promover a integração entre diferentes áreas do conhecimento. A construção de um novo mecanismo, ou a busca pela solução de um novo problema obriga o aluno a buscar conceitos em diversas disciplinas. A robótica tem, em tal contexto, um grande potencial como ferramenta interdisciplinar, religando fronteiras anteriormente estabelecidas entre várias disciplinas, possibilitando aos alunos ter uma vivência, na prática, do método científico, simulando mecanismos, através da construção de protótipos (SILVA, 2009).

Alguns projetos pedagógicos de robótica em sala de aula fazem uso, por exemplo, da metarreciclagem de lixo computacional para incentivar a criatividade sustentável na era digital, como por exemplo, o projeto EduBOT-v0.1 (EDUBOT, 2014). O projeto apresenta uma plataforma robótica para educação para auxiliar no ensino de componentes de *hardware* (mecânica e eletrônica) e *software* (programação). O projeto é aberto e encontra-se disponível na internet. O robô é baseado em projetos de desenho CAD abertos para a construção (MARTINS, 2011), Figura 27 .

O trabalho foi aplicado com alunos dos cursos de graduação de Ciência da Computação e Sistemas de Informações da Universidade Regional Integrada do Alto Uruguai e das Missões, Campus Santo Ângelo-RS (EDUBOT, 2014). Foram desenvolvidos robôs móveis utilizando equipamentos descartados da universidade (Metareciclagem), dando origem ao primeiro (I/2011) e segundo (II/2011) torneio de Robótica Livre da URI. Esse projeto de pesquisa foi desenvolvido em conjunto com o Departamento de Engenharias e Ciência da Computação da universidade. Os torneios tiveram como base as plataformas robóticas móveis desenvolvidas para navegação dentro de labirintos, o projeto teve inicio sobre os estudos e projetos de sistemas computacionais embarcados de *software* e *hardware* do autor(2010), durante o curso de Ciência da Computação. A Figura 28, apresenta as validações dos robôs móveis nos torneios.

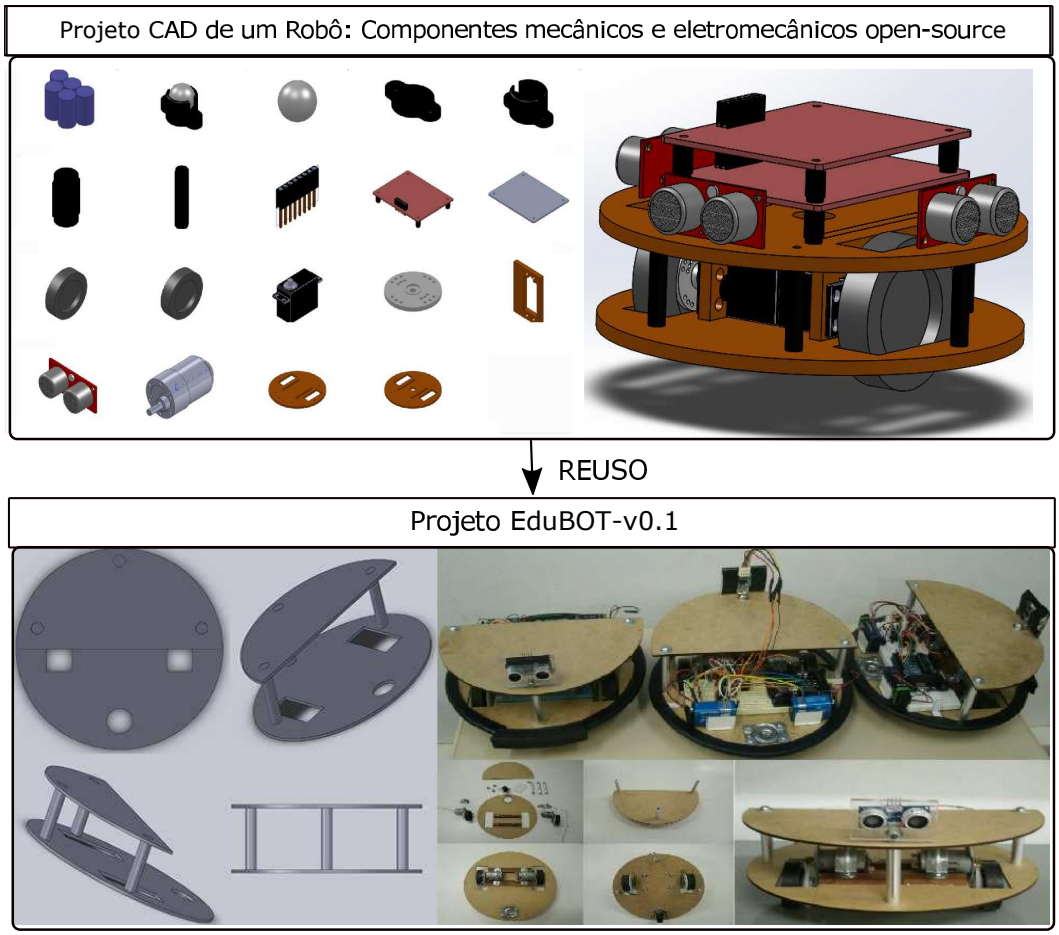


Figura 27: Projeto EduBOT-v0.1. Fonte: (EDUBOT, 2014; MARTINS, 2011).

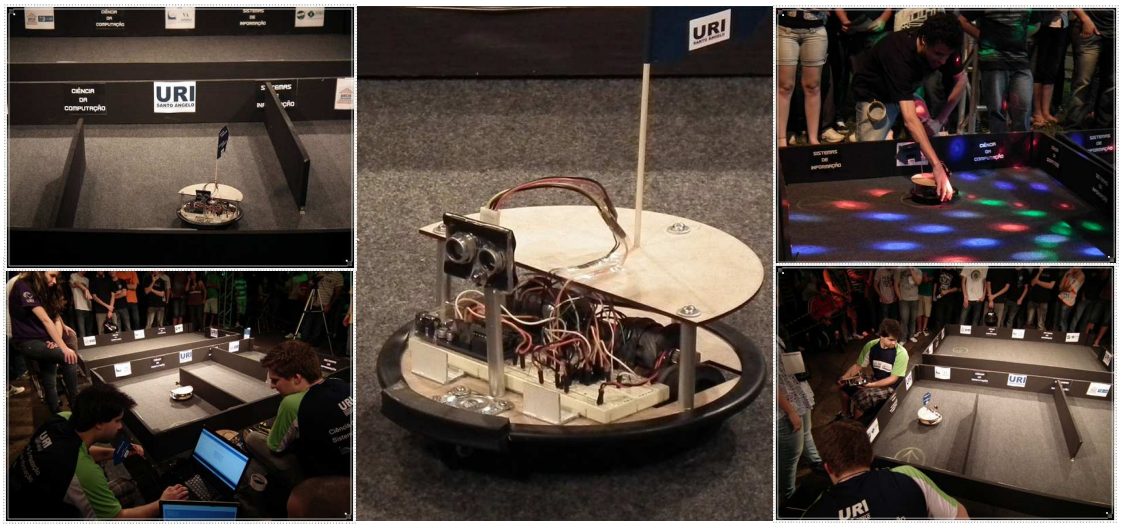


Figura 28: Torneios de robótica libre utilizando as plataformas robóticas montadas. Fonte: (EDUBOT, 2014; MARTINS, 2011).

Com o decorrer do tempo, o projeto passou a ter novas versões, EduBOT-v2 e v3 (GUIMARÃES; TAMAYO; HENRIQUES, 2014). O novo projeto não utiliza metariclagem, pois pretende-se navegar de maneira coordenada desde uma localização de origem até uma localização final em um ambiente de forma controlada. Para isso os modelos cinemáticos e dinâmicos que permitem a descrição do sistema de controle devem ser considerados. O projeto vem sendo aplicado em discentes do ensino superior de engenharia da UFRGS. Para a integração dos componentes físicos de maneira modular é levado em consideração diferentes *kits* de robôs móveis existentes (SILVA, 2009; MIRANDA; SAMPAIO; BORGES, 2010), o modelo definido para reutilização e referência dos componentes mecânicos e eletromecânicos do projeto EduBOT, são baseados nos modelos de robôs móveis da Pololu (POLOLU, 2014), Figura 29.

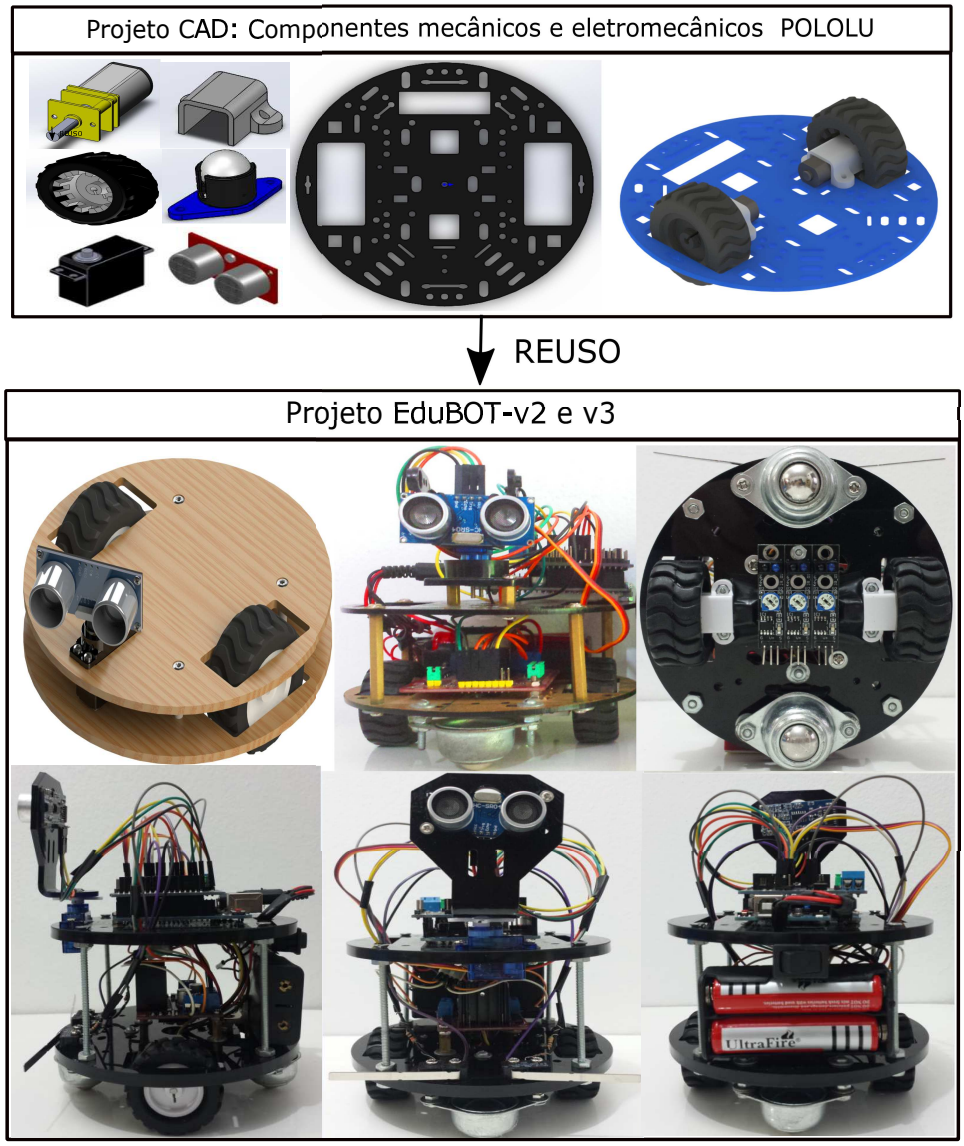


Figura 29: Projeto EduBOT-V0.2. Fonte: (GUIMARÃES; TAMAYO; HENRIQUES, 2014).

O robô tipo uniciclo é, em geral, o eleito por pesquisadores para experimentar novas estratégias de controle por possuir uma cinemática simples. É uma estrutura formada por duas rodas fixas convencionais, sobre um mesmo eixo, controladas de maneira indepen-

dente, e por uma roda passiva que lhe confere estabilidade. O sistema de tração-direção associado ao robô lhe permite controlar de forma independente suas velocidades linear e angular. As vantagens que derivam de sua estrutura mecânica e da eletrônica de controle fazem desta configuração a preferida para robôs de laboratório (SECCHI, 2008).

Os protótipos desenvolvidos do EduBOT v2 e v3 foram aplicados nos discentes da graduação de Engenharia da Computação, Engenharia Elétrica e Engenharia de Controle e Automação da UFRGS, ENG04009 - Turma U (2013/2), ENG04009 - Turma U (2014/1), CCA99003 - Turma A (2014/1), ENG04479 -Turma U (2014/2) e ENG04009 - Turma U (2014/2). A Figura 30, apresenta algumas imagens das aplicações do projeto com as turmas.

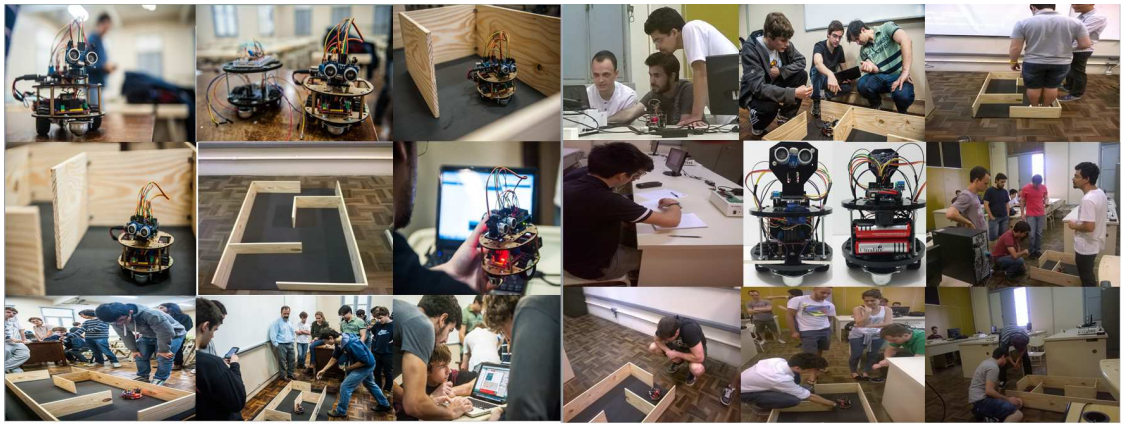


Figura 30: Validação do projeto EduBOT-V0.2 com os estudantes. Fonte: (GUIMARÃES; TAMAYO; HENRIQUES, 2014).

A seguir é apresentado o fluxograma de um algoritmo aplicado com os estudantes de simples solução de labirinto, o qual não usa memória extra e é de fácil implementação, Figura 31. O principio é começar a seguir paredes, e sempre que você chegar a um cruzamento sempre vire à direita (ou esquerda, mas sempre para o mesmo lado). Equivalente a um ser humano resolver um labirinto, colocando a mão sobre a parede direita ou esquerda) e segui-lá até encontrar uma saída. Também é possível marcar caminhos ou células que foram visitadas, para poder refazer a solução seguindo essas células posteriormente. Este método não irá necessariamente encontrar a solução mais curta, mas irá encontrar a saída caso haja uma. Esse algoritmo pode ser visto em funcionamento em vídeo (GUIMARÃES, 2015).

Para as turmas de semestres iniciais é utilizado paradigma de programação procedural, já para turmas de semestres avançados são utilizados paradigmas da programação orientado a objetos, cada grupo deve desenvolver algoritmos autônomos para resolver um pequeno labirinto. O código abaixo, mostra a estrutura parcial de um dos códigos que são passados aos estudantes, a fim de servir como referência inicial para a programação procedural do robô EduBOT v3. O código é passado incompleto para os estudantes para que eles completem a estrutura e lógica da programação procedural através do processo de heurística do projeto.

```
1 /*
2  @author Carlos Solon
3  ------------------------*/
4 /*Estrutura parcial de um dos código procedural distribuído para os
    estudantes. O robô possui dois motores DC independentes, sonar,
```

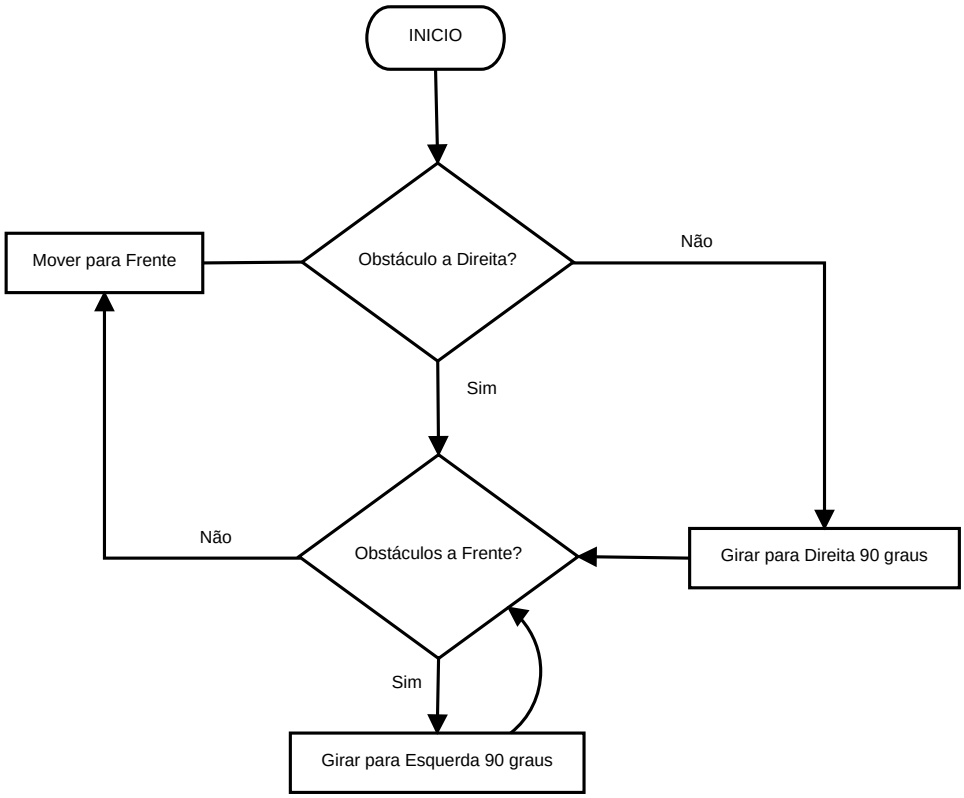


Figura 31: Fluxograma para algoritmo seguidor de parede. Fonte: (GUIMARÃES; TAMAYO; HENRIQUES, 2014).

```
    motor servo, botões fim de curso e sensores infravermelhos. Cada
    grupo deve implementar a sua lógica de programação!*/

// Declaração das bibliotecas utilizadas
#include<Servo.h>
            :
            :
// protótipos das funções

//Declaração de constantes e variáveis globais

//pinos digitais I/O Motor, Sonar, Botões, etc...
const int pinEpwm = 3; //PWM para o motor esquerdo
const int pinEa = 5; // pino 1 motor esquerdo
const int pinEb = 6; // pino 2 motor esquerdo
            :
            :
void setup(){
  // comunicação Série - USB.
  Serial.begin(9600);
  // função para setar os pinos dos motores (OUTPUT).
  setupMove();
  // função para setar os pinos do Sensor SR04
  setupUltrasonic();
            :
            :
}
void loop(){
```

```
31   unsigned int dist=0;
32   // retorna a distância em cm
33   dist = leDistancia();
34             :
35             :
36   // chamada da função para os motores
37   moverFrente();
38   delay(1000);
39             :
40             :
41 }
42 // Implementar função dos motores para frente.
43 void moverFrente(){
44 }
45 // Implementar função dos motores para trás.
46 void moverTras(){
47 }
48 // Implementar função dos motores para esquerda.
49 void moverEsquerda(){
50 }
51 // Implementar função dos motores para direita.
52 void moverDireita(){
53 }
54 // Implementar função para parar os motores.
55 void moverPara(){
56 }
57 // Implementar função para setar os pinos dos motores como Output
58 void setupMover(){
59 }
60 // Implementar função do sensor SR04. Retornar a distância em cm.
61 unsigned int leDistancia(){
62 }
63 // Implementar função para setar os pinos do sensor SR04
64 void setupUltrasonic(){
65 }
```

Para os estudantes de semestres mais avançados são pedidos as implementações de pelo menos duas das bibliotecas dos componentes eletrônicos utilizados no robô EduBOT v3. Eles devem criar as classes em C++ seguindo as boas praticas de programação, separando a modelagem da estrutura da classe em arquivos com extensões .h e .cpp. Depois de implementado as bibliotecas os estudantes devem criar um arquivo principal (*main.ino*) para instanciar os objetos das classes e criar a lógica de programação conforme os requisitos do problema ser a resolvido.

```
1  /*
2   @author Carlos Solon
3   ------------------------*/
4  /*Programa Main distribuído para os estudantes para programação
       orientado a objetos. Cada grupo deve implementar pelo menos duas
       bibliotecas dos componentes utilizados no projeto!*/
5
6  // Declaração das bibliotecas utilizadas
7  #include<Servo.h> //reutilizado do Arduino
8  #include <DCMotorBot.h> // implementar do zero (.h e .cpp)
9  #include <Sonar.h> // implementar do zero (.h e .cpp)
10             :
11             :
12 //Declaração de constantes e variáveis globais
```

```
13
14 //pinos digitais I/O Motor, Sonar, Botões, etc...
15 const int pinE1pwm = 3; //PWM para o motor esquerdo
16 const int pinEa = 5; // pino 1 motor esquerdo
17 const int pinEb = 6; // pino 2 motor esquerdo
18             :
19             :
20 // criação dos objetos
21 Servo myservo;
22 DCMotorBot MyMotors = DCMotorBot();
23 Sonar sr04(pinTrig, pinEcho, MAX_DISTANCE);
24             :
25             :
26 void setup(){
27    myservo.attach(10); //pino digital
28    myservo.write(90); // motor servo em 90 graus
29    //objeto MyMotors utilizando os métodos da classe
30    MyMotors.setEnablePins(pinE1pwm,pinE2pwm);
31    MyMotors.setControlPins(pinEa, pinEb, pinDa, pinDb);
32             :
33             :
34 }
35 void loop(){
36   unsigned int dist=0;
37   // ogjeto sr04 retorna a distância em cm
38   dist = sr04.ping_cm();;
39             :
40             :
41   // chamada da função para os motores
42   MyMotors.moveForward();
43   delay(1000);
44             :
45             :
46 }
```

Para melhorar a manutenção do código, é preferível sempre manter dois arquivos para cada classe, um para declaração (arquivo .h ou .hpp) e outro para a implementação (arquivo .cpp). E de preferência evitar colocar mais de uma classe num único arquivo, a não ser que as classes sejam extremamente dependentes uma da outra. Vejamos um exemplo de como fica o arquivo .h para a biblioteca do componente eletrônico Sonar.

```
1 /*
2  @author Carlos Solon
3  ------------------------*/
4 /*Exemplo da modelagem da classe Sonar.h*/
5
6 //diretivas de compilação condicional, verifica se um determinado
     identificador foi definido
7 #ifndef Sonar_H
8
9 //compilando ou não parte do código fonte. Se verdadeiro, define e
     compila todo código fonte
10 #define Sonar_H
11
12 // Diretiva do pré-processador com estrutura condicional
13 #if defined(ARDUINO) && ARDUINO >= 100
14     // Se o Hardware Arduino estiver conectado será utilizado a
        biblioteca "Arduino"
```

```
15    #include "Arduino.h"
16 #else
17      // Caso contrario sera utilizada a biblioteca "Wiring".
18    #include "WProgram.h"
19 #endif
20
21 //arquivo de cabeçalho que faz parte do C standard library e API(known
       as C99).
22 #include <inttypes.h>
23
24 //valores constantes, dados gravados em memória
25 #define PULSE_TIMEOUT 150000L
26 #define DEFAULT_DELAY 10
27 #define DEFAULT_PINGS 5
28
29 class Sonar {   // Classe do sonar HC-SR04;
30
31 public: //visibilidade: visível em qualquer classe
32
33    /*Construtor
34     * Sensor de ultra-SR04, quatro pinos de conexões
35     * VCC, ECHO, TRIGGER, GND *
36     * Param echoPin entrada digital Pino para medir a distância
37     * Param triggerPin um sinal alto de 10uS enviado pelo
           microcontrolador
38     * para o transmissor do SR04
39     **/
40    Sonar(int echoPin, int triggerPin);
41
42    /** Faz uma medição para este sensor. Retorna a distancia em
          centímetros */
43    long distancia();
44
45    /** calcula média.
46     * Param atraso de Espera, padrão = DEFAULT_DELAY / ms
47     * Param Contagem de numero de medições e Padrão DEFAULT_PINGS
48     * Retorna distancia em centímetros */
49    long distanciaMedia(int wait=DEFAULT_DELAY, int count=DEFAULT_PINGS)
          ;
50
51    /**Este metodo sonarPing recebe o retorno de "distancia()" */
52    void sonarPing() ;
53
54    /**retorna a última distância  do metodo "sonarPing ()"*/
55    long getDistancia();
56
57 private: //visibilidade: visível somente dentro da classe
58
59    /**Faz o cálculo de milisegundos para centimetros
60     * recebe como parametro o tempo em milisegundos do sonar
61     * o calculo leva em consideracao Velocidade do som 340m/sec (esse
           valor varia conforme a temperatura do ambiente)
62     * A duração em milisegundos / 5882 * 100 = distância em cm */
63    long microsegundosPCentimetros(long duracao);
64
65    // propriadades da classe Sonar
66    int _echoPin, _triggerPin; //pinos do uC
67    long _duracao, _distancia; //variaveis de armazenamento e controle;
```

```
68      bool _autoMode;
69      long _atualDistancia;
70  };
71  #endif
```

4.3 Robô Móvel Não-Holonômico com Acionamento Diferencial

A morfologia escolhida para o robô móvel educacional EduBOT é do tipo uniciclo (SIEGWART; NOURBAKHSH, 2004). Os modelos cinemáticos e dinâmicos serão apresentados parcialmente nas subseções posteriores. No que se refere ao projeto do robô móvel, a proposta faz uso de ferramentas *open-source* para diminuir o tempo e os custos de projeto. O *software* do sistema do robô móvel possui componentes implementados em C++ e estão inseridos na biblioteca do *framework* SOA. A Figura 32 apresenta a arquitetura de blocos do robô móvel, novos blocos podem ser adicionados ou removidos conforme as necessidades dos requisitos do projeto. O robô móvel EduBOT vem sendo validado em diferentes cursos de Engenharia da Universidade Federal do Rio Grande do Sul (UFRGS): Engenharia Elétrica, Engenharia da Computação e Engenharia de Controle e Automação.

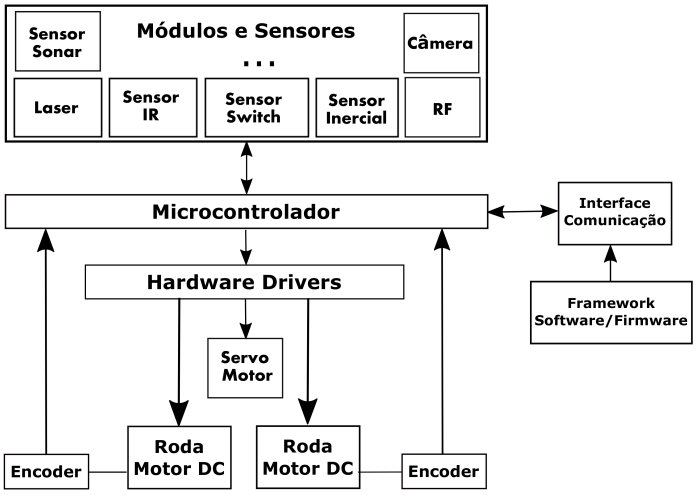


Figura 32: Diagrama de blocos para o robô móvel uniciclo. Fonte: Autor.

O projeto mecânico do projeto foi atualizado recentemente para a versão EduBOT-v0.3. Na Figura 33 é apresentado o desenho parcial em 2D e 3D do protótipo do robô móvel uniciclo. A plataforma de suporte ou chassi é a base para a fixação dos componentes de *hardware* que constituem o robô móvel. O desenho do chassi da base do robô móvel é mapeado para os elementos mecânicos e eletromecânicos como: motores, sensores, rodas, baterias, parafusos e demais componentes de *hardware* que são projetados e desenhados em *software* CAD de maneira modular e bem organizada (SOLIDWORKS, 2014). Para automatizar a fase de projeto, alguns dos componentes de *hardware* CAD (mecânicos e eletromecânicos) são projetados e alguns componentes reutilizados, podendo-se tornar um projeto simples ou complexo dependendo dos requisitos necessários para a aplicação.

Existe uma grande variedade de componentes eletromecânicos que podem ser escolhidos para o protótipo. A Figura 34 apresenta o desenho 3D com mapeamento parcial dos componentes do *hardware* para o projeto do EduBOT-v0.3.

Os sensores são dispositivos essenciais em um sistema de controle para robótica. Eles

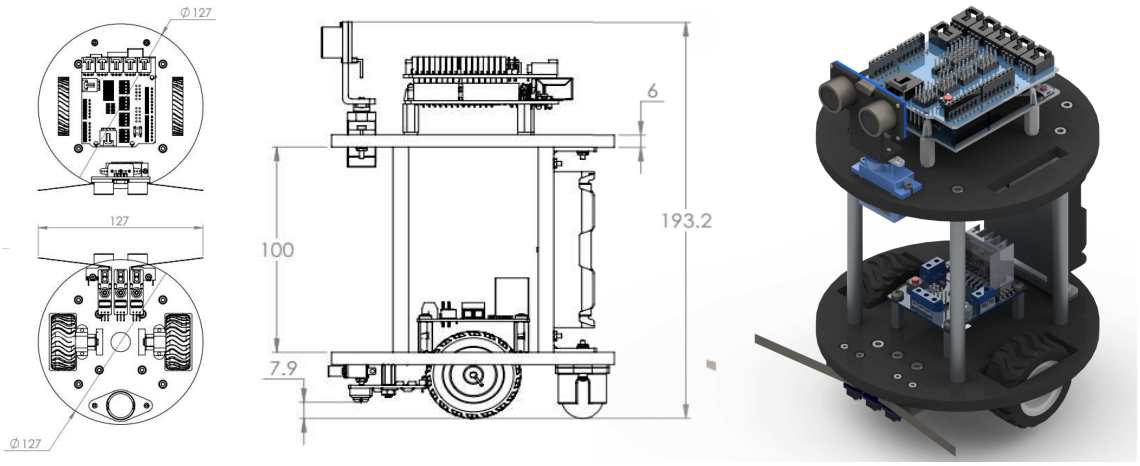
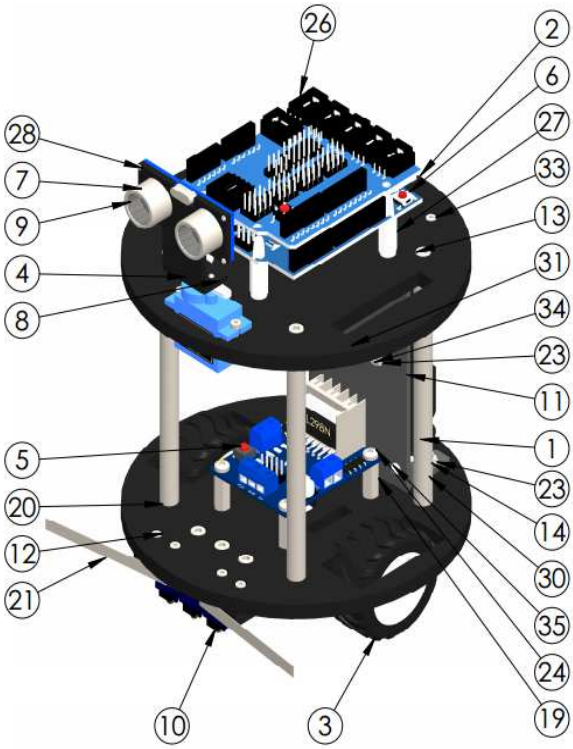


Figura 33: Projeto EduBOT-v0.3. Fonte: Eduardo Henrique Maciel e Autor.



Item	Componentes de *Hardware*	Quant.
1	Chassi 01	1
2	Chassi 02	1
3	Rodas Pololu 42x19mm	2
4	Micro Servo 9G SG90	1
5	*Driver* Ponte-H L298	1
6	Arduino Uno	1
7	Base do Sonar	1
8	Ponteiras Servo 9G SG90	1
9	Sonar HC-SR04	1
10	Sensor Infravermelho	3
11	Suporte da bateria	1
12	Parafusos Allen M2x14mm	6
13	Parafusos Allen M3x10mm	11
14	Porca M2	16
15	Parafusos Allen M3x6mm	4
16	Parafusos Allen M3x8mm	4
17	Parafusos Allen M2x16mm	2
18	Extensor Fêmea Hex 10mm M3x6mm	3
19	Extensor Fêmea Hex M3x15mm	4
20	Extensor Fêmea Hex M3x100mm	4
21	Botão Micro *Switch*	2
22	Porca M3	3
23	Arruela M2	16
24	Parafuso *Cross-Slotted* M3x6mm	4
25	Arruela M3	3
26	Sensor Shield Versão 5.0	1
27	Extensor Superior *Lock CB Support*	4
28	Parafuso Cross-Slotted M1x4mm	6
29	Parafuso Cross-Slotted M1x7mm	1
30	Base em L para Bateria	4
31	Suporte *pack* para as baterias	1
32	Bateria UltraFire 18650	2
33	Parafusos Allen M2x10mm	2
34	Parafuso Cross-Slotted M2x6mm	2
35	Parafuso Cross-Slotted M2x8mm	4

Figura 34: Tabela com mapeamento parcial dos diferentes componentes de *hardware* do projeto EduBOT-v0.3. Fonte: Eduardo Henrique Maciel e Autor.

serão utilizados no robô móvel para dar sinais de referência sobre cada tarefa a ser executada. Isto significa que um sensor é um dispositivo que converte um estímulo físico ou uma variável de interesse (tal como uma distância de um objeto) em uma forma mais conveniente (em geral em um sinal elétrico), cujo propósito é medir o estímulo.

Nos sistemas de controle, um atuador é um dispositivo de *hardware* que converte um sinal de comando do controlador em uma mudança em um parâmetro físico. Um atuador é um transdutor, visto que transforma um tipo de quantidade física, como uma corrente elétrica, em outro tipo de quantidade física, como uma velocidade de rotação de um motor elétrico.

As duas rodas fixas de tração do robô móvel são acionadas por motores CC através da ponte-h que envia de acordo com o controle do microcontrolador corrente suficiente para produzir o torque que aciona o rotor. A magnitude do torque do rotor é de acordo com a corrente que circula através do enrolamento.

4.3.1 Modelo Cinemático e Dinâmico

Para se ter uma ideia clara do movimento de um robô móvel, é de absoluta necessidade conhecer o seu modelo, o qual pode ser dividido em duas partes: um modelo cinemático e um modelo dinâmico. O modelo cinemático corresponde às características geométricas e às restrições de movimentos do robô móvel, enquanto que o modelo dinâmico representa como o robô móvel responde às entradas de controle que produzem o seu movimento no decorrer do tempo, levando em conta a sua massa e o modelo dinâmico dos seus atuadores, por exemplo. (LAGES, 1998).

4.3.2 Modelo Cinemático do Robô Uniciclo

O robô móvel possui uma estrutura formada por duas rodas fixas convencionais, controladas de maneira independente, e por uma roda de apoio do tipo rodas esféricas (*castor wheel*), que caracteriza-se como uma roda passiva ou orientável não centrada. O modelo cinemático do robô móvel do tipo uniciclo está baseado nas seguintes considerações: o robô é constituído por um chassi rígido e rodas não-deformáveis, que se movem em um plano horizontal. Assume-se que o plano das rodas permanece na vertical durante a movimentação, com a rotação se dando em torno de um eixo horizontal cuja orientação é fixa ou variável com relação a um sistema de coordenadas associado ao corpo do robô, $\{X_c, Y_c, \theta_c\}$, descrito em relação ao sistema inercial $\{X_0, Y_0, \theta_0\}$, com θ sendo o ângulo que define a orientação do robô. A partir da Figura 35 pode-se definir o vetor ${}^0\xi_c$ que contém as coordenadas da pose do robô e a matriz ${}^0R_c$ que expressa a orientação do robô em relação ao sistema inercial $\{X_0, Y_0, \theta_0\}$, dessa forma, tem-se que a sua postura é dada pelo vetor (3x1).

$$ {}^0\xi_c = \begin{bmatrix} X_c \\ Y_c \\ \theta_c \end{bmatrix} \tag{1} $$

$$ {}^0R_c\xi_c = \begin{bmatrix} cos(\theta_c) & -sin(\theta_c) & 0 \\ sin(\theta_c) & cos(\theta_c) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2} $$

Ainda, a velocidade do robô descrita em relação ao sistema inercial pode ser dada por:

$$ {}^{0}\dot{\xi}_c = \begin{bmatrix} \dot{X}_c \\ \dot{Y}_c \\ \dot{\theta}_c \end{bmatrix} \tag{3} $$

e descrita no sistema $\{X_0, Y_0, \theta_0\}$ por ${}^{c}\dot{\xi}_c = {}^{c}R_0 \; {}^{0}\dot{\xi}_c$, com ${}^{c}R_0$ dado por:

$$ {}^{c}R_0 = {}^{0}R_c^{-1} = {}^{0}R_c^{T} = \begin{bmatrix} cos(\theta_c) & sin(\theta_c) & 0 \\ -sin(\theta_c) & cos(\theta_c) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4} $$

Considera-se que o contato entre as superfícies das rodas e do piso satisfaz as condições de rotação pura, isto é, uma velocidade constante de rotação das rodas equivale a uma velocidade constante de translação das mesmas.

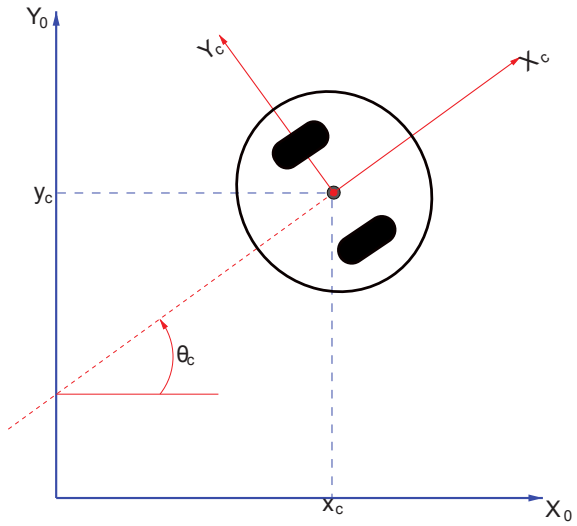


Figura 35: Esquema para obtenção do modelo cinemático do robô móvel uniciclo.

Além disso, em cada instante de tempo, o movimento do robô pode ser visto como uma rotação instantânea ao redor do seu Centro Instantâneo de Rotação (CIR), que no caso do robô móvel uniciclo encontra-se no eixo central que conecta as suas rodas, conforme mostra a Figura 36. Na figura, v_e e v_d representam, respectivamente, as velocidades das rodas esquerda e direita, ambas em relação ao solo, e R é o raio de curvatura instantâneo da trajetória do robô, isto é, a distância do CIR ao ponto central do eixo virtual que conecta as suas rodas.

Desse modo, se $v_e = v_d$, então o raio R é infinito e o robô move-se em linha reta. Por outro lado, se $v_e = -v_d$, então o raio R é zero e o robô gira ao redor do ponto central do eixo virtual que conecta as suas rodas. Para qualquer outro valor de v_e e v_d o robô move-se em linha curva, ao redor do seu CIR, o qual está situado a uma distância R do ponto central do eixo virtual que conecta as suas rodas, alterando tanto a sua posição quanto a sua orientação.

Ainda, podemos descrever o modelo cinemático de postura do robô uniciclo, em notação matricial, dado por

$$ \begin{bmatrix} \dot{X}_c \\ \dot{Y}_c \\ \dot{\theta}_c \end{bmatrix} = \begin{bmatrix} cos(\theta_c) & -sin(\theta_c) \\ sin(\theta_c) & cos(\theta_c) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ \omega \end{bmatrix}, \tag{5} $$

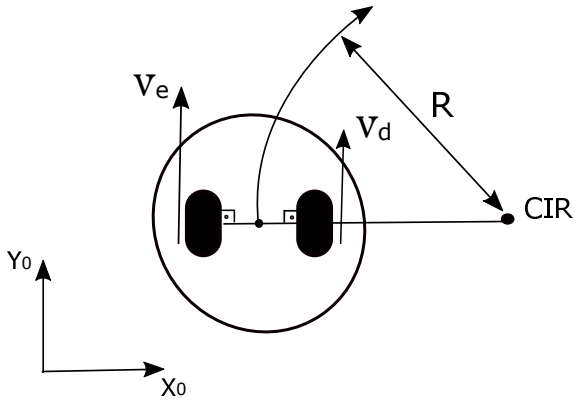


Figura 36: Representação do CIR do robô móvel uniciclo.

onde a velocidade linear u e a velocidade angular ω são as entradas de controle. O ponto central é fixo em relação ao eixo que conecta as rodas do robô uniciclo e pode ser determinado pelos eixos X_0 e Y_0 do sistema de coordenadas no plano cartesiano. Cabe mencionar, no entanto, que em robôs reais, sinais de entrada desse tipo não podem ser aplicados aos robôs diretamente, uma vez que o acionamento dos mesmos é feito por motores elétricos. Em virtude disso, as velocidades são aplicadas indiretamente, através de um controle de velocidade de baixo nível. Além disso, é possível considerar que as velocidades u e ω são desenvolvidas instantaneamente, desde que as mesmas sejam baixas e a inércia do robô seja pequena.

Por fim, tem-se que o robô móvel uniciclo é controlável. Isto significa que ele sempre pode ser conduzido de uma postura inicial ξ_0 a uma postura final ξ_f em um tempo finito, por meio da manipulação dos seus sinais de entrada (LAGES, 1998).

4.3.3 Modelo Dinâmico do Robô Uniciclo

Conforme mencionado na subseção anterior, pode-se supor que as velocidades desenvolvidas pelo robô rastreiam perfeitamente as entradas de controle, sob condição de que as velocidades e a inércia do robô sejam baixas. Sendo assim, essa consideração não é válida para muitas aplicações importantes, onde, por exemplo, os robôs precisam desenvolver velocidades altas e/ou transportar cargas pesadas. Nesses casos, é essencial considerar a dinâmica do robô, pois ela passa a ter grande influência na realização dos seus movimentos.

O modelo dinâmico do robô pode ser obtido de duas formas. Uma delas é através da modelagem analítica, também chamada de modelagem fenomenológica, que usa equações diferenciais e/ou algébricas para descrever os fenômenos físicos que ocorrem no sistema. No entanto, esse procedimento nem sempre é trivial, principalmente quando o sistema é muito complexo.

A outra forma é através da modelagem empírica, que usa dados de entrada e saída coletados do sistema para obter um modelo matemático aproximado. Nesse sentido, os diversos modelos dinâmicos de robôs uniciclos disponíveis na literatura diferem, essencialmente, pelo método através do qual foram obtidos e pelo tipo de suas variáveis de entrada, na maior parte das vezes tensões ou torques. No entanto, assim como os robôs da linha *Pioneer* (MOBILEROBOTS, 2014), a maioria dos robôs comerciais geralmente recebem comandos de controle do tipo velocidade, que são aplicados indiretamente através de um controle de velocidade de baixo nível. Para a futura modelagem dinâmica do projeto do estudo de caso em questão, serão seguidos os trabalhos de Lages (1998) e Barros

(2014), que apresentam um modelo dinâmico que tem a vantagem de possuir velocidades linear e angular como variáveis de entrada do modelo do robô uniciclo, utilizando-se dos parâmetros físicos do projeto. Agrupando essas informações obtém-se o chamado modelo dinâmico completo, Figura 37.

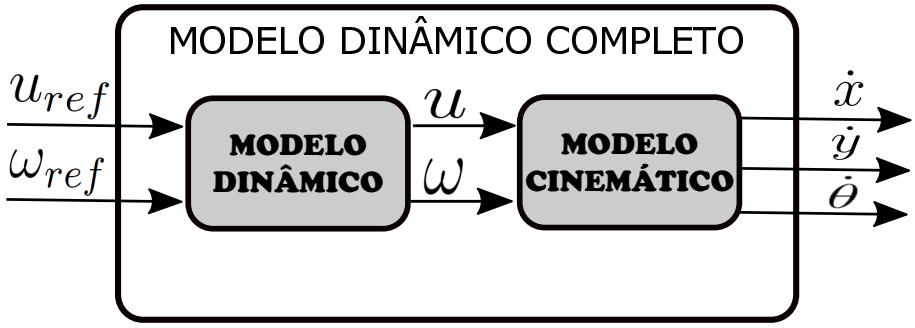


Figura 37: Diagrama de blocos do modelo dinâmico completo do robô móvel uniciclo.

Para movimentar-se em um ambiente e, por conseguinte, ser caracterizado como móvel, o robô deve ter seu movimento controlado. Assim ele poderá seguir trajetórias ou mover-se com exatidão para um ponto determinado. Para tanto, deve-se projetar um controlador que, agindo sobre os atuadores ajuste uma ou mais variáveis de estado do sistema para um valor desejado. O conhecimento do modelo matemático da planta é importante para a seleção da técnica de projeto de controlador a ser usada. O presente trabalho tem como objetivo desenvolver futuramente o projeto de um controlador de trajetória não-linear para o robô móvel uniciclo. Um modelo matemático dinâmico não-linear com múltiplas entradas e múltiplas saídas (ou MIMO, do inglês *Multiple Inputs, Multiple Outputs*).

4.3.4 Controle de Sistemas Não-Holonômicos

O controle de sistemas não-holonômicos pode ser agrupado como na teoria clássica de controle em: métodos de malha aberta e métodos de malha fechada. Os métodos de malha aberta são também conhecidos como planejamento do movimento para sistemas não-holonômicos e buscam leis de controle em malha aberta que desloquem o sistema de um estado inicial até um estado final. Já os métodos de malha fechada são aqueles que possuem alguma lei de realimentação para estabilizar o sistema em torno de um ponto de equilíbrio, seguir uma trajetória, ou rejeitar distúrbios (BAZANELLA; GOMES DA SILVA, 2005).

Tais métodos, em contraste com técnicas tradicionais, devem levar em consideração as restrições instantâneas ao movimento. As técnicas mais difundidas são as baseadas em:

- Geometria diferencial e álgebra diferencial;
- Fase geométrica;
- Parametrização da entrada;
- Controle ótimo do movimento;
- Planejamento do movimento evitando-se obstáculos.

Muitas vezes é necessária a utilização de controle por realimentação. Dependendo do tipo de resposta desejada existem diversas formulações para o problema de controle. Pode-se destacar três abordagens mais comuns: estabilização, rastreamento, e rejeição/atenuação de distúrbios (e várias combinações das mesmas). No caso da estabilização, procura-se leis de realimentação (variantes ou invariantes no tempo) que estabilizem um sistema para um determinado ponto de equilíbrio. Para o rastreamento, a meta básica é projetar um sinal de controle de forma que a saída controlada de y siga um sinal de referência y_R, isto é,

$$e(t) = y(t) - y_R(t) \approx 0, \forall_t \geq t_0 \tag{6}$$

onde t_0 é o instante em que o controle se inicia. Como o valor inicial de y depende do estado inicial (x_0), é necessário partir de um estado "pre-estabelecido".

Um postulado bem conhecido dos pesquisadores em controle não-linear diz que: Um sistema não-holonômico, embora seja completamente controlável, não pode ser estabilizado para uma configuração final de repouso através de leis suaves de realimentação nos estados. Este postulado se aplica a estabilização em torno de um ponto de equilíbrio e leis de controle invariantes no tempo, porém não é válido para o rastreamento de uma trajetória. Neste ultimo caso, pode-se usar as leis de controle conhecidas se o sistema for linearizável por realimentação estática ou dinâmica de estados, e uma vez que a trajetória de referência não contenha configurações de repouso (FIGUEIREDO; JOTA, 2004).

Para sistemas lineares invariantes no tempo, se os autovalores instáveis do sistema são controláveis, então a origem pode ser estabilizada de forma assintótica. Sistemas não-holonômicos apresentam características que o diferenciam consideravelmente do caso anterior. Mesmo que linearizado em torno de um ponto de operação, o sistema não é assintoticamente estável, logo, as estratégias tradicionais não podem ser empregadas. As principais técnicas de controle desenvolvidas para vencer esta limitação são:

- Estabilização variante no tempo;
- Estabilização usando sinais não contínuos ou transformações não-lineares;
- Leis híbridas de realimentação.

Restringe-se a abordagem aqui a exemplos da utilização do método de rastreamento de trajetória por linearização dinâmica, estabilização variante no tempo e não contínua em um ponto de equilíbrio e uma lei híbrida (KUHNE, 2005).

4.3.5 Projeto de Controle

Para o controlador do robô móvel esta sendo seguido o modelo utilizado em Garro (2013). O sistema do robô móvel é composto por vários subsistemas. Enfatizada, especificamente, no desenvolvimento de padrões de projeto para o subsistema de movimento.

A modelagem do robô móvel do estudo de caso é representado pelo diagrama de implantação, o qual descreve os nós do processador e dispositivos (sensores, atuadores e módulos). Os dispositivos sensores, por exemplo, são caracterizados por aprender e responder a estímulos internos e externos. O projeto vem sendo modelado de modo a obter três tipos de percepções no protótipo:

- Proprioceptivos, fornecem informações internas relativas ao sistema robótico, por exemplo, velocidade das rodas, posição dos motores CC e nível da bateria.

- Exteroceptivos, adquirem informações sobre a área de atuação do robô, por exemplo, medidas de distância e luminosidade do ambiente.
- Exproprioceptivos, fornecem informação sobre o corpo do robô ou partes dele relativa ao ambiente, por exemplo, posição no ambiente.

Um diagrama de implantação pode conter diferentes processadores e dispositivos interligados entre si, esses nós podem ser mantidos, excluídos ou adicionados no diagrama de implantação do projeto. Inicialmente a configuração do diagrama de implantação para o estudo de caso é representado na Figura 38, o qual apresenta a modelagem parcial do projeto, desenvolvido conforme as necessidades do desenvolvedor e da disponibilidade dos dispositivos e ferramentas para experimentar diferentes estrategias para navegação e controle. Não é objetivo deste trabalho detalhar a construção de um robô, portanto serão feitas aqui apenas algumas considerações sobre algumas possíveis implementações para o controle do projeto.

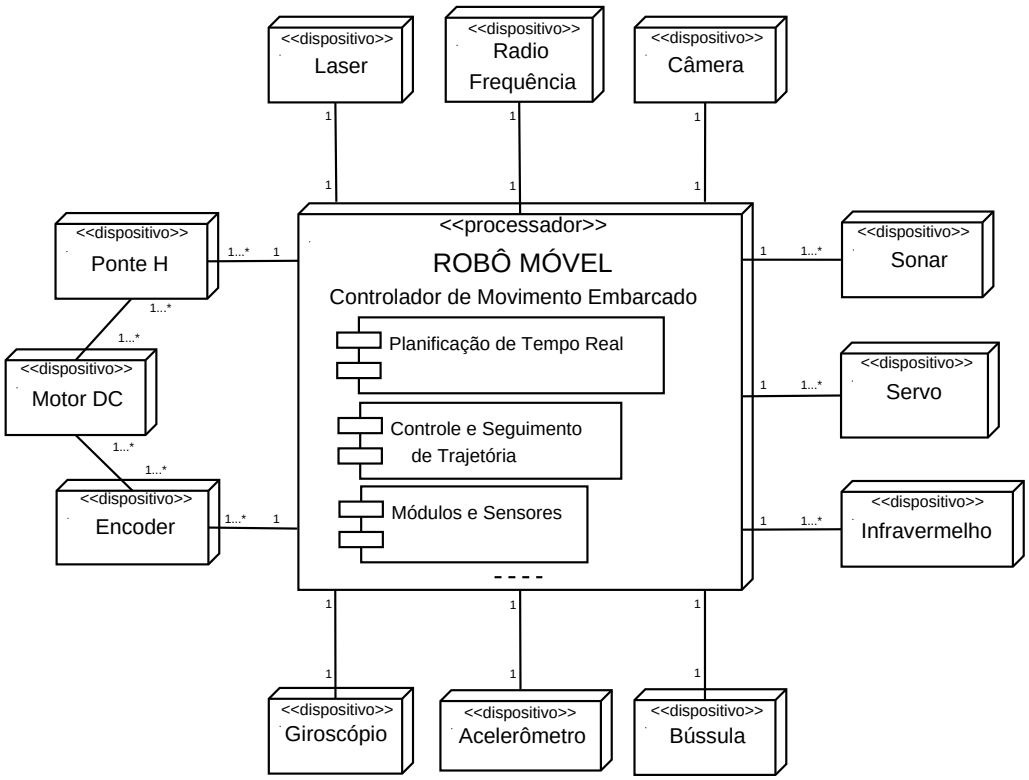


Figura 38: Modelagem do diagrama de implantação parcial para o subsistema de movimento do robô. O diagrama de implantação é composto por processadores, componentes e dispositivos, alocados conforme as necessidades do projeto.

Uma das tarefas mais importantes que o subsistema realiza, é o de examinar os dados com a frequência definida pelo planificador de tempo real. Os dados são enviados pelos diferentes módulos e sensores para o controlador central, que realiza os cálculos de trajetória. Com os dados retornados, o algoritmo de controle e rastreamento de trajetória é executado. O rastreamento de trajetória é obtido ao atuar com um sinal de controle na ponte-h, indicando a intensidade e direção de rotação de cada motor individualmente.

Na Figura 38, o subsistema de movimento ocorre com vários módulos e sensores. O objetivo é que estes dados permitam ao robô realizar um percurso de uma maneira controlada para se obter uma navegação eficiente.

Será utilizado como modelo para apresentar a estrutura de padrões de projeto, a proposta apresentado por Pont (2001).

CONTEXTO: Verificar o caminho de um robô móvel com acionamento diferencial em um ambiente.

PROBLEMA: Devido à forma direção diferencial escolhida para o robô, uma das principais dificuldades apresentadas por esta configuração, é o de assegurar que ambas as rodas movam-se de uma maneira controlada para assegurar que o deslocamento seja retilíneo ou seguindo dada uma curva. O objetivo da reutilização deste padrão é responder a este problema específico.

ANTECEDENTES: A figura 29 da subseção 4.3.2 mostra um diagrama a partir do qual o modelo cinemático do robô diferencial é gerado. Um sistema de coordenadas polares é utilizado, permitindo simplificar expressões matemáticas. São v_e e v_d as velocidades tangenciais da roda esquerda e à direita; e ω_E e ω_D suas velocidades angulares. Assim, é possível definir a velocidade linear do robô como:

$$V = \dot{R} = \frac{v_d + v_e}{2} = \frac{\omega_d + \omega_e}{2} r \tag{7}$$

Em que r é o raio da roda, V é a velocidade linear; e R é a distância radial do robô a partir da posição inicial. A velocidade de rotação do robô, será proporcional à diferença nas velocidades angulares das rodas. Expresso como:

$$\dot{\theta} = \frac{(\omega_d - \omega_e) r}{l} \tag{8}$$

Onde l é a distância entre as rodas. Do exposto anteriormente, pode-se ver que, se a velocidade angular da roda direita ω_d e esquerda ω_e é a mesma, o robô segue um caminho linear. Se houver uma diferença, é apresentado um desvio para um dos lados em proporção com a diferença das velocidades angulares.

Para a medição da velocidade angular, estão sendo pesquisados e utilizados diferentes sensores: giroscópio, codificadores e bússola eletrônica. Aqui estão três possíveis soluções a serem submetidas para o controle, onde cada uma é apresentada a partir de três componentes conceituais: a) *Hardware*, b) Controle, c) *Software*.

SOLUÇÃO 1 - Robô móvel com giroscópio:

Aqui é apresentada a solução mais simples de implementar, dado que se tenha sido usado um giroscópio analógico como sensor. É possível garantir que o robô móvel diferencial se mova ao longo de um caminho reto, porém com apenas um sensor deste tipo, não é possível determinar a distância radial do robô móvel em relação à posição inicial. Para isso, é necessário ter uma outra variável a ser medida. A solução 3 apresentada posteriormente, resolve o problema, incluindo um subsistema com bússola digital.

a) *Hardware*: Na Figura 39, se observa o esquema do *hardware* do subsistem a de movimento em que se destacam três componentes principais: microprocessador, giroscópio e *driver* de potência (ponte-h). Para as medições de velocidades angulares, um giroscópio analógico é utilizado, o qual é amostrado periodicamente, utilizando um conversor A/D do microcontrolador. Com o sinal obtido, o microcontrolador executa os cálculos implementados no controlador Proporcional, Integral e Derivativo (PID) para determinar

o sinal de Pulse-Width Modulation (PWM) necessário, o qual é enviado para o *driver* de potência. A modulação por Largura de Pulso ou PWM é a modulação ou alteração da largura do pulso de um sinal de onda quadrada que pode ser dados à ser transmitido. O estágio de potência é responsável pela adaptação às tensões nos terminais de cada um dos motores com base no sinal de controle recebido.

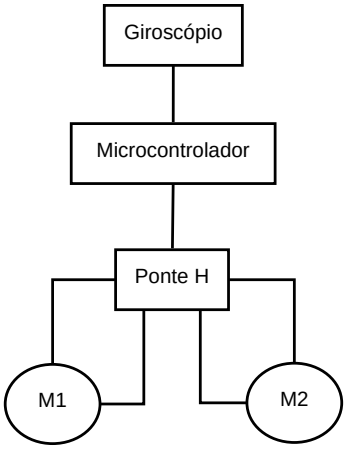


Figura 39: Subsistema de movimento com giroscópio.

b) Controle: Seguindo o modelo cinemático da figura 29 da subseção 4.3.2, é possível estimar os valores de ω_d e ω_e a partir da medição da velocidade angular $\dot{\theta}$ do giroscópio. Na Figura 40, observa-se um diagrama de um controlador PID, que toma como referência θ_{ref} uma tensão predeterminada. Com base nesta referência e a diferença com o sinal obtido a partir da detecção do giroscópio, se obtêm o erro, o qual irá convergir para o valor desejado, neste caso, zero. Quando o erro é zero, significa que ω_e e ω_d são iguais e, por consequência, a trajetória do robô é retilínea.

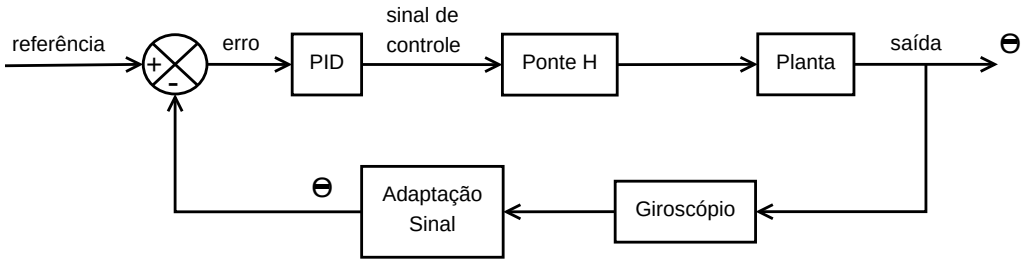


Figura 40: Controle PID.

Esta convergência, é obtida modificando as velocidades angulares individuais dos motores através da ponte-h, utilizando para tal um sinal de PWM correspondente.

c) *Software*: Na Figura 41, é apresentado o diagrama de estados para ser implementado no microcontrolador. Pode ser observado que, uma vez inicializadas as variáveis próprias do microcontrolador, a detecção do giroscópio é realizada utilizando o canal A/D do microcontrolador, e o sinal é condicionado através de um filtro *lowpass* digital. Uma vez que o sinal tenha sido filtrado, é comparado com o sinal de referência do controlador PID, e o erro é calculado. Com este erro é gerado um sinal de controle PWM, que age nos motores que utilizam o *driver* ponte-h. Como já comentado, o objetivo do PID, é igualar as velocidades angulares das rodas para garantir a trajetória retilínea.

SOLUÇÃO 2 - Sistema implementado com *encoders*:

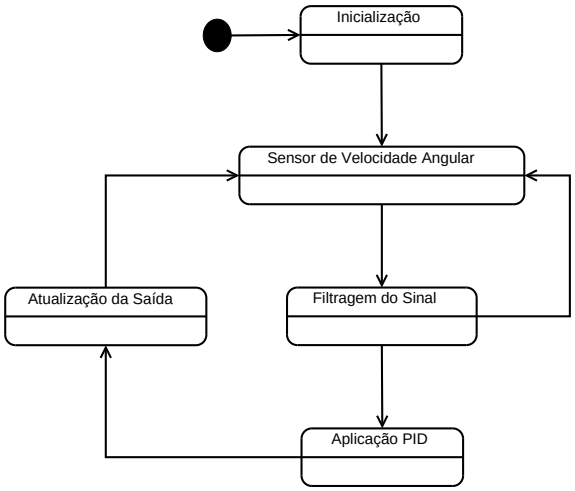


Figura 41: Diagrama de estados.

Esta solução utiliza dois *encoders* para determinar a velocidade angular, mas também a distância radial do robô em relação à posição inicial. Para isso, é necessário conhecer o modo como cada roda se move linearmente para cada impulso registado pelo *encoder*. Portanto, deve-se transformar o movimento angular do eixo em movimento linear da roda. A equação para calcular o deslocamento linear é:

$$C_m = \frac{2\pi r}{n * C_e} \tag{9}$$

Onde n é o eixo da roda de redução. No nosso caso, o *encoder* está integrado com as rodas de modo a que n=1; C_e é a resolução do *encoder*. Com esses dados, podemos calcular a velocidade angular de cada roda e o deslocamento linear.

a) *Hardware*: Na Figura 42, podemos observar o subsistema de movimento, que destaca três componentes principais: microcontrolador, *encoders* e o *driver* de potência. Os *encoders* presentes em cada uma das rodas, geram um impulso, o qual é detectado por interrupção de nível de *hardware*. Estes impulsos permitem medir a velocidade angular e o movimento linear, tal como discutido na Equação (9). Com o sinal obtido, o microcontrolador executa os cálculos necessários e define o sinal de PWM, o qual é enviado para o amplificador de potência. Esta etapa, é responsável pela adaptação das tensões dos terminais dos motores utilizando como *driver* a ponte-h.

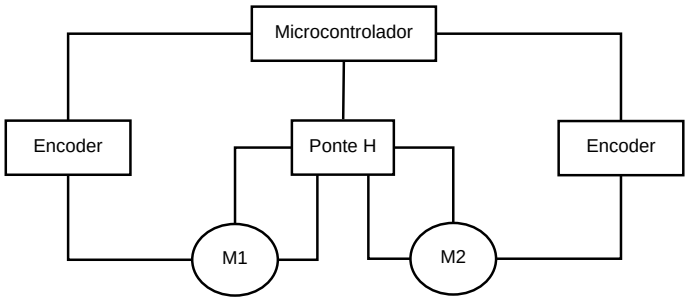


Figura 42: *Subsistema de movimento com encoders.*

b) Controle: A partir do modelo cinemático e conhecendo os valores de ω_d e ω_e obtidos a partir da medição dos *encoders*, a Figura 43 apresenta a solução usando um

controlador PID. São determinadas duas variáveis de referência (ângulo: θ_{ref}; distância radial R_{ref}). O objetivo do controlador é determinar a velocidade angular e distância de deslocamento angular do raio R. Por outras palavras, se a diferença da medição de velocidade angular determinada pela detecção dos *encoders* é igual a zero, o robô está se movendo de maneira retilínea. Caso contrário, realizará uma correção proporcional à diferença medida. Esta correção é feito pelo microcontrolador, que resulta no envio de um sinal PWM para ajustar o caminho pretendido e efetuar os cálculos necessários para determinar a distância radial do robô a partir da posição inicial.

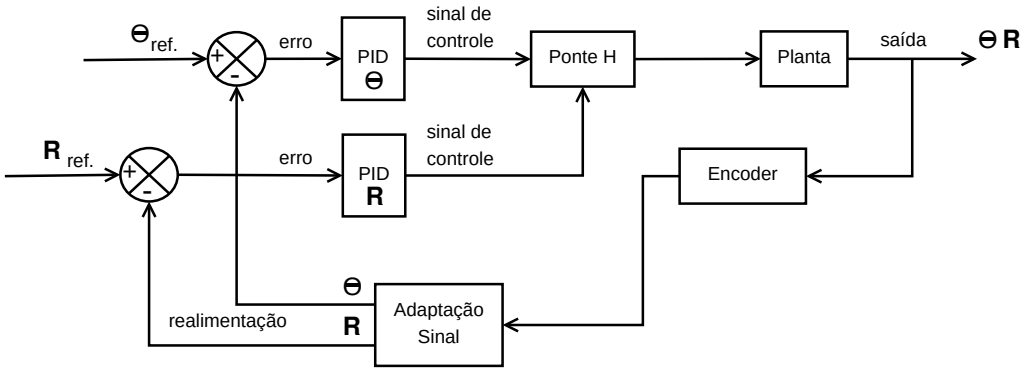


Figura 43: Controle PID

c) *Software*: A Figura 44, apresenta o diagrama de estados para ser implementado no microcontrolador. Uma vez inicializadas as variáveis próprias do microcontrolador e as do PID, podemos observar como se realiza a detecção dos impulsos gerados pelos dois *encoders* por meio de interrupção do *hardware*. O *software* calcula a velocidade angular e distância percorrida pelas rodas de acordo como descrito na Equação (9). Tomando os sinais de referência θ_{ref}, R_{ref} e a diferença com as variáveis detectadas, o erro é determinado. O erro irá convergir na velocidade angular, como na distância radial do robô a partir da posição inicial de acordo com os sinais de referência. A convergência, se dá pela ação do microcontrolador através da saída de controle PWM gerado, que atua sobre os motores utilizando o *driver* da ponte-h.

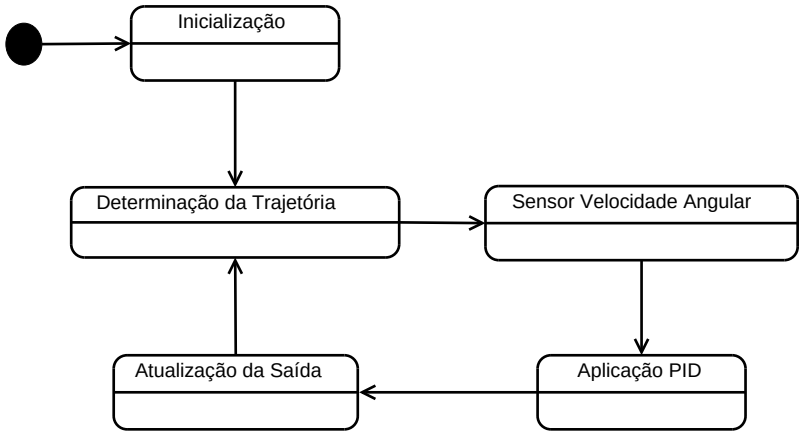


Figura 44: Diagrama de estado

SOLUÇÃO 3 - Sistema implementado com giroscópio e bússola:

Como no sistema proposto na solução anterior, com este desenho de *hardware*, *software* e de controle podemos determinar a velocidade angular e a distância radial do robô

em relação à posição inicial.

a) *Hardware* : A Figura 45, mostra o esquemático do *hardware* do subsistema de movimento os quais se destacam quatro componentes principais: microprocessador, giroscópio, bússola eletrônica e *driver* de potência.

A velocidade angular é detectada utilizando um giroscópio analógico que é periodicamente amostrado, usando um conversor A/D do microcontrolador. No entanto, esta medida não é suficiente para determinar a distância radial do robô com respeito à posição inicial. Portanto, é adicionada uma bússola digital, que entre outros parâmetros, transmite ao microcontrolador a posição angular média em graus norte magnético. Com os parâmetros recebidos, o microcontrolador executa os cálculos definidos nas variáveis de estados - as quais são apresentadas na seção seguinte -, e o sinal de PWM necessário, o qual é enviado para a etapa de potência. Esta etapa de potência é responsável por ajustar a tensão de cada um dos motores com base no sinal PWM recebido, um fato que se manifesta nas velocidades angulares ω_d e ω_e dos motores.

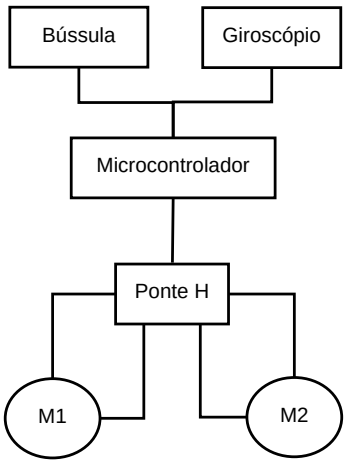


Figura 45: *Hardware* Subsistema de movimento com bussola e giroscópio

A velocidade angular é detectada utilizando um giroscópio analógico que é periodicamente amostrado usando um conversor A/D do microcontrolador.

b) Controle: Continuando conforme o modelo apresentado em Garro (2013), para realizar o controle do modelo cinemático do robô uniciclo com variáveis de estado e chegar em suas equações diferenciais (10), foi preciso desprezar a indutância interna do motor, em que k_1 e k_3 são constantes próprias do motor que relacionam a aceleração com a tensão (v_1 e v_2) aos terminais. E k_2 e k_4 representam os coeficientes de atrito das rodas.

$$\begin{cases} \omega_d = k_1 v_1 - k_2 \omega_d \\ \omega_i = k_3 v_2 - k_4 \omega_i \\ \dot{R} = \dfrac{\omega_d + \omega_i}{2} r \\ \dot{\theta} = \frac{(\omega_d - \omega_i) r}{l} \end{cases} \tag{10}$$

Logo, selecionando como variáveis de estado ω_d, ω_i e θ, e a v_1 e v_2 como as entradas do sistema, se obtêm a seguinte representação matricial de estados.

$$\dot{x} = Ax + bu, \tag{11}$$

Onde:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \omega_d \\ \omega_i \\ \theta \end{bmatrix}, A = \begin{bmatrix} -k_2 & 0 & 0 \\ 0 & -k_4 & 0 \\ \frac{r}{l} & -\frac{r}{l} & 0 \end{bmatrix}, b = \begin{bmatrix} k_1 & 0 \\ 0 & k_3 \\ 0 & 0 \end{bmatrix}, u = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}. \quad (12)$$

Conforme em Garro (2013), as constantes k_i se relacionam com as características eletromecânicas dos motores e também são afetadas por questões mecânicas como o sistema de transmissão ou o coeficiente de atrito entre as rodas ao deslocar-se sobre a superfície. O modelo de controle com as variáveis de estado são utilizados na implementação para o *software* do microcontrolador.

c) *Software*: Na Figura 46, se apresenta o diagrama de estado que deve ser implementado no microcontrolador. Se pode observar que, uma vez inicializada as variáveis características do microcontrolador, como as do PID, é feita a detecção da velocidade angular do giroscópio, assim como o desvio em graus norte magnético da bússola digital.

Com base nos dados obtidos a partir dos dois sensores, o microcontrolador realizará o cálculo da velocidade angular e de distância percorrida de cada uma das rodas. Finalmente, é gerado um sinal de controle PWM, que atua sobre os motores, utilizando como *driver* a ponte-h. Esta ação, possibilita que o sistema convirja na velocidade angular, e na distância radial do robô em relação à posição inicial desejada (GARRO; ORDINEZ; SCASSO, 2013).

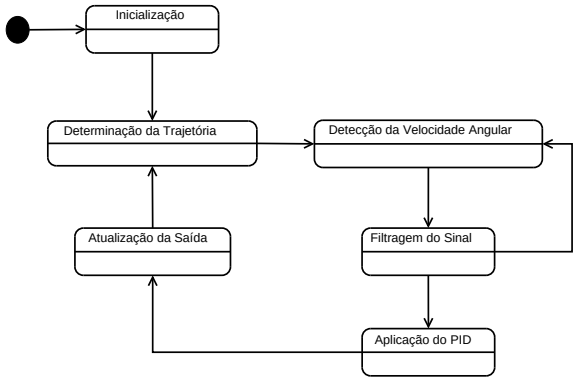


Figura 46: Diagrama de estados

4.4 Telemetria e Telecontrole de um Sistema Embarcado Aplicado na Macro e Micro Navegação de Deficientes Visuais

Visando as dificuldades existentes e as tecnologias disponíveis, o estudo de caso em questão tem por objetivo projetar um sistema nacional de rastreamento de bengalas eletrônicas. Aplicado na macro e micro navegação de deficientes visuais durante seus deslocamentos. A micro navegação se preocupa com as informações ocorridas durante a marcha ou deslocamento do deficiente visual dentro do ambiente que o circunda. Enquanto que a macro navegação é referente a posição geográfica e monitoramento remoto (GUIMARÃES; HENRIQUES; PEREIRA, 2013) do deficiente visual com sua bengala eletrônica. *Frameworks* SOA irão auxiliar no desenvolvimento do serviço de rastreamento para macro e micro navegação. O projeto é aplicado na orientação e mobilidade de deficientes visuais, durante o seu deslocamento em curtas e longas distâncias. E todo o acompanhamento da trajetória e controles de algumas funcionalidades devem estar acessíveis via *Internet*(GUIMARÃES; PEREIRA; HENRIQUES, 2014).

4.4.1 Tecnologia Assistiva

Tecnologia Assistiva (TA) é qualquer item, equipamento ou sistema, adquirido comercialmente, modificado ou customizado, que é usado para aumentar, manter ou melhorar capacidades funcionais de indivíduos com deficiência. A definição, apesar de bastante abrangente, é usada no contexto da mesma para referenciar ferramentas usadas para melhorar ou aumentar a funcionalidade das tecnologias baseadas em *software* ou *hardware*. Dentre os recursos utilizados pelos deficientes visuais para locomoção, a bengala branca apresenta-se como um dos mais baratos e seguros, isto é, quando manipulado corretamente. Mobilidade segura é um dos maiores desafios enfrentado pelos deficientes visuais em seu dia-a-dia. A bengala branca tradicional é da categoria auxilio a mobilidade na TA e é amplamente utilizado para sondar através da percepção tátil-cenestésica o espaço à frente, Figura 47.

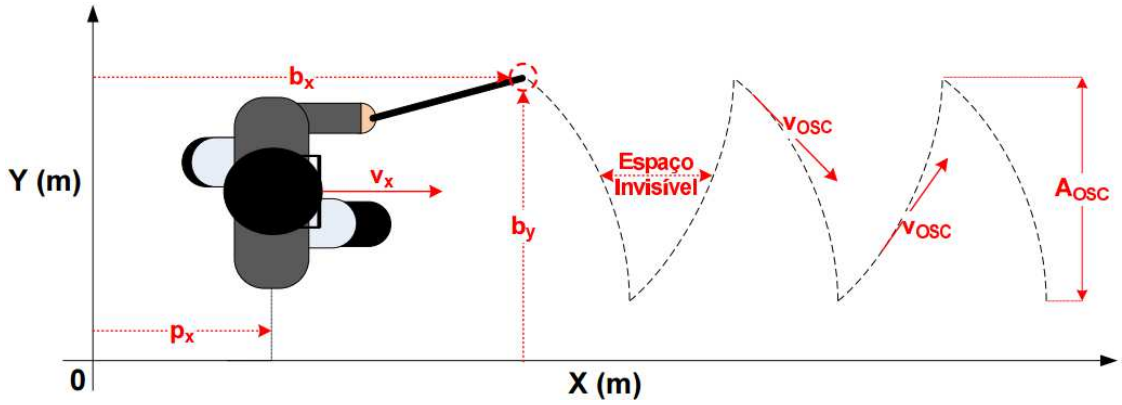


Figura 47: Ilustração do movimento característico de uma bengala branca, deixando por sondar a maioria do espaço na frente do utilizador. A vermelho estão representados os parâmetros medidos e calculados a partir dos ensaios realizados (ROSA, 2009).

Além disso, as bengalas brancas tradicionais são mecanismos úteis apenas em ajudar a detectar obstáculos dentro de um ambiente imediato que circunda o deficiente visual (micro navegação) e não pode prever obstáculos com antecedências localizadas acima ou abaixo da linha da cintura e nem informar a localização ou posição global (macro navegação). Devido ao avanço tecnológico grande parte das tarefas executadas por seres humanos estão sendo auxiliadas por computadores. Nesse pressuposto, sistemas computacionais embarcados estão presentes em diversas atividades, são sistemas dedicados que possuem uma funcionalidade restrita para atender uma tarefa específica em sistemas maiores nos quais estão inseridos. Estes sistemas são naturalmente heterogêneos, pois são constituídos de componentes de *hardware* e *software*.

4.4.2 O Movimento da Bengala

Em Regalado (2009), foi registrado os dados necessários para o cálculo dos parâmetros desejados para o cálculo da velocidade de marcha, v_x. Tendo sido as filmagens realizadas a 25fps (*frames per second*), tem-se que:

$$v_x = \frac{p_{xfinal} - p_{xinicial}}{\frac{frame_{final} - frame_{inicial}}{25}} \tag{13}$$

em que p_x representa a posição do deficiente visual no eixo X (Figura 47).

As medições de b_x e b_y referem-se à posição da ponta da bengala aquando do toque no chão. Com estas medidas é possível calcular a amplitude e velocidade de oscilação da bengala (A_{OSC} e v_{OSC}), respectivamente), bem como o espaço invisível para a mesma. A velocidade de oscilação usa o utilizador como referência e consiste na velocidade media, tangencial ao arco descrito pela bengala. Como o comprimento do arco descrito pela bengala é dado por:

$$a = 2.r.arcsin(A_{OSC}/2) \tag{14}$$

e se mediu que r (raio de rotação da bengala) é aproximadamente $1m$, tem-se que:

$$v_{OSC} = \frac{2.arcsin(A_{OSC}/2).(n^{\underline{o}} \text{ de toques} - 1)}{\frac{frame_{final} - frame_{inicial}}{25}} \tag{15}$$

sendo que $n^{\underline{o}}$ de toques representa o número de vezes que a bengala toca no chão, em ambos os lados.

Através dos resultados obtidos em Regalado (2009), chegou-se a uma conclusão importante, existe uma razão aproximadamente constante entre v_x e v_{OSC}. Isto deve-se ao fato de o ritmo da bengala ter que acompanhar o ritmo da passada, de forma a estar sempre no lado contrário ao do pé dianteiro. Quanto à amplitude de oscilação, verificou-se que existe uma maior tendência a alargar mais o movimento com o aumento da velocidade de marcha o que, prejudica bastante o espaço invisível. Neste caso, e apenas para efeitos de quantificação, considera-se que o espaço invisível é a distância que vai desde o ponto em que a ponta da bengala passa à frente do utilizador até à passagem seguinte, no sentido contrário (ilustrado na Figura 47). Assumindo que v_{OSC} é constante ao longo de todo o trajeto, tem-se que:

$$\text{Espaço Invisível } = \frac{b_{xfinal} - b_{xinicial}}{n^{\underline{o}} \text{ de toques} - 1} \tag{16}$$

Este parâmetro só foi calculado para o primeiro e terceiro ensaio realizado em Regalado (2009), pois nestes foi realizada uma filmagem adicional com uma câmara de lado, a acompanhar o deslocamento do cego, facilitando a medida de b_x (ROSA, 2009).

4.4.3 Bengala Eletrônica

O estudo de caso descreve parte das metodologias e ferramentas utilizadas para a análise e projeto parciais de uma bengala eletrônica: Protótipo de um sistema embarcado para auxílio à macro e micro navegação de deficientes visuais. O projeto em desenvolvimento é um sistema eletrônico de apoio a mobilidade para substituição da visão pelo som e tato, Figura 48. Existem muitas configurações que podem ser definidas para um projeto de uma bengala eletrônica. Portanto, para a proposta inicial desse estudo de caso, foram definidos dois modelos conceituais que serão apresentados a seguir.

O projeto vem sendo desenvolvido no PPGEE (Programa de Pós-Graduação em Engenharia Elétrica) e encontra-se como parte da validação do trabalho de dissertação.

A fim de distinguir os obstáculos acima e abaixo da cintura, as marcações $1a$ e $1b$ das figuras seguintes, indicam o uso de micromotores para informar através de vibrações o utilizador quando existem obstáculos. Para obstáculos abaixo da linha da cintura é acionado o micromotor referente a marcação $1a$, localizado no cabo ou pega da bengala eletrônica. Assim, quando um obstáculo é detectado acima da linha da cintura do utilizador, o microcontrolador transmite um sinal através de um módulo de Radiofrequência

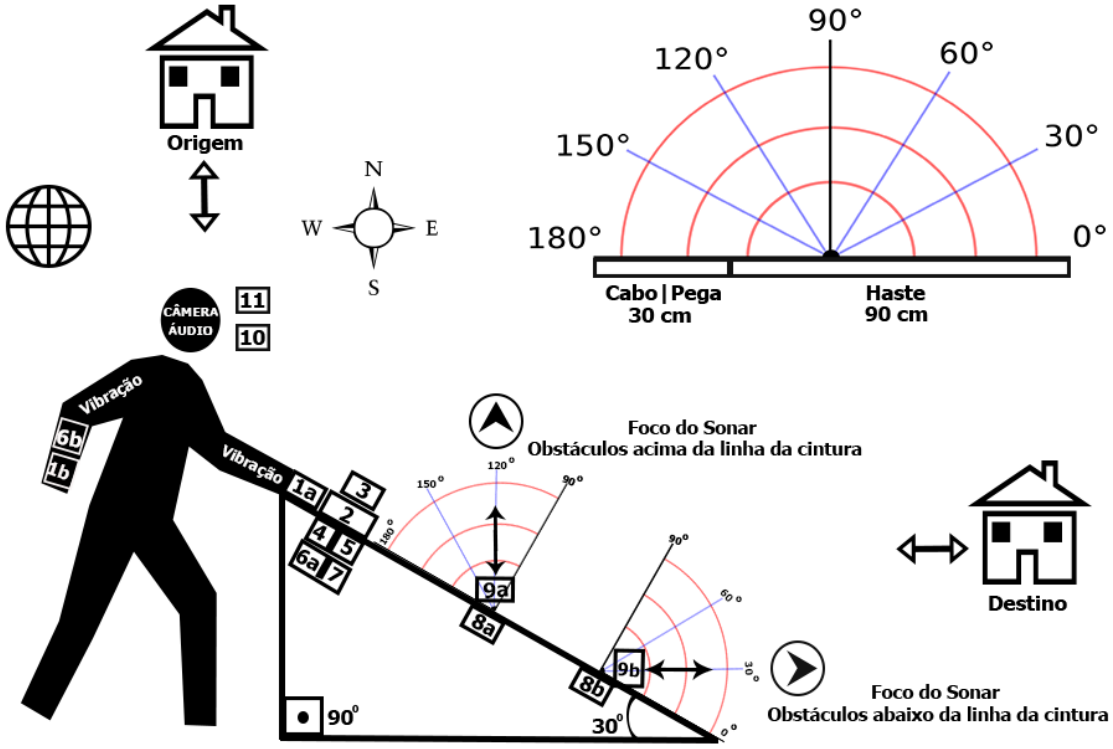


Figura 48: Modelo conceitual do projeto da bengala eletrônica

(RF). O receptor de RF faz operar o micromotor de vibração $1b$, avisando o usuário antecipadamente sobre a presença de obstáculos nessa região.

A Tabela 2, mostra a fusão dos componentes eletrônicos do sistema com as marcações apresentadas na primeira configuração do modelo conceitual do projeto.

Tabela 1: Relação dos componentes eletrônicos com as marcações da Figura 48.

Componentes	Marcações na Imagem										
	1a e 1b	2	3	4	5	6a e 6b	7	8a e 8b	9a e 9b	10	11
Micromotor de Vibração	X										
MCU ou SoC		X									
LDR			X								
GPS				X							
GSM					X						
Radiofrequência						X					
Sensor Inercial							X				
Servomotor								X			
Sonar									X		
Áudio										X	
Câmera											X

O protótipo do *hardware* eletrônico pode ser composto por microprocessadores, atuadores, sensores e módulos, como por exemplo: *MicroController Unit Board* (MCU) ou *System On Chip* (SOC), servomotores, micromotores de vibração, radiofrequência, *Global Positioning System* (GPS), *Global System for Mobile Communications* (GSM), *Light Dependent Resistor* (LDR) e sensores ultrassônicos (sonar) que devem estar inseridos na

haste da bengala eletrônica através de um modelo analítico trigonométrico, para calcular a distância e a altura de obstáculos, utilizando as relações geométricas de inclinação (ângulo) da bengala branca no solo e dos sonares na bengala, através dessas informações pode-se determinar o melhor foco do sensor ultrassônico na relação bengala-objeto. Para garantir o melhor foco dos sensores ultrassônicos em ângulos retos com os objetos, deve ser utilizado um sistema de controle em malha fechada, realimentado com sensores inerciais para garantir a melhor abrangência dos sensores ultrassônicos com os objetos. Servomotores precisam estar acoplados ao sensores ultrassônicos para ajustar a sua posição conforme o ângulo da bengala com o solo.

A seguir vemos a configuração para o segundo modelo conceitual do projeto, a nova proposta contém apenas o sistema embarcado referente a micro navegação na bengala eletrônica. Já para a macro navegação, o sistema embarcado deve ser dedicado e adaptado no corpo do deficiente visual, retirando assim, um pouco do peso da bengala eletrônica, a Figura 49 mostra o projeto do sistema.

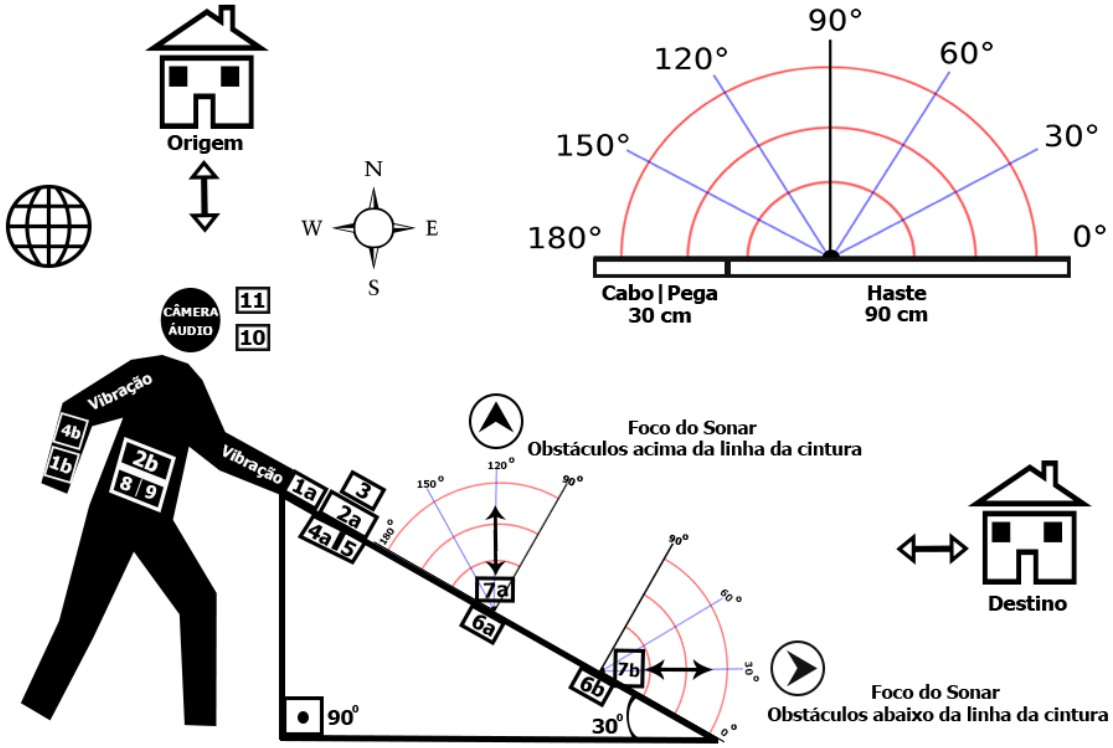


Figura 49: Modelo conceitual do projeto da bengala eletrônica

A Tabela 3, mostra a fusão dos componentes com as marcações apresentadas, agora para a segunda configuração do modelo conceitual do projeto.

Diferentemente da bengala branca tradicional, que acompanha os desníveis do piso, mas não pode prever objetos em diferentes posições e nem informar a localização global, o equipamento proposto possui sistema embarcado. Para uma melhor descrição do projeto foi desenvolvido um diagrama de blocos, Figura 50, que é um diagrama cujo objetivo é a representação gráfica do processo e modelo do sistema. Através de figuras geométricas e ligações, descrevem-se as relações entre cada subsistema e o fluxo de informação.

Os módulos e os sensores estimulam os atuadores indicando a geolocalização, nível de luz do ambiente e a evidência de objetos. A geolocalização e a intensidade de luz são informadas por áudio, já a existência de obstáculos é confirmada pela vibração gerada por atuadores vibratórios (micromotores), as intensidades de vibrações mudam conforme

Tabela 2: Relação dos componentes eletrônicos com as marcações da Figura 49.

Componentes	Marcações na Imagem										
	1a e 1b	2a e 2b	3	4a e 4b	5	6a e 6b	7a e 7b	8	9	10	11
Motor Vibração	X										
MCU ou SoC		X									
LDR			X								
Radiofrequência				X							
Sensor Inercial					X						
Servomotor						X					
Sonar							X				
GPS								X			
GSM									X		
Áudio										X	
Câmera											X

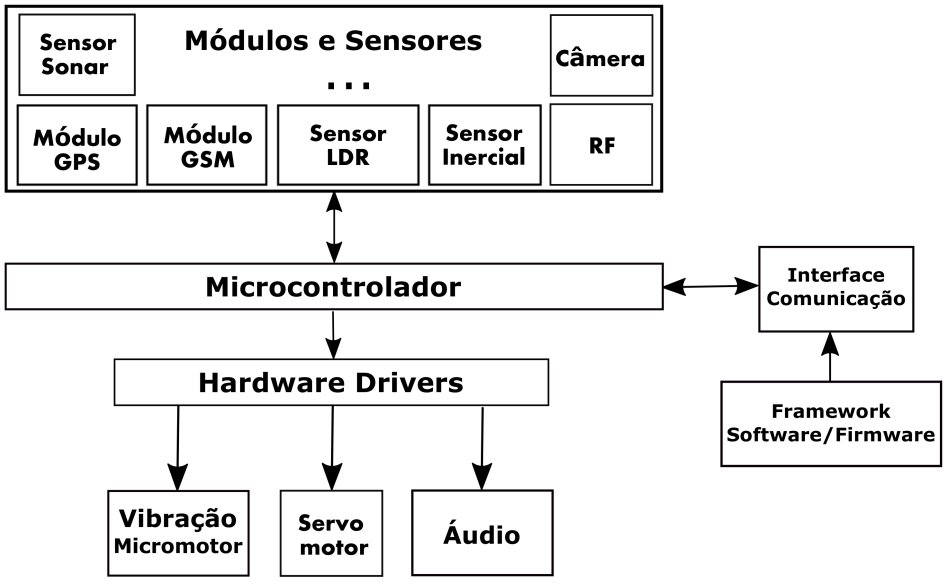


Figura 50: Diagrama de Blocos do projeto da Bengala Eletrônica.

mudam as distancias da bengala eletrônica com os obstáculos. Um micromotor deve ser inserido em um dispositivo de tato, localizado no cabo (pega) do protótipo e outro inserido no pulso (da mão contrária que segura a bengala) do deficiente visual, em forma de pulseira eletrônica. Assim o usuário do dispositivo pode tomar decisões conforme as informações recebidas dos micromotores de vibração e ganhar mais confiança no seu deslocamento, podendo detectar previamente possíveis obstáculos abaixo (vibrações no cabo) ou acima (vibrações na pulseira) da linha de cintura.

No entanto, os estágios do desenvolvimento de sistemas de tempo real são modulares e estruturados, como em qualquer projeto de robótica, ferramentas e metodologias estão disponíveis para auxiliar no desenvolvimento, dentre as quais podemos citar a especificação utilizando análise estruturada com metodologias SART (*Structured Analysis Real-Time*) e metodologias de projeto baseadas no paradigma de orientação a objetos com modelagem UML. Independente da metodologia escolhida ela deve lhe proporcionar a decomposição do projeto em partes de forma a possibilitar sua compreensão e administrar sua complexidade. Essas partes podem ser representadas como modelos que descrevem e abstraem aspectos essenciais do sistema.

4.4.4 Desenvolvimento do Sistema com Análise Estruturada

Assim como todas as metodologias de análise de requisitos de *software*, a análise estruturada é utilizada para ser uma atividade de desenvolvimento de modelos com fluxo de dados e conteúdo das informações divididas em partições funcionais, representando as características do que se deve ser desenvolvido. As primeiras metodologias de projeto voltadas para sistemas tempo real surgiram como extensão às técnicas de análise estruturada. Uma das mais importantes destas extensões foram as metodologias SA-RT (*Structured Analysis Real-Time*). A partir do método SA-RT, podemos descrever o problema a ser solucionada com uma decomposição funcional com uso de diagramas, listas, máquina de estado e dicionários de dados. Os requisitos do projeto são apresentados por Diagrama de Fluxo de Dados (DFD) que mostram como os dados são processados pelo sistema em termos de entradas e saídas. O modelo fundamental do sistema ou modelo de contexto (nível 0) representa o controle do sistema através de um processo principal e em sua volta entidades e fluxos de dados os quais possuem relação direta com o sistema do controle. A Figura 51 apresenta o diagrama de contexto do projeto.

O DFD de nível 1, representado na Figura 52, é particionado utilizando as extensões de Ward e Mellor (1986) para representar mais detalhes do sistema global. O primeiro nível do DFD mostra os processos principais do sistema. Os processos que compõe o sistema são subfunções do sistema global descrito no modelo de contexto e são definidos como tarefas que são executadas em paralelo. O processo de controle (bolha tracejada) denominado como "Controle Operações"executa todos os fluxos de dados recebidos dos demais processos de dados do sistema e conforme os fluxos de informação realiza o acionamento dos atuadores de vibração e áudio.

O diagrama de transição de estado (DTE), Figura 53, é o detalhamento do processo de controle "Controle Operações"e representa o comportamento do sistema mostrando os estados e os eventos que fazem o sistema mudar de estado. O DTE indica que ações são iniciadas como consequência de eventos. Um estado é qualquer modo de comportamento observável, através dele os fluxos de dados do sistema são sequenciados e inicializados de acordo com a solicitação.

A principal vantagem da análise estruturada é no desenvolvimento de um requisito de sistema. Ter um modelo de requisitos completo em um diagrama estruturado ajuda a

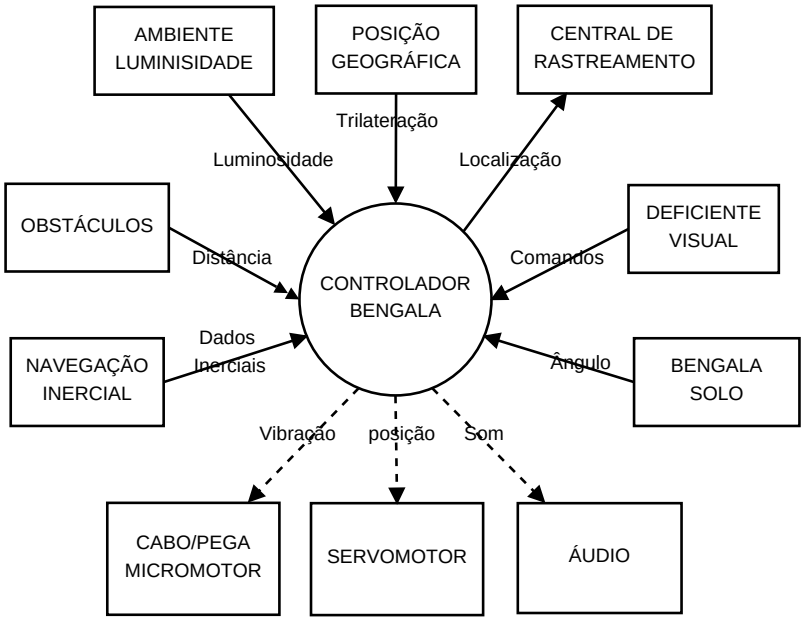


Figura 51: Diagrama de Contexto do Dispositivo Eletrônico. Fonte: Autor.

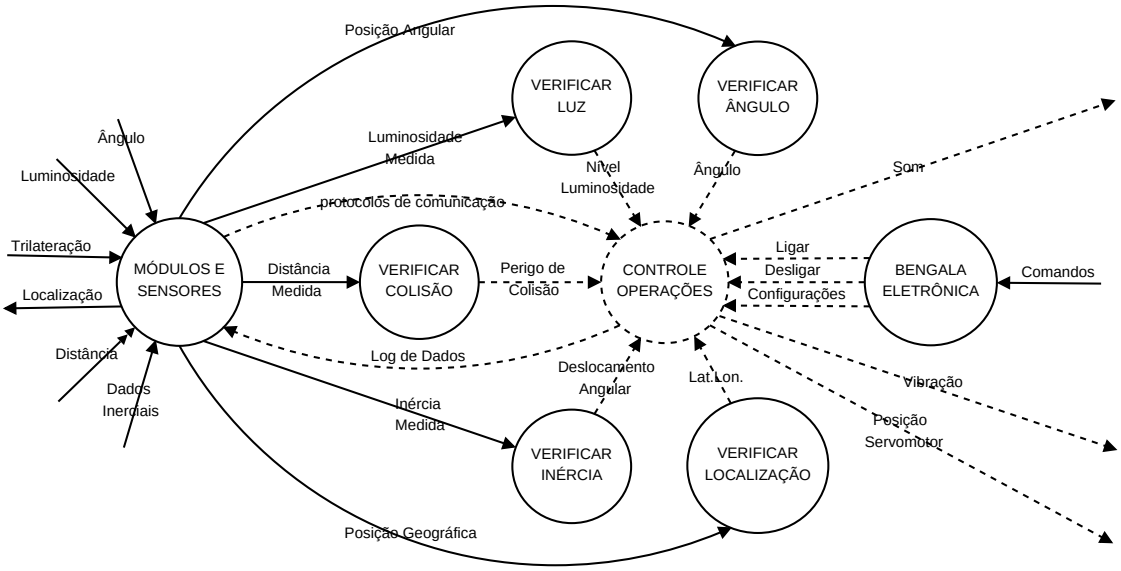


Figura 52: Extensão Representando Ward e Mellor - DFD nível 1

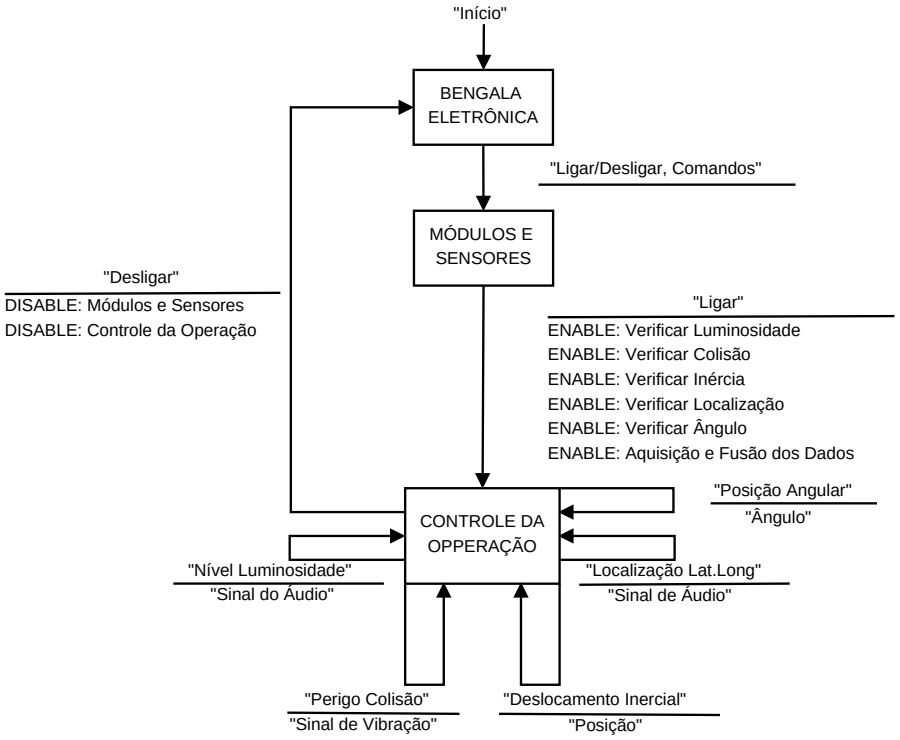


Figura 53: diagrama de transição de estado

assegurar que todos os requisitos são alocados aos componentes da arquitetura do sistema, que leva a apresentar uma situação do ponto de vista dos dados, ao invés de apresentá-lo do ponto de vista de qualquer pessoa ou empresa. Os usuários obtêm uma ideia mais clara do sistema proposto pelo diagrama de fluxo de dados, do que a obtida através da narrativa e fluxograma de sistemas físicos, a apresentação em termos de fluxo lógico consegue mostrar mal entendidos e pontos controversos.

Como desvantagem, podemos citar as dificuldades causadas por problemas de comunicação, dificuldade em descrever procedimentos, falta de metodologias apropriadas para ajudar na especificação dos sistemas, problemas de manutenção do documento de especificação, grau de detalhamento necessário, principalmente na construção do dicionário de dados e mudanças de requisitos tornam a análise estruturada uma fase crítica no desenvolvimento de sistemas (WARD; MELLOR, 1986).

4.4.5 Desenvolvimento do Sistema com Análise Orientada a Objetos

Nos últimos anos, as metodologias de projeto baseadas no paradigma de orientação a objetos têm sido apontadas como uma alternativa interessante para combater as deficiências apresentadas pelas técnicas de análise estruturada. Este paradigma apresenta diversas características que facilitam o entendimento do modelo, bem como permite um maior encapsulamento para os dados e facilita o reuso. Consequentemente, o paradigma de orientação a objetos também acabou sendo aplicado com sucesso no desenvolvimento de sistemas tempo real. A seguir são apresentados um conjunto de classes e seus relacionamentos, Figura 54.

A Linguagem de Modelagem Unificada (UML), é uma linguagem e não um método, a UML é uma linguagem padrão de notação de projetos. Por notação entende-se especificar, visualizar e documentar os elementos de um sistema orientados a objeto.

Um diagrama de classes na UML é um tipo de diagrama de estrutura estática que

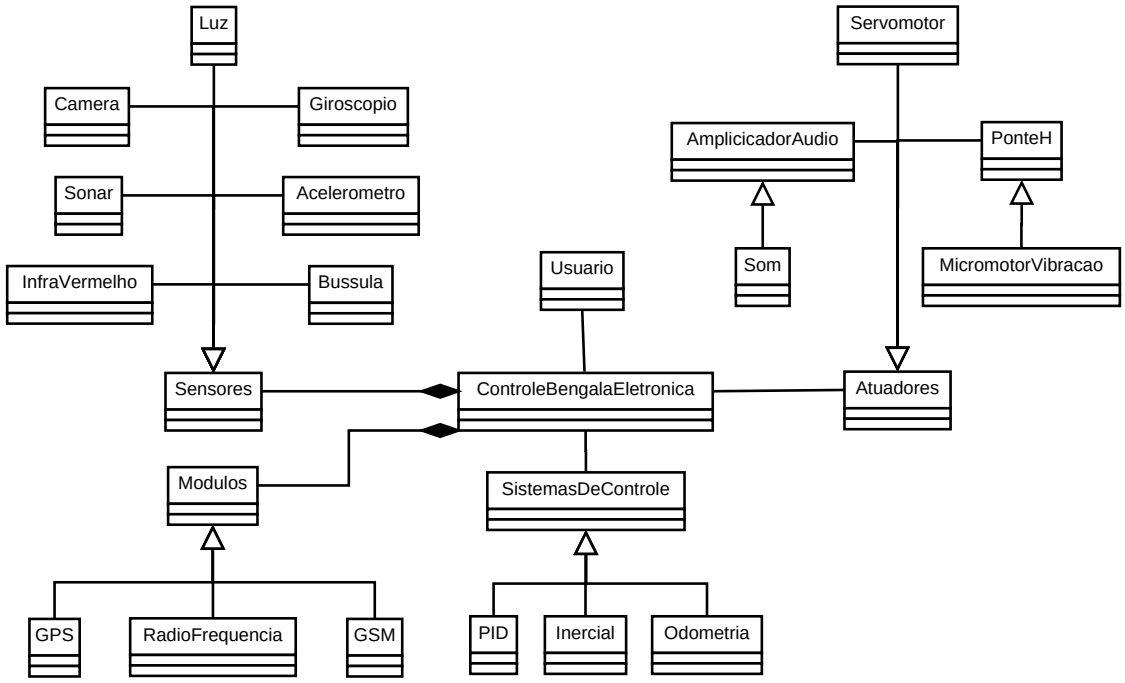


Figura 54: Diagramas de classes de análise do projeto. Fonte: Autor.

descreve a estrutura do sistema, mostrando as classes do projeto, seus atributos, operações, e as relações de mensagens entre as mesmas. Todas as classes devem fazer sentido no domínio da aplicação. Nem todas as classes surgem explicitamente na definição do problema; algumas estão implícitas no domínio da aplicação ou no conhecimento geral. Para testar alguns dos dispositivos do estudo de caso em questão, foram desenvolvidos diagramas de classes parciais, como mostra a Figura 55. O modelo de classes parciais do projeto é resultante de refinamentos no modelo de classes de análise.

O diagrama de casos de uso, descreve a funcionalidade proposta para o sistema, representa uma unidade discreta da interação entre um usuário (humano ou máquina) e o sistema, exercendo um papel importante na análise de sistemas, é o principal diagrama para ser usado no diálogo com o usuário na descoberta e validação de requisitos, constituem elementos que estruturam todas as etapas do processo de software. Em particular, o diagrama de casos de uso, não mostra a ordem na qual as etapas são executadas para atingir as metas de cada caso de uso. Você pode descrever os detalhes em outros diagramas e documentos, que é possível vincular a cada caso de uso. O projeto possui nove atores e suas respectivas interações com os casos de uso. Desta forma, o diagrama da Figura 56, está demostrando a funcionalidade do sistema pretendido.

O diagrama de sequência parcial do projeto, Figura 57, é uma espécie de diagrama de interação que mostra como os processos interagem uns com os outros e em que ordem. É uma construção de um gráfico de sequência de mensagens e interações de objetos dispostos em sequência temporal. Este diagrama é construído a partir do diagrama de casos de usos. Primeiro, define-se qual o papel do sistema (*Use Cases*), depois, é definido como o software realizará seu papel (Sequência de operações). O diagrama de sequência dá ênfase a ordenação temporal em que as mensagens são trocadas entre os objetos de um sistema. Entende-se por mensagens os serviços solicitados de um objeto a outro, e as respostas desenvolvidas para as solicitações.

A vantagem da análise orientada a objetos é que ela pode representar melhor o mundo real. A mesma é usada desde a análise até o projeto e a implementação, de modo que a in-

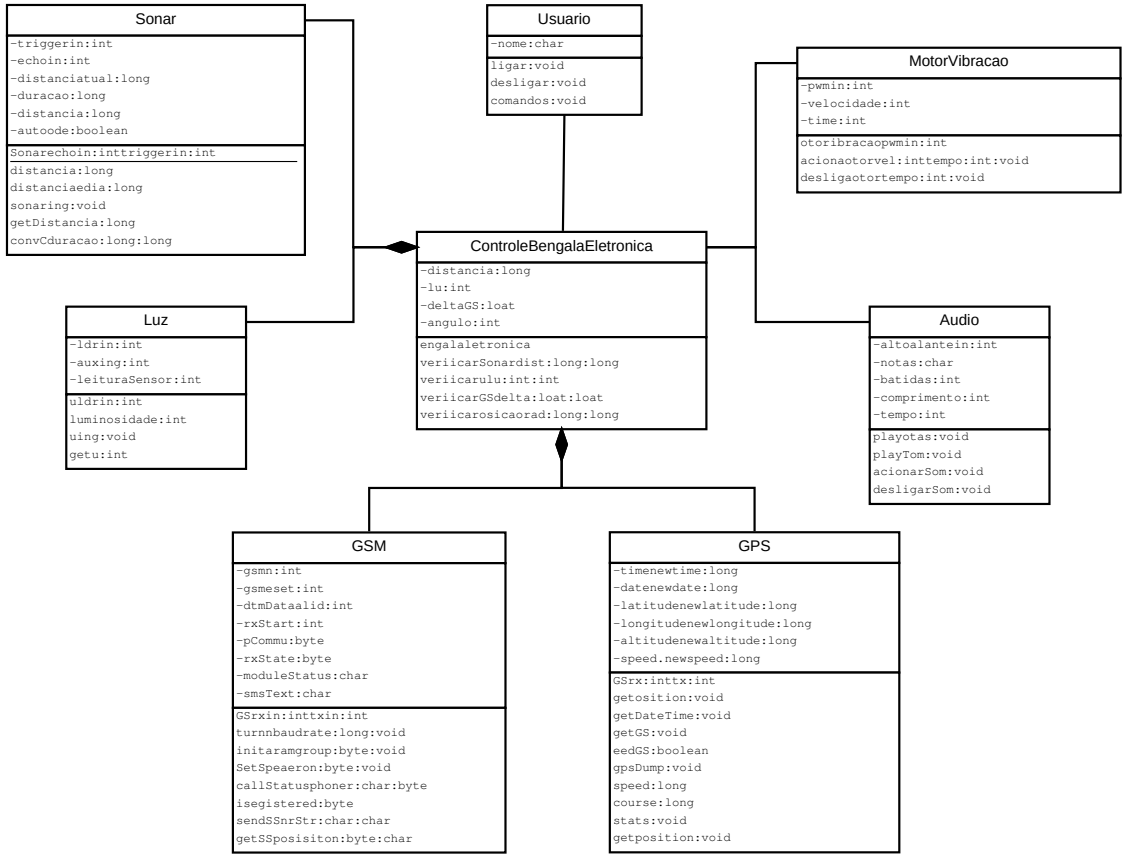


Figura 55: Diagrama de classes parciais do projeto, desenvolvido para testes e implementações. Fonte: Autor.

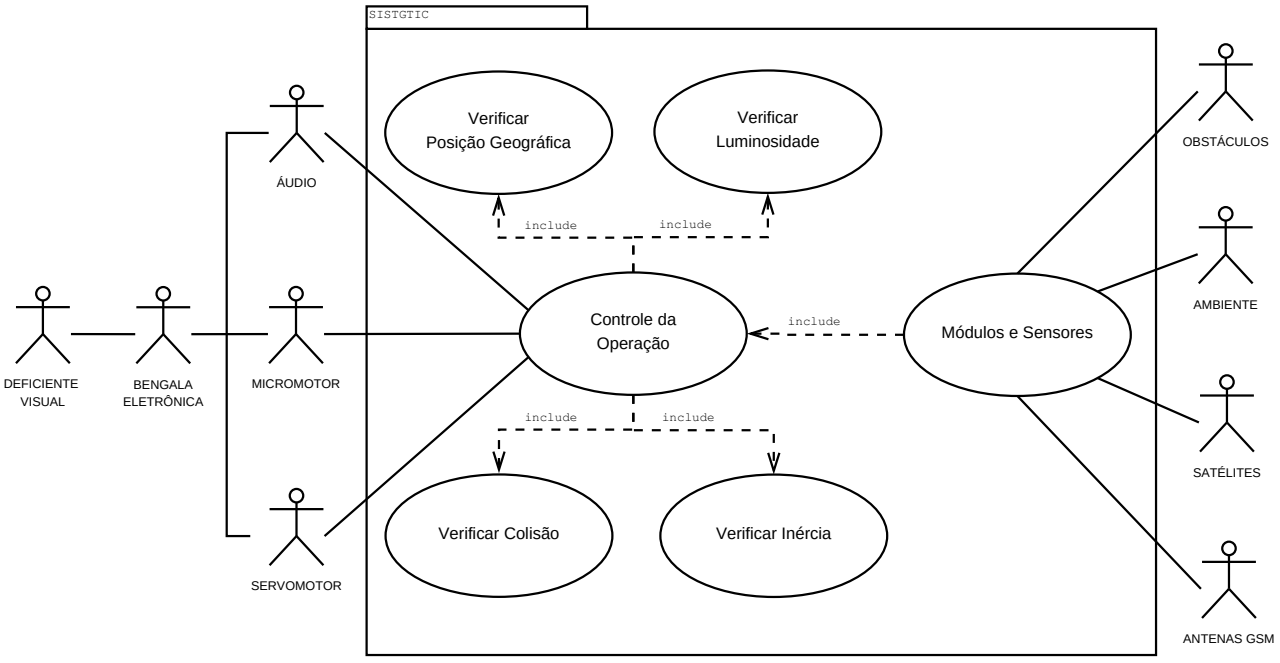


Figura 56: Diagrama de caso de uso do projeto

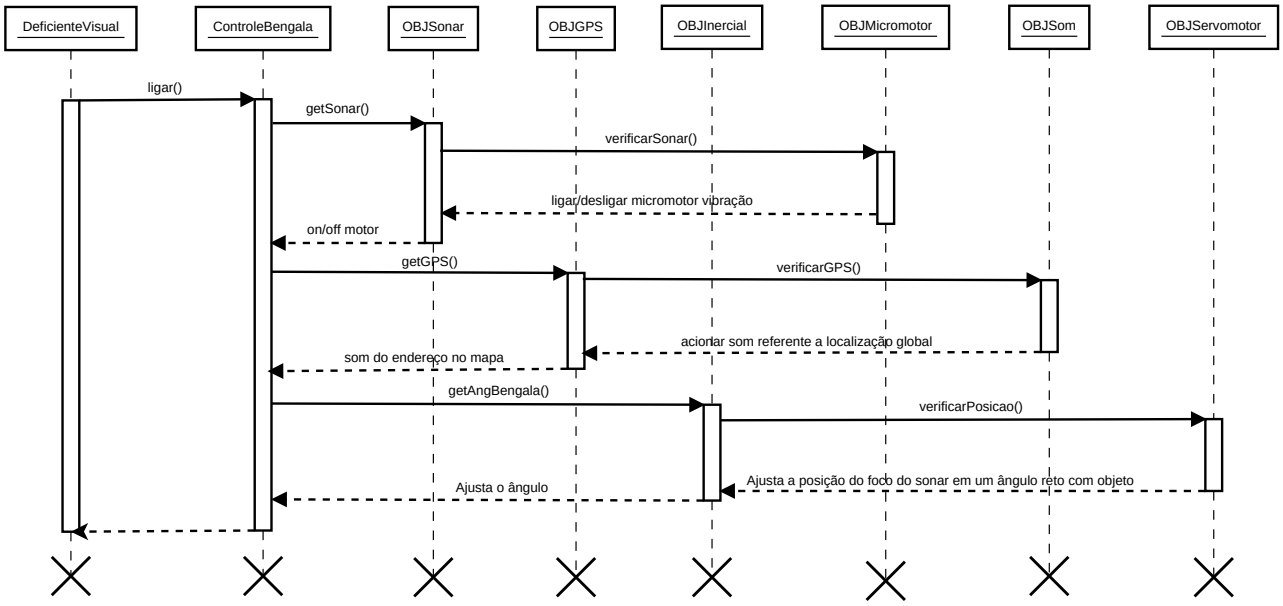


Figura 57: diagrama de sequência parcial

formação adicionada em uma etapa do desenvolvimento não é necessariamente perdida ou traduzida para a etapa seguinte. Ocorre uma redução na quantidade de erros com consequente diminuição do tempo nas etapas de codificação e teste. A criação de novos objetos que se comuniquem com os já existentes não obriga o desenvolvedor a conhecer o interior destes últimos. As análises de projeto orientadas a objetos têm como meta identificar o melhor conjunto de objetos para descrever um sistema de *software*. O funcionamento deste sistema se dá através do relacionamento e troca de mensagens entre estes objetos. As desvantagens incluem a apropriação, porque a análise orientada a objetos nem sempre soluciona os problemas elegantemente. Os critérios para classificar objetos podem mudar significativamente. Além disso, algumas vezes não é possível decompor problemas do mundo real em uma hierarquia de classes. O paradigma de objetos não trata bem de problemas que requerem limites nebulosos e regras dinâmicas para a classificação de objetos. Isto leva ao próximo problema: fragilidade. Desde que uma hierarquia orientada a objetos requer definições precisas, se os relacionamentos fundamentais entre as classes chave mudam, o projeto original orientada a objetos é perdido. Torna-se necessário reanalisar os relacionamentos entre os objetos principais e reprojetar uma nova hierarquia de classes (GUIMARÃES; HENRIQUES; PEREIRA, 2013).

4.4.6 Sistema de Rastreamento

O módulo GPS envia as informações de posicionamento global utilizando um protocolo de comunicação, como por exemplo, o protocolo *NMEA 0183*. Nesse caso, o protocolo é baseando em *American Standard Code for Information Interchange* (ASCII) e transmitido serialmente para o controlador que transfere os dados através de uma conexão GSM para GGSN (*Gateway GPRS Support Node* do operador móvel que fornece os dados para um servidor remoto através de uma conexão TCP (*Transmission Control Protocol*), como mostra a Figura 58.

Cada mensagem contida em um pacote começará com símbolo $. Os próximos cinco caracteres identificam a troca de informação e o tipo de mensagem. Os dados são separados por uma vírgula. Há quase vinte tipos de frases interpretadas no protocolo NMEA,

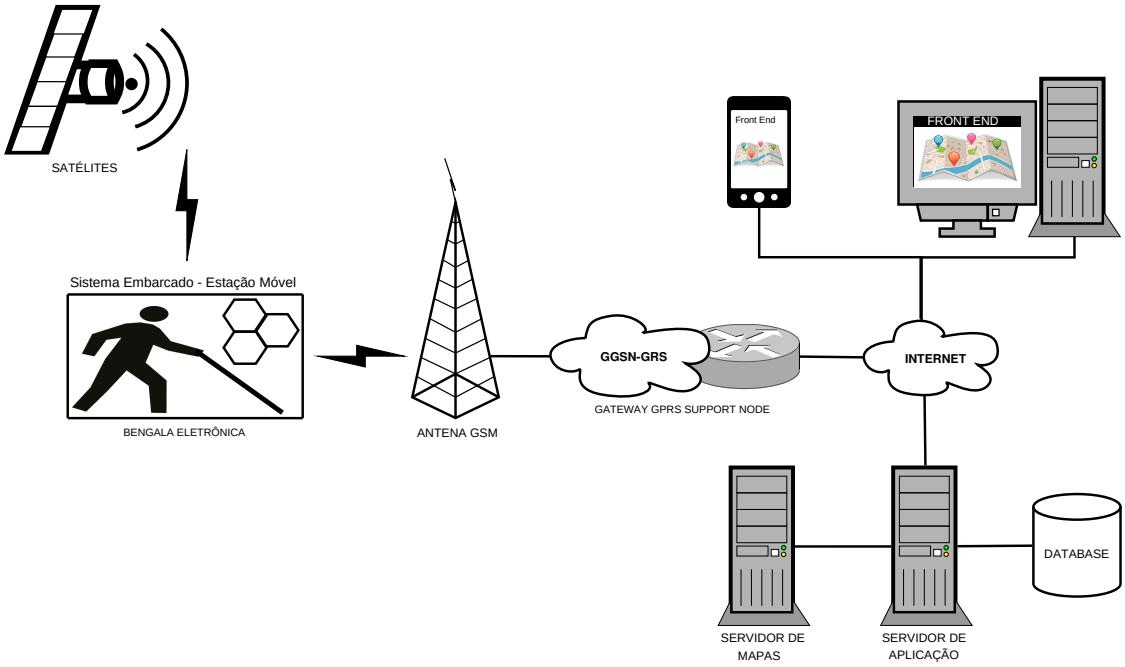


Figura 58: Arquitetura do sistema de rastreamento. Fonte: Autor.

mas quando ele é usado com um dispositivo GPS, apenas um par são utilizados.

Através das expressões regulares, correspondentes aos padrões das sentenças, é possível extrair do conjunto de informações fornecidas pelo módulo receptor GPS somente os dados necessários, como latitude, longitude, indicadores Norte, Sul, Este e Oeste, data, hora, altitude e velocidade.

A vírgula é um parâmetro importante no protocolo *NMEA 0183*, pois quando o equipamento não consegue captar as informações do satélite somente as vírgulas são transmitidas. Com isso, o *software* implementado no microcontrolador conta as vírgulas para buscar as informações úteis ao projeto. Depois de receber, a conexão, todas as informações enviadas pela rede de celular são recebidas, processadas e armazenadas no servidor da aplicação com bancos de dados, Figura 59. Deste modo, pretende-se obter um sistema para o rastreamento de bengalas eletrônicas para as pessoas com deficiência visual e assim auxiliar em seu deslocamento retornando informações da macro e micro navegação.

A primeira camada, chamada *Front-End*, usualmente são *browsers*, que servem para apresentação e algumas validações. A segunda camada, é a de aplicação, executada no servidor de aplicação, fazendo assim com que seja mais fácil o desenvolvimento, manutenção e gerenciamento de sistemas complexos. E a terceira camada é o servidor de um sistema de banco de dados para armazenar as informações do projeto. O Zope (*Z Object Publishing Environment*) é uma plataforma de desenvolvimento de aplicações *Web* de código aberto baseada em *Python*. *Zope* integra um grande número de ferramentas e funcionalidades das quais uma base de dados objeto, um módulo de publicação de objetos *Web*, e uma linguagem de geração dinâmica de páginas. Contrariamente às outras soluções do mercado, a finalidade de *Zope* não é publicar páginas HTML mas objetos que podem ser montados automaticamente a partir de componentes cujo comportamento, dados e a aparência são configuráveis pelo projetista. Esta abordagem torna possível à publicação de conteúdo *Web*. O servidor de aplicação, é o sistemas de *software* que fornece a infraestrutura de serviços para a execução de aplicações distribuídas do projeto. Os servidores de aplicação são executados em máquinas servidoras e são acessados pelos cli-

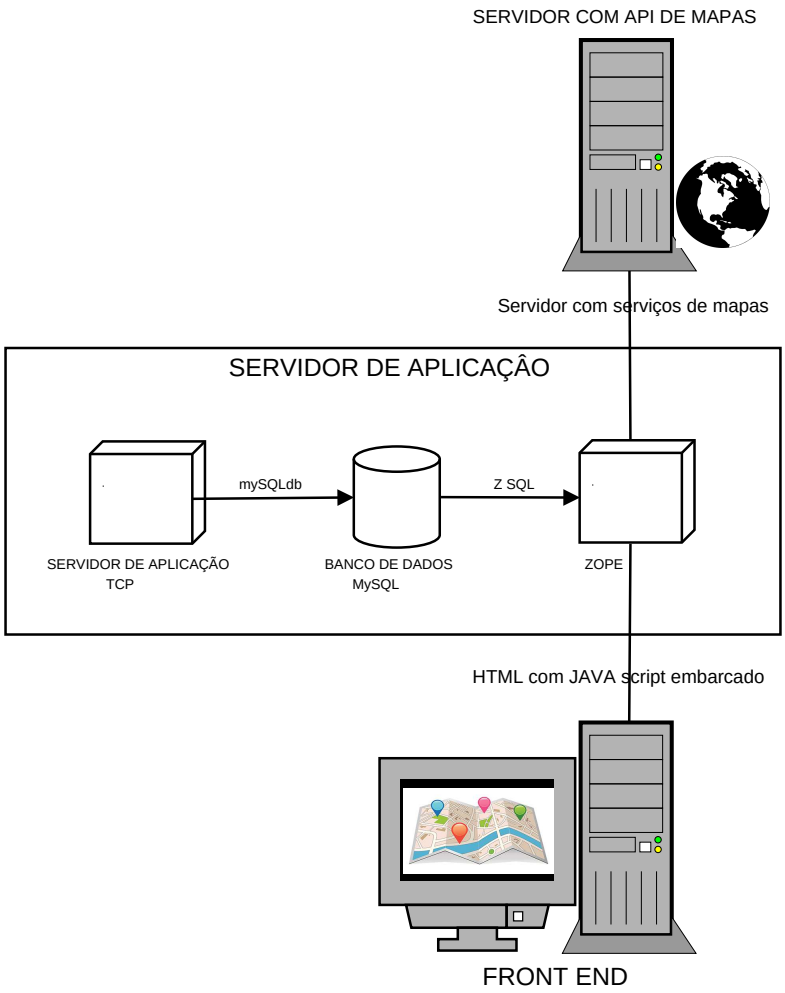


Figura 59: Descrição do sistema com Servidor de Aplicação, Banco de Dados e Web Browsers Clientes para o sistema de rastreamento

entes através de uma conexão de rede. Em geral estes serviços diminuem a complexidade do desenvolvimento, controlam o fluxo de dados e gerenciam a segurança. O servidor de aplicação utiliza a arquitetura chamada de 3-camadas ou n-camadas, que permite um melhor aproveitamento das características de cada componente (servidor de banco de dados, servidor de aplicação e cliente). E por fim o MySQL é o sistema de gerenciamento de banco de dados que utiliza a linguagem SQL (*Structured Query Language*) para realizar consultas a uma base de dados, é também rápido e flexível o suficiente para permitir armazenar *logs* e informações provenientes da bengala (estação móvel) para armazenar as informações das aquisições dos diferentes dispositivos utilizados no projeto. As principais vantagens do MySQL são velocidade, robustez, facilidade de uso, e também por que trabalha com diferentes plataformas de *software* de sistema (GUIMARÃES; PEREIRA; HENRIQUES, 2014).

5 *FRAMEWORK* SOA E OS ESTUDOS DE CASOS

Este capítulo descreve os resultados obtidos dos dois estudos de casos embarcados integrados com um *framework* SOA. A implementação faz referências diretas sobre o que foi apresentado no capítulo anterior, onde a proposta geral dos estudos de casos foram definidos. A forma como as entidades que fazem parte da arquitetura orientada a serviços são implementadas são descritos e são apresentados resultados parciais do projeto.

5.1 Proposta de Arquitetura para o Framework SOA

A proposta da arquitetura orientada a serviços é um modelo para construção de soluções de *software* que utiliza como seu principal elemento unidades de desenvolvimento denominadas serviços, que são elementos auto-descritos, agnósticos de plataforma, que executam funções e que podem variar desde simples requisições até processos de negócio complexos. O modelo em camadas da arquitetura orientada a serviços prove serviços consumidos por pessoas ou outras organizações para executarem suas atividades, viabilizando a composição de novos serviços e processos. A arquitetura proposta foi criada a partir dos conceitos abstraídos em Erl (2007, 2005) e Fugita (2009), fazendo uma fusão dos agentes junto com as camadas de uma arquitetura SOA, Figuras 09 e 10 da subseção 2.6.1.

Para iniciar, é importante observar que neste contexto dispomos de dois papéis: o provedor, que é a organização que efetivamente executa o serviço; e o consumidor, que é a organização que consome o serviço. Para que a relação entre provedor e consumidor ocorra de maneira adequada, ambos precisam acordar que funções serão disponibilizadas pelos serviços, as informações que devem ser informadas pelos consumidores para sua execução e o nível de serviço esperado, contemplando variáveis como tempo de resposta, volume e disponibilidade do serviço. Estas informações são descritas de maneira formal através de uma especificação denominada contrato. É uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes. Com esta tecnologia é possível que novas aplicações possam interagir com aquelas que já existem e que sistemas desenvolvidos em plataformas diferentes sejam compatíveis. A Figura 60 mostra a arquitetura de camadas proposta para o desenvolvimento do *framework* SOA.

Os serviços são componentes que permitem às aplicações enviar e receber dados. A proposta da arquitetura SOA, desenvolvida é constituída por três componentes básicos: o servidor de registro (*broker server* ou *service registry*), o provedor de serviços (*service provider*) e o solicitante de serviços (*service consumer* ou *service requestor*). As interações entre esses componentes são de busca, publicação e interação de operações.

Na operação de publicação o provedor publica a descrição do serviço de tal forma que um solicitante possa localizá-la. Na operação de busca o solicitante obtém a descrição

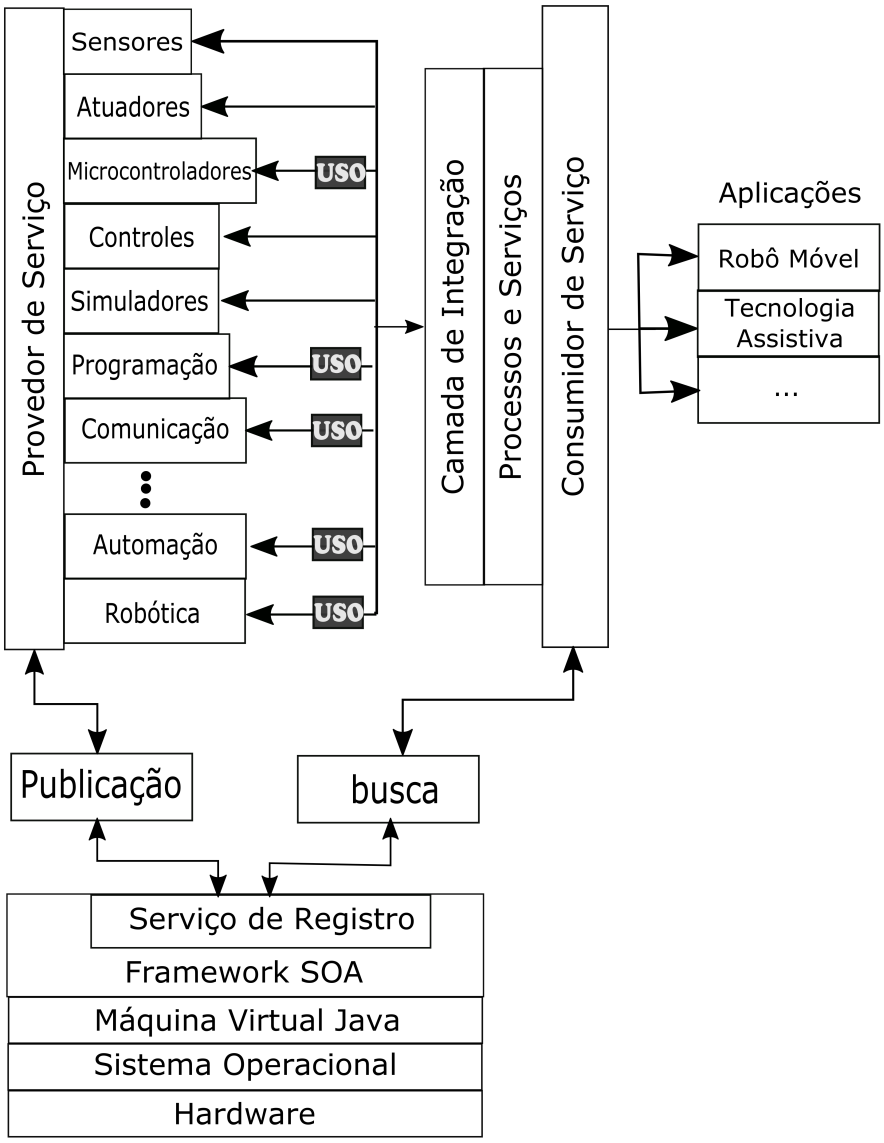


Figura 60: Arquitetura de camadas do *Framework* SOA. Fonte: Autor.

do serviço diretamente ou consulta o servidor de registro procurando pelo tipo de serviço desejado. Essa operação pode ser executada em duas fases distintas: desenvolvimento ou execução. Na operação de interação o solicitante chama ou inicia uma interação com o provedor, em tempo de execução, utilizando os detalhes contidos na descrição do serviço para localizar, contactar e chamar o serviço.

O provedor de serviços representa a camada que hospeda o serviço permitindo que os clientes acessem o serviço. O provedor de serviços fornece o serviço e é responsável por publicar a descrição do serviço que provê. O solicitante de serviços é a aplicação que está procurando, invocando uma interação com o serviço, ou seja, requisita a execução de um serviço. O consumidor de serviço pode ser uma pessoa ou organização. O servidor de registro é um repositório central que contém a descrição (informação) de um serviço, e é por meio do servidor de registro que essas descrições são publicadas e disponibilizadas para localização. Os consumidores buscam por serviços no servidor de registro e recuperam informações referentes à interface de comunicação para os serviços durante a fase de desenvolvimento ou durante a execução do cliente, denominadas interação estática (*static bind*) e interação dinâmica (*dinamic bind*), respectivamente. Na interação estática, o cliente recupera a assinatura do serviço, necessária à codificação. Na interação dinâmica, o cliente recupera os valores de parâmetros e a localização do serviço.

Por último, os consumidores se comunicam com os diversos serviços por meio do barramento ESB (*Enterprise Service Bus*) como camada de integração, também responsável por coordenar os processos e serviços. A gestão dos processos de negócio (*Business Process Management*) deve ser realizada entre negócio e TI, a fim de realizar a identificação e definição dos serviços bem como a composição dos fluxos de negócio, promovendo a implementação de SOA de forma adequada às reais necessidades da aplicação. SOA, então é uma evolução dos paradigmas de arquitetura existentes até o momento. SOA combina elementos interdisciplinares, tais como modelagem e gestão de processos de negócio, arquitetura de *Software*, computação distribuída e gestão de sistemas (FUGITA, 2009).

5.2 Papéis Envolvidos no Uso e Desenvolvimento do *Framework* SOA

O desenvolvimento tradicional de aplicações envolve dois tipos de indivíduo: desenvolvedor de aplicação e usuário de aplicação (nos dois casos isto pode corresponder a grupos de indivíduos, com diferentes funções). Desenvolvedores devem levantar os requisitos de uma aplicação, desenvolvê-la (o que inclui a documentação que ensina a usar a aplicação, como manuais de usuário) e entregá-la aos usuários. Usuários interagem com uma aplicação apenas através de sua interface. A Figura 61 apresenta os indivíduos envolvidos neste caso.

O desenvolvimento de *frameworks* introduz outro indivíduo, além de desenvolvedor e usuário de aplicação: o desenvolvedor de *framework*. No contexto dos *frameworks*, o papel do usuário de aplicação é o mesmo descrito acima. O papel do desenvolvedor de aplicações difere do caso anterior pela inserção do *framework* no processo de desenvolvimento de aplicações. Com isto, o desenvolvedor de aplicações é um usuário de um *framework*, que deve estender e adaptar a estrutura deste *framework* para a produção de aplicações. Ele tem as mesmas funções do caso anterior: obter os requisitos da aplicação, desenvolvê-la usando o *framework*, (o que em geral, não dispensa completamente do desenvolvedor de aplicações a necessidade de produzir código) e desenvolver a documentação da aplicação. O novo papel criado no contexto dos *frameworks*, o desenvolvedor

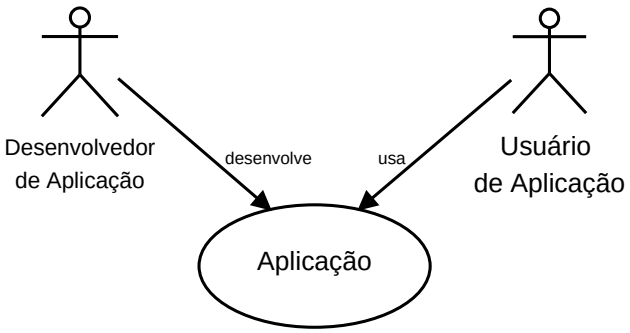


Figura 61: Elementos do desenvolvimento tradicional de aplicações. Fonte: (SILVA, 2000).

de *framework*, tem a responsabilidade de produzir *frameworks* e algum modo de ensinar como usá-los para produzir aplicações. A Figura 62 apresenta os indivíduos envolvidos neste último caso

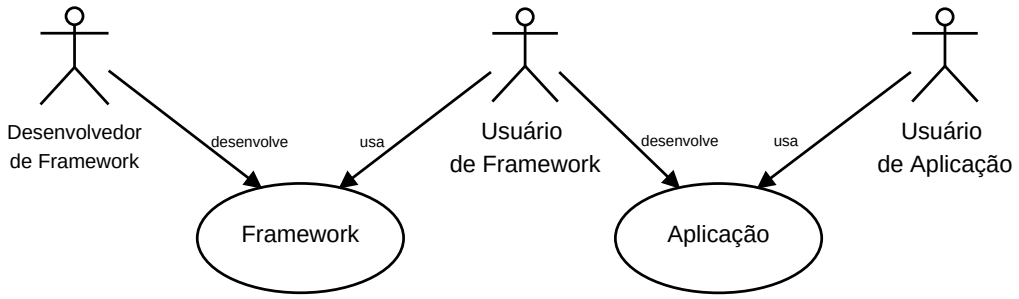


Figura 62: Elementos do desenvolvimento de aplicações baseado em *frameworks*. Fonte: (SILVA, 2000).

Para uma melhor abstração dos papéis envolvidos no projeto do *framework SOA*, é apresentado um modelo conceitual, que provê uma solução para uma família de problemas semelhantes, onde as aplicações possuem em comum um mesmo tipo de problema. Para tanto, o *framework* deve ser constituído por um conjunto de classes abstratas e concretas, bibliotecas de código e componentes de *software* que colaboram para atingir um mesmo objetivo. A Figura 63 mostra os elementos contidos no modelo conceitual.

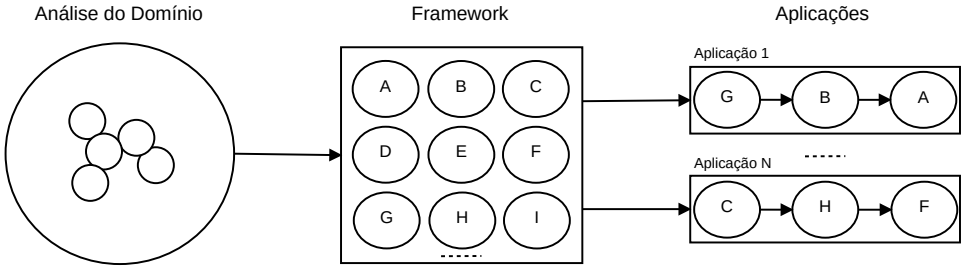


Figura 63: Os elementos utilizados nas aplicações são baseados nos serviços fornecidos pelo *framework*. Fonte: Autor.

O uso de *framework* em projetos apresenta algumas vantagens: modularidade, pois permite encapsular implementações flexíveis em uma interface estável; reuso, uma vez

que define elementos genéricos que podem ser reaproveitados e ao mesmo tempo permitir a eficiência das aplicações. Na fase de análise do domínio é feita uma avaliação do problema a ser tratado para tornar conhecido seu domínio e aprofundar a experiência numa dada família de aplicações (SILVA, 2000).

5.3 Repositório do *Framework SOA*

No contexto da Engenharia de *Software*, diferentes abordagens buscam melhorar a qualidade dos artefatos de *software*, bem como diminuir o tempo e o esforço necessários para produzi-los. Como já mencionado no capítulo 2, *frameworks* são estruturas de classes que constituem implementações incompletas que, estendidas, permitem produzir diferentes artefatos de *software*. A grande vantagem desta abordagem é a promoção de reuso de código e projeto, que pode diminuir o tempo e o esforço exigidos na produção de *software*.

Foi definida a utilização da linguagem de programação Java para o desenvolvimento do *framework* SOA pretendido. Java é o termo geral usado para denotar o *software* e seus componentes, que incluem Java *Runtime Environment* (JRE), Java *Virtual Machine* (JVM) e também *plug-in*. A linguagem suporta reflexão, o que ajuda a expor de forma dinâmica a funcionalidade dos serviços e provê centenas de *frameworks* e API's para as mais diversas finalidades, também há uma enorme quantidade de projetos Java bem organizados e de código aberto que são relevantes para a robótica. A linguagem de programação foi escolhido para facilitar a interoperabilidade e alavancar essa funcionalidade potencial, além de suportar implementação de bibliotecas escrita em C/C++, *Assembler*, e outras tantas linguagens de programação utilizando Java *Native Interface* (JNI) em conjunto com Java *Native Access* (JNA), que é uma biblioteca que abstrai essas chamadas e os tipos de dados de uma linguagem para outra que facilita muito a integração.

Para fazer a integração de um ambiente para a manipulação e controle de diferentes serviços de *software* e *hardware* foi escolhido o *framework* SOA *Myrobotlab*, que disponibiliza um conjunto de classes, bibliotecas de código e componentes, que colaboram para prestar diferentes tipos de serviços. Do ponto de vista prático do projeto pretendido, o *framework* SOA Myrobotlab é utilizado como uma semi-aplicação flexível e extensível para permitir a elaboração de partes complementares específicas das aplicações possíveis, como por exemplo, os estudos de caso apresentados no capítulo 5. Há duas maneiras de usar o *framework* SOA MyRobotLab. Eles são:

- Modo Desenvolvedor - modo desenvolvedor é alguém que está interessado em reutilizar as classes, bibliotecas de códigos e artefatos de *software*, para adaptar e estender as funcionalidades para diferentes aplicações ou criar novos serviços. Para utilizar o *framework* no modo desenvolvedor o requisito obrigatório é saber programar. No modo desenvolvedor é preciso descarregar os pacotes do repositório do servidor do *Myrobotlab*, local do armazenamento dos pacotes do *software* que podem ser recuperados e instalados em um computador *host*. Muitos editores de *software* e outras organizações mantêm servidores na *internet* para este fim. Este é o modo utilizado para o desenvolvimento da proposta do *framework* SOA, onde através do reuso é possível criar um *framework* SOA aplicado para os estudos de casos.

- Usuário - alguém que usa o *framework* SOA *Myrobotlab* em algum projeto. Eles estão interessados em utilizar as capacidades atuais do *framework* SOA e não estão

interessados no desenvolvimento de novos serviços. Embora seja possível desenvolver novas funcionalidades em *Python* dentro de um contexto 'Usuário' e não precisando de uma IDE para desenvolvimento ou quaisquer outras dependências.

Myrobotlab é executado no Java, portanto é multi-plataforma e qualquer computador ou dispositivo que suporte esta JVM conceitualmente poderá executar o *framework*. Alguns dos serviços oferecidos por Myrobotlab incluem, suportar tarefas de *multithreaded*, soluções distribuídas e capacidade de multiprocesso. Integra outros componentes de projetos de código aberto para as funcionalidade de serviços, suporta comunicação serial com microprocessadores. Possui serviço de visão de máquina, sistema de controle, reconhecimento de Voz, síntese de fala, entre outros serviços. A Figura 64 mostra o diagrama de blocos da instalação do repositório do *Myrobotlab*.

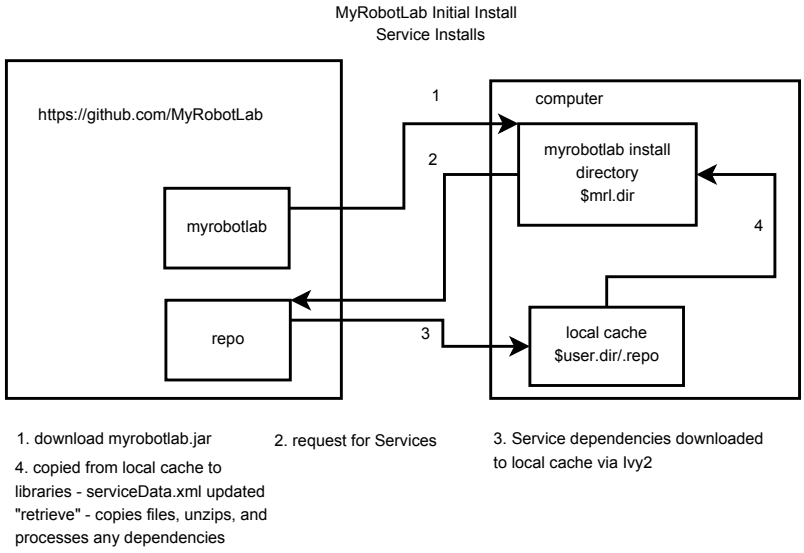


Figura 64: Repositório dos pacotes Myrobotlab. Fonte: (MYROBOTLAB, 2014).

Para utilizar o *Myrobotlab* no modo desenvolvedor é utilizado o ambiente Eclipse para desenvolvimento Java. O Eclipse é a IDE (*Integrated Development Environment*) ou seja ambiente integrado de desenvolvimento. O que pesou a escolha da IDE foi a grande comunidade *open-source* do *software*, cujos projetos estão focados em construir uma plataforma aberta de desenvolvimento composta por ferramentas e *runtimes* para a construção, implantação e gestão de *software* em todo o ciclo de vida, além de ser o ambiente de desenvolvimento padrão do *Myrobotlab* (MYROBOTLAB, 2014).

5.4 Implantação do Projeto

Entre as versões 1.5 e 2.0 da UML, diversas alterações ou evoluções foram realizadas. Muitos dos diagramas abordados ao longo deste artigo são resultados nítidos de tal evolução da UML. O Diagrama de Implantação determina as necessidades de hardware do sistema, as características físicas como servidores, estações, topologias e protocolos de comunicação, ou seja, todo o aparato físico sobre o qual o sistema deverá ser executado. Os Diagramas de Componentes e de Implantação são bastante associados, podendo ser representados em separado ou em conjunto. É o diagrama com a visão mais física da UML. Este diagrama foca a questão da organização da arquitetura física sobe a qual o software irá ser implantado e executado em termos de hardware, ou seja, as máquinas

(computadores pessoais, servidores etc.) que suportam o sistema, além de definir como estas máquinas serão conectadas e por meio de quais protocolos se comunicarão e transmitirão as informações.

Os elementos básicos deste diagrama são os Nós, que representam os componentes, Associações entre Nós, que são as ligações entre os Nós do diagrama, e os Artefatos, representações de entidades físicas do mundo real. Nós podem conter outros nós, sendo comum encontrar um nó que representa um item de hardware contendo outro nó que representa um ambiente de execução, embora nó que represente um item de hardware possa conter outros nós representando itens de hardware, e um nó que represente um ambiente de execução possa conter outros ambientes de execução. Quando um nó representa um hardware, pode possuir estereótipos «device» e «processor»; quando, porém, um nó representa um ambiente de execução, pode utilizar o estereótipo «ExecutionEnvironment» (Da Silva, 2015).

Antes de apresentar os diagramas de implantação da integração do *framework* SOA *Myrobotlab* com os estudos de caso do projeto, foram desenvolvidos diagramas de implantação individuais para cada nó de *hardware*do projeto: computador servidor, robô móvel e bengala eletrônica, cada um dos nós modelam o inter-relacionamento entre recursos de infra-estrutura, de rede ou artefatos de sistemas.

A finalidade do modelo de implantação é capturar a configuração dos elementos de processamento e a conexões entre eles no sistema, a Figura 65 apresenta o diagrama de implantação parcial do robô móvel.

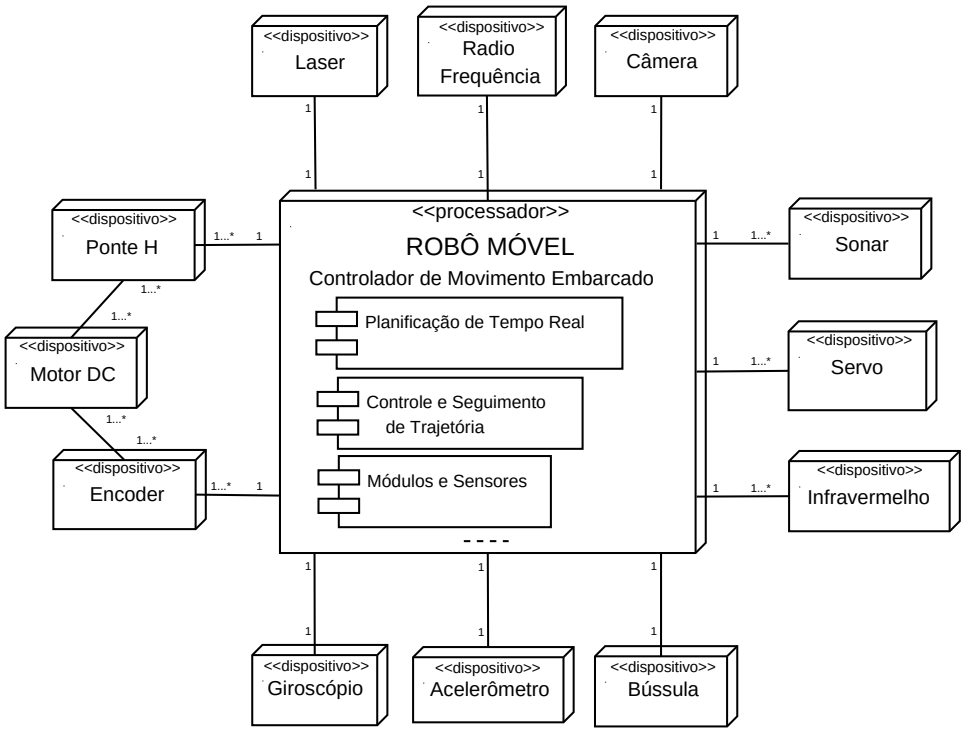


Figura 65: Modelo parcial para o diagrama de implantação do robô móvel. Nós dispositivos (ou processadores) podem ser adicionados ou removidos conforme as necessidades do problema a ser resolvido. Fonte: Autor.

O modelo de implantação é constituído de um ou mais nós (elementos de processamento com pelo menos um processador, memória e possivelmente outros dispositivos), dispositivos (nós estereotipados sem capacidade de processamento no nível de abstração

modelado) e conectores, entre nós, e entre nós e dispositivos, a Figura 66 apresenta o diagrama de implantação parcial da bengala eletrônica.

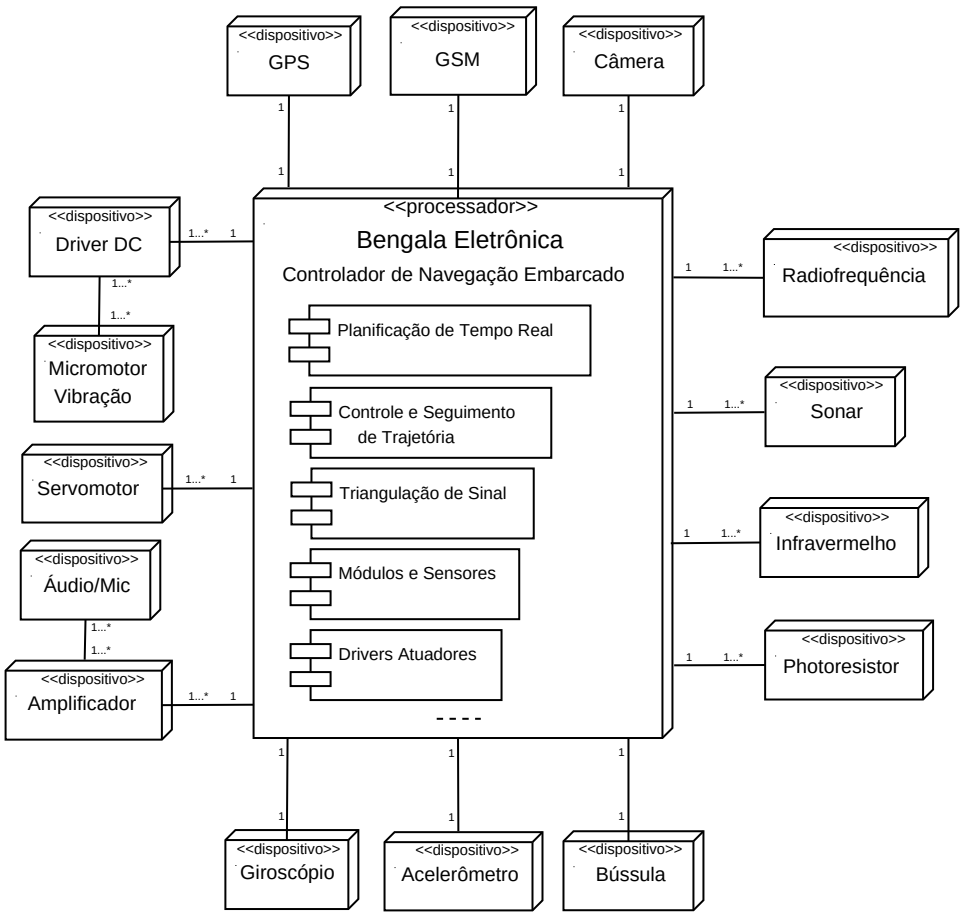


Figura 66: Modelo parcial para o diagrama de implantação da bengala eletrônica. Nós dispositivos (ou processadores) podem ser adicionados ou removidos conforme as necessidades do problema a ser resolvido. Fonte: Autor.

O modelo de implantação também mapeia processos para esses elementos de processamento, permitindo a distribuição do comportamento em nós a serem representados. Os seguintes papéis usam o Modelo de Implantação:

- O arquiteto de software - usa o modelo de implantação para capturar e compreender o ambiente de execução físico do sistema e compreender as questões de distribuição;
- Os designers (inclusive de software e de banco de dados) - para compreender a distribuição do processamento e dos dados no sistema;
- Os gerente de sistema - para compreender o ambiente físico em que o sistema é executado;
- O gerente de projeto - na estimativa de custos para o caso de negócios e para o planejamento de aquisição, instalação e manutenção.

Na fase de elaboração, o modelo de implantação será refinado para um nível de especificação, permitindo que o arquiteto de software preveja o desempenho com confiança,

antes de finalmente passar o modelo para o nível físico, no qual ele especifica a quantidade real de *hardware* e modelo a ser usada. Assim, ele se tornará um plano para a aquisição, instalação e manutenção do sistema. Se o ambiente de implantação já existir, será examinado para determinar se é capaz de suportar as novas capacidades do sistema que está sendo desenvolvido. Se forem necessárias mudanças no ambiente de implantação, elas serão identificadas nessa fase. Se o ambiente de implantação ainda não existir, serão definidos os números, os tipos e as configurações dos nós e da conexão entre os nós necessários para suportar a arquitetura.

5.4.1 Resultados do Computador *Host* com *Framework* SOA *Myrobotlab*

O diagrama de implantação representa como é realizada a distribuição do sistema através de nós de *hardware*, componentes e dependências de *software* e suas devidas relações de comunicação. A a Figura 67 apresenta o diagrama de implantação parcial do computador hospedeiro do projeto.

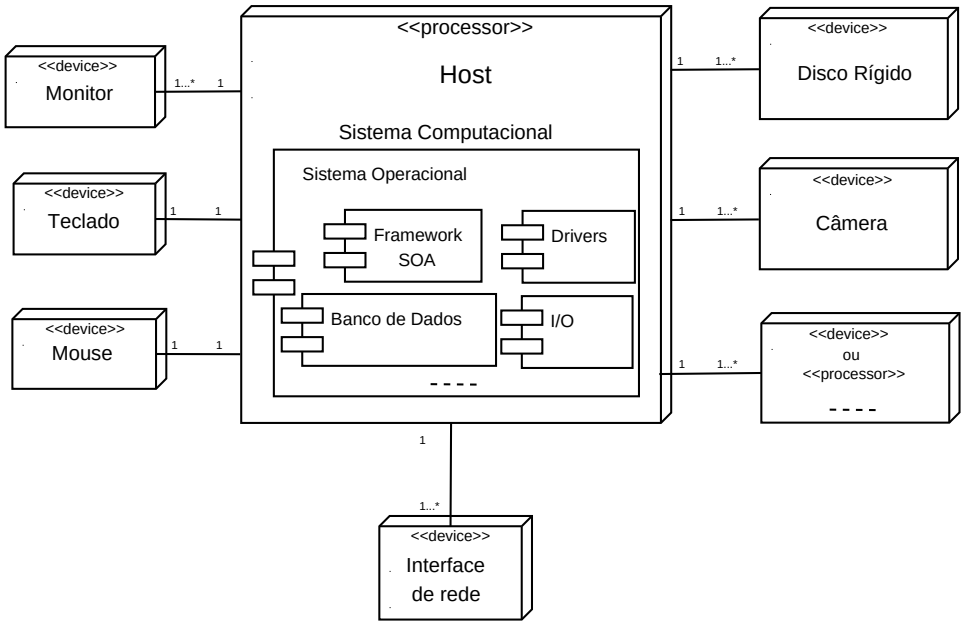


Figura 67: Modelo parcial para o diagrama de implantação do computador *Host*. Nós podem ser adicionados ou removidos, as definições são estabelecidas conforme as necessidades do problema. Fonte: Autor.

Para implementações do *framework* SOA *Myrobotlab* no modo desenvolvedor, é preciso descarregar os pacotes *repo* e *myrobotlab* do repositório para o computador *Host* via IDE Eclipse para desenvolvimento Java(MYROBOTLAB, 2014), a Figura 68 apresenta o sistema parcial dos diretórios do sistema *Myrobotlab* versão 1.0.12.

Posteriormente é preciso estudar o JavaDoc da API do *Myrobotlab*, a interface de programação é o conjunto de padrões que permitem a construção de novos aplicativos e a sua utilização. Há uma página de hierarquia para todos os pacotes, além de uma hierarquia para cada pacote. Cada página hierarquia contém uma lista de classes e uma lista de interfaces. As classes são organizadas por herança, começando a estrutura com *java.lang.Object*. As interfaces não herdam *java.lang.Object*.

A API do *Myrobotlab* fornece uma lista de todos os pacotes com uma descrição geral de cada pacote com listas de informações de classes, interfaces, métodos, construtores e

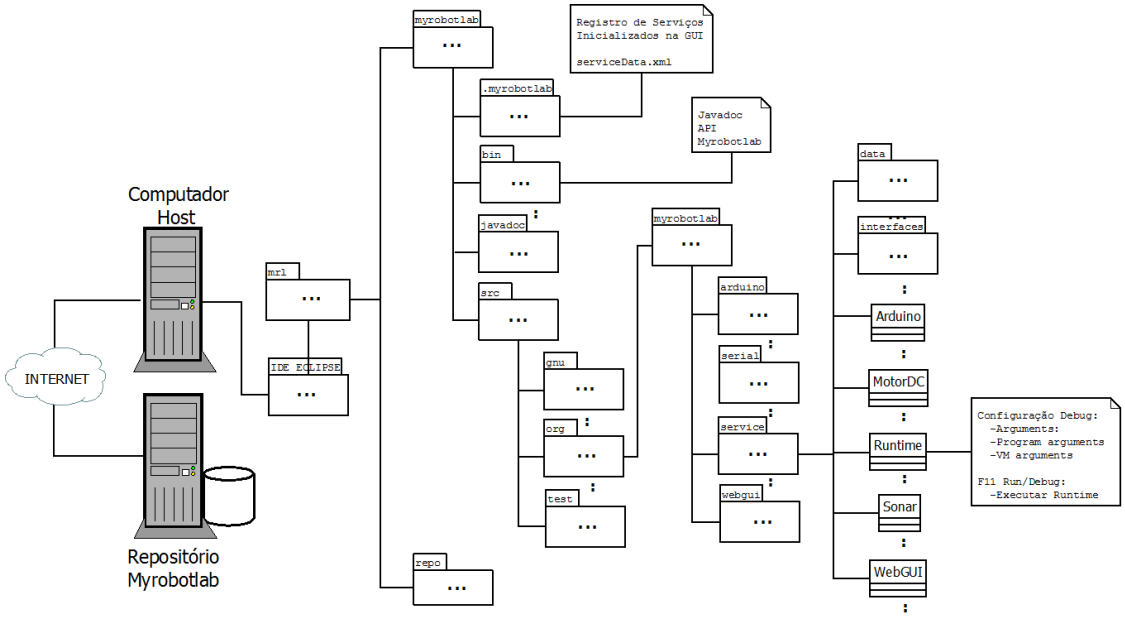


Figura 68: Sistemas de diretórios parciais do *framework* SOA *Myrobotlab* para desenvolvedores. Fonte: Autor.

outros campos referentes a implementação, essas informações foram amplamente utilizados neste trabalho com objeto de utilizar e estender o sistema para os estudos de casos. O Myrobotlab tem alta granularidade de códigos e bibliotecas, as quais possuem documentação limitada para desenvolvedores menos experientes com linguagens de programação de alto nível. Também possui pouca documentação de diagramas da arquitetura de *software* do sistema, o que torna complexo a utilização e entendimento do sistema legado. A falta de materiais é proporcional ao tempo de vida do *framework* SOA *Myrobotlab*, bem como a quantidade de colaboradores que ajudam no desenvolvimento do projeto. A classe *Runtime* do sistema, fornece o acesso ao ambiente de tempo de execução em que o aplicativo está sendo executado. Os métodos do tempo de execução permitem executar programas externos a partir de um aplicativo Java. *Runtime* é responsável pela criação e remoção de todos os serviços existentes e registros estáticos associados, ele mantém informações de estado em relação a possíveis serviços locais em execução e mantém informações de estado sobre *Runtimes* externos.

A classe *Runtime* deve ser o único serviço executando em um processo *host* e registros de mapas são utilizados no roteamento de comunicação para o serviço adequado (seja ele local ou remoto) será o primeiro serviço criado e ele também envolve o objeto real JVM *Runtime*. A Figura 69 mostra parte da representação da estrutura interna da classe *Runtime*(pacote de serviços), o diagrama apresentado, faz parte das poucas documentações existentes no diretório *"docs"*, baixado do repositório *Myrobotlab* (MYROBOTLAB, 2014).

A classe *Runtime* é um dos principais recursos existentes do pacote de serviços e pode ser acessado dentro do pacote *org.myrobotlab.service.Runtime*, descarregado do repositório. Antes de executar e depurar a classe *Runtime*, é preciso antes configurar o *debug*, conforme os argumentos da *Virtual Machine* (VM) do sistema operacional utilizado.

- *Windows* VM Argumento :`-Djava.library.path="libraries/native x86.32.windows;libraries/native/x86.64.windows";`

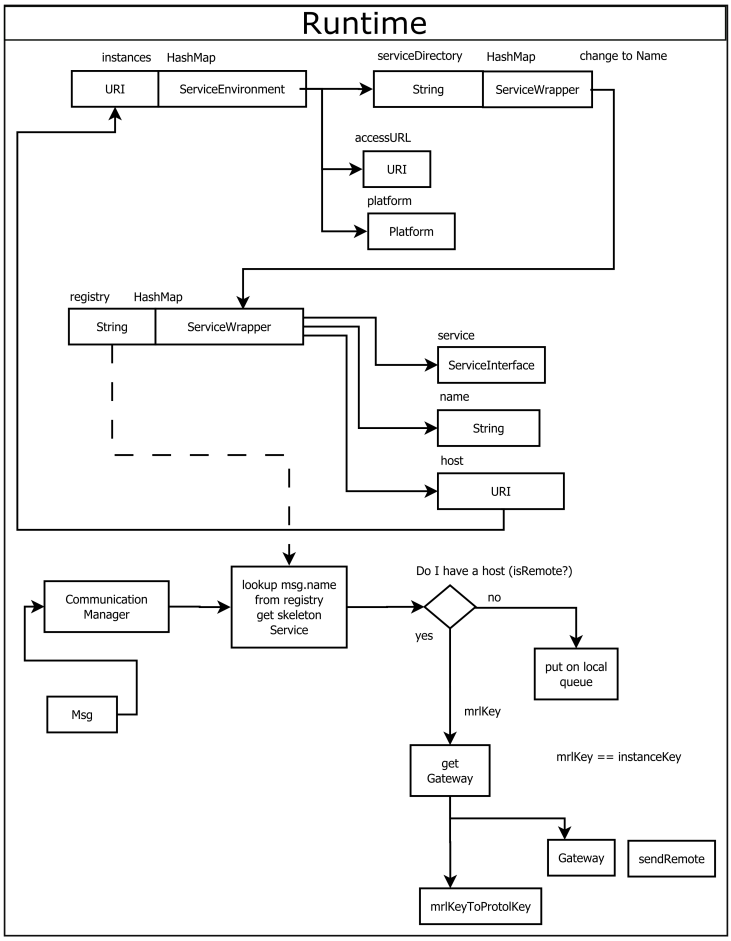


Figura 69: Representação da estrutura interna da classe *Runtime* do pacote de serviços. Fonte: (MYROBOTLAB, 2014).

- *Linux* VM Argumento: `-Djava.library.path="./libraries/native/x86.32.linux:./libraries/native/x86.64.linux`;
- *Mac* VM Argumento: `-Djava.library.path="./libraries/native/x86.32.mac"`;

Para executar a GUI do *framework* SOA *Myrobotlab* é preciso configurar o argumento de programa com os seguintes parâmetros:

- Argumento de programa: `-service gui GUIService -logToConsole`;

A seguir, a Figura 70 mostra parte da representação da estrutura interna da classe *GUIService* (pacote de serviços). A classe *GUIService* é onde são criados e utilizados os serviços, permite que recursos de controle de outros serviços sejam exibidos. Ele é definido para iniciar automaticamente quando um dos scripts (*myrobotlab.bat, myrobotlab.sh, jython.bat, ou jython.sh*) são executados. As aplicações gráficas são aquelas que possibilitam o uso ou a criação de uma *Graphical User Interface* (GUI).

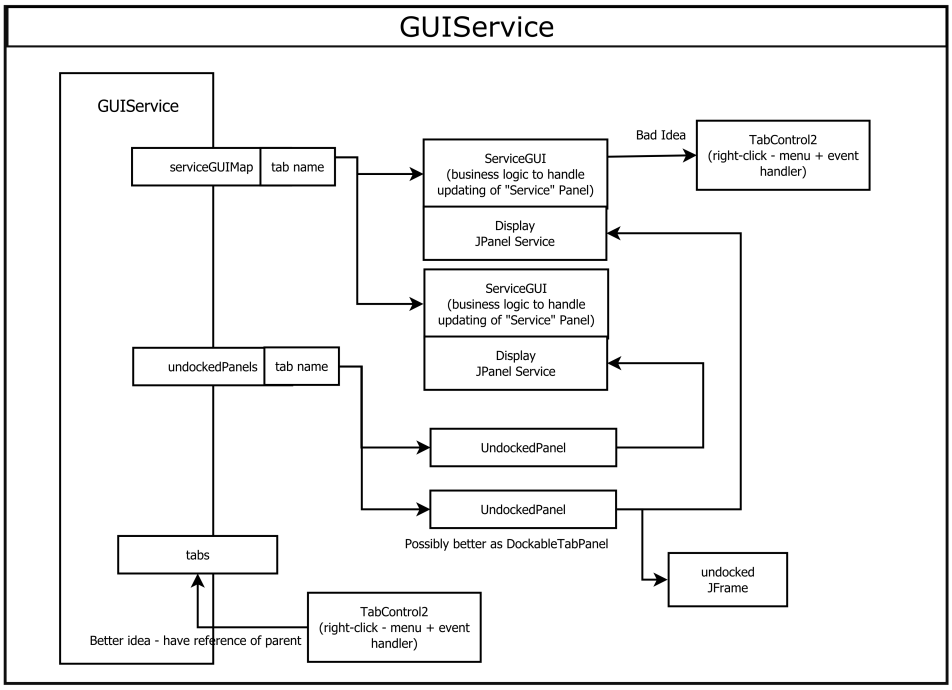


Figura 70: Representação da estrutura interna da classe GUIService do pacote de serviços. Fonte: (MYROBOTLAB, 2014).

Ao desenvolver uma aplicação dotada de uma GUI, é necessário definir quais componentes (objetos) serão utilizados e a disposição que eles terão na janela, conforme o projeto Interação Homem-Máquina (IHM). O *swing* possui inúmeras classes utilizadas na construção da GUI do *framework* SOA *Myrobotlab*. Neste ponto, o projeto começa a ficar interessante, pois as aplicações são criadas e controladas a partir de janelas gráficas. Ao projetar uma aplicação gráfica, é necessário definir todos os componentes que serão utilizados, seus objetivos e sua posição na janela. A Figura 71 mostra a primeira execução e depuração da classe *Runtime* da versão 1.0.12 do *framework* SOA *Myrobotlab*.

Após a inicialização da GUI, podem ser observadas três abas principais: *frame "welcome"*, *frame "gui"* e *frame "runtime"*. A aba do *frame "welcome"*, é responsável pelas boas vindas do sistema e é o primeiro *frame* que o usuário tem contato.

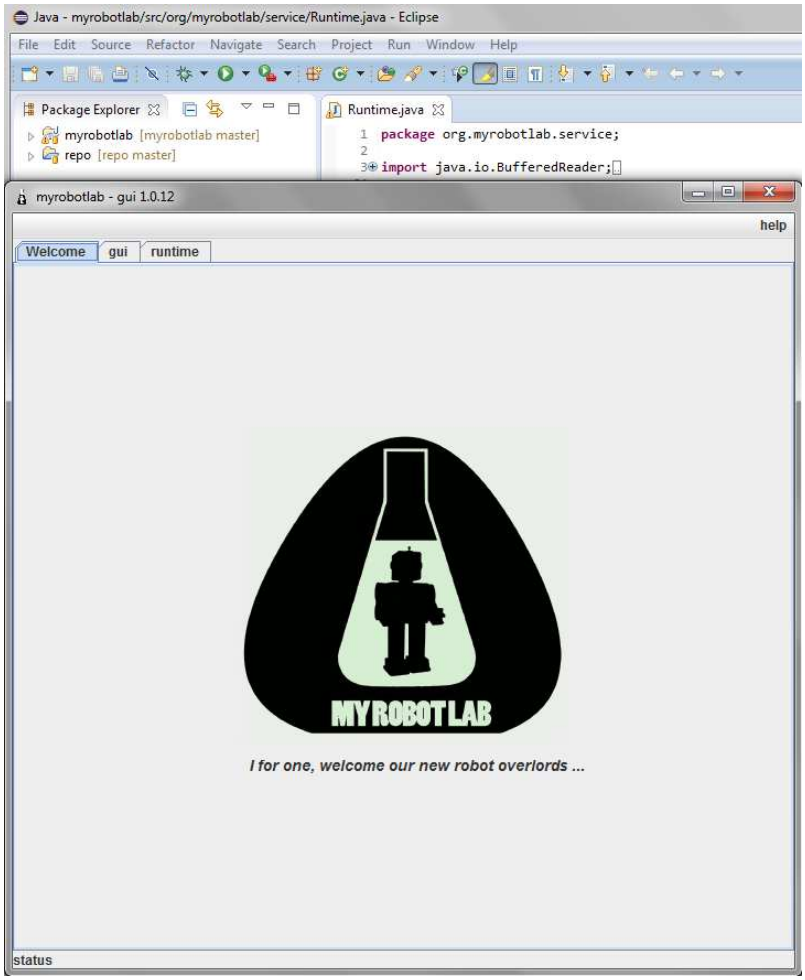


Figura 71: Primeira inicialização do *Runtime* após configurações dos argumentos do sistema. Execução da GUI do *framework* SOA *Myrobotlab* v1.0.12. Fonte: Autor.

A aba do *frame "gui"*, é responsável por fornecer serviços com mapas de rotas de mensagens e ícones dos serviços atualmente em execução, onde também podem ser passados parâmetros de entradas e saídas para os objetos. A Figura 72 apresenta o *frame* do serviço *"gui"*.

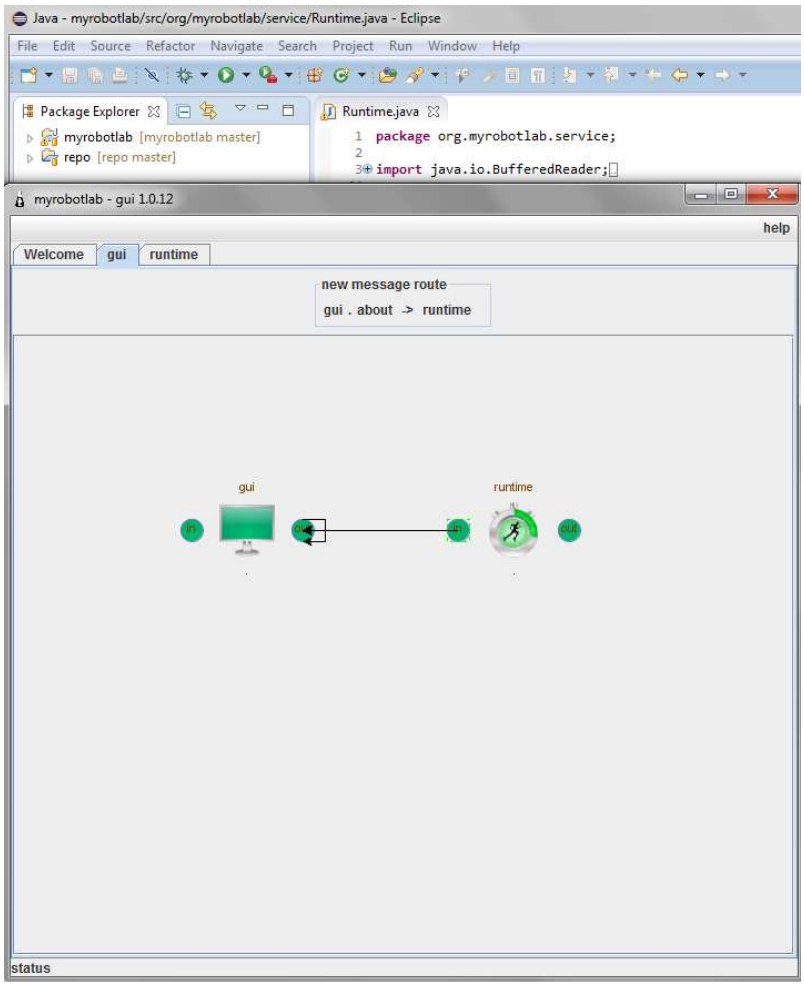


Figura 72: Acesso a aba do *frame* "gui". Responsável por fornecer o controle dos serviços em execução. Fonte: Autor.

E por último a aba do *frame "runtime"*, é responsável pelo fornecimento de todos os serviços disponíveis, onde o serviço requerido pelo consumidor é inicializado em uma nova aba na GUI do sistema. A Figura 73, mostra alguns dos serviços fornecidos por *default* na GUI, acessados pela aba *runtime*.

Do repositório *Myrobotlab* v1.0.12, apenas alguns dos serviços vem instalados por *default*, para fazer a instalação de todos os serviços deve-se inicializar a GUI do usuário e acessar a aba *"runtime"*, menu "system"e *"install all"*, assim, todos os serviços serão instalados e atualizados direto do repositório do *framework* SOA *Myrobotlab*. A instalação de serviços também pode ser feita individualmente, instalando somente o serviço requerido, caso não haja a necessidade de ter todos os serviços instalados. Após a instalação ou atualização é preciso reiniciar a execução do sistema para os serviço ficarem disponíveis. A Figura 74, apresenta todos os serviços instalados na GUI, acessados pela aba do *frame "runtime"*. Vale ressaltar, que alguns serviços, mesmo depois de instalados, podem não estar ativados ou implementados,

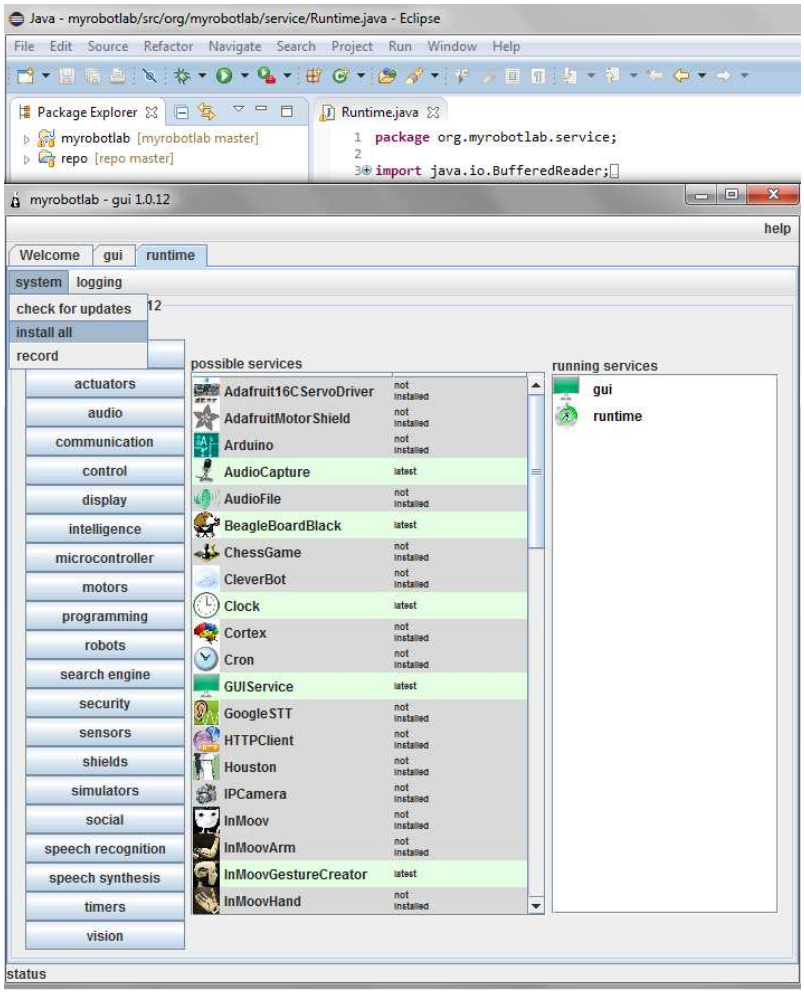


Figura 73: Serviços disponíveis na aba *"runtime"* da GUI. Fonte: Autor.

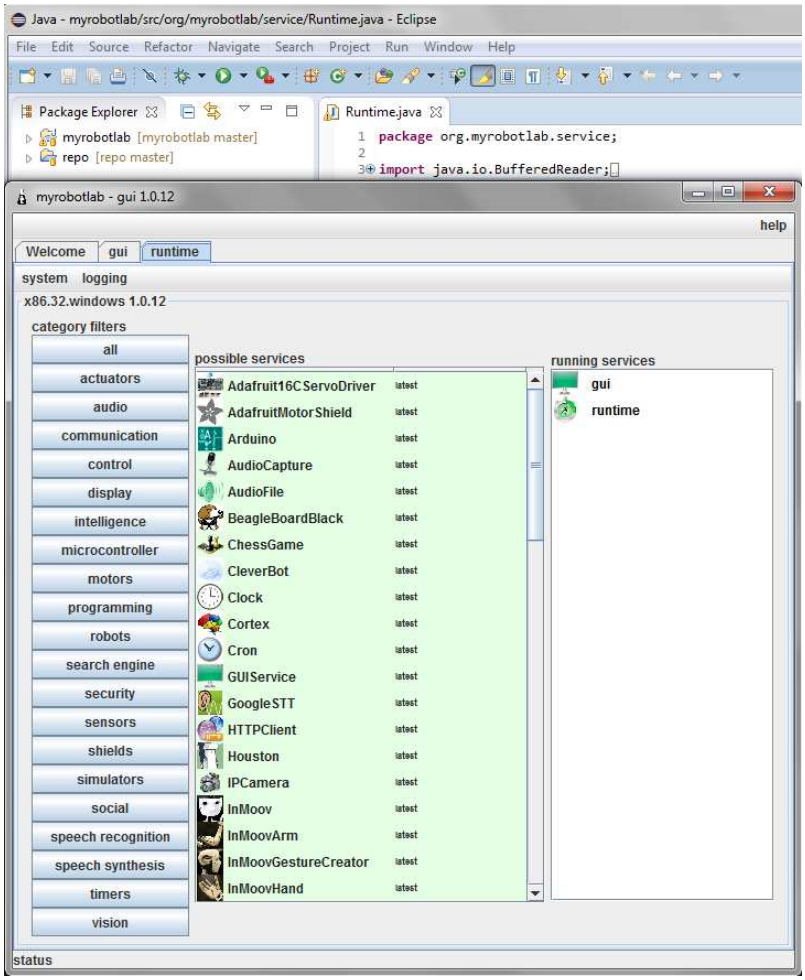


Figura 74: *Frame* "runtime"da GUI do *framework* SOA *Myrobotlab* inicializado com todos os serviços instalados e atualizados. Fonte: Autor.

A Figura 75, apresenta como é feito o acesso e a inicialização de serviços no *frame "runtime"*, como exemplo, usaremos dois serviços do *Myrobotlab*: o primeiro serviço é pouco implementado e não é fornecido(*"RasPi"*), já o segundo serviço é fornecido e bem consolidado(*OpenCV*).

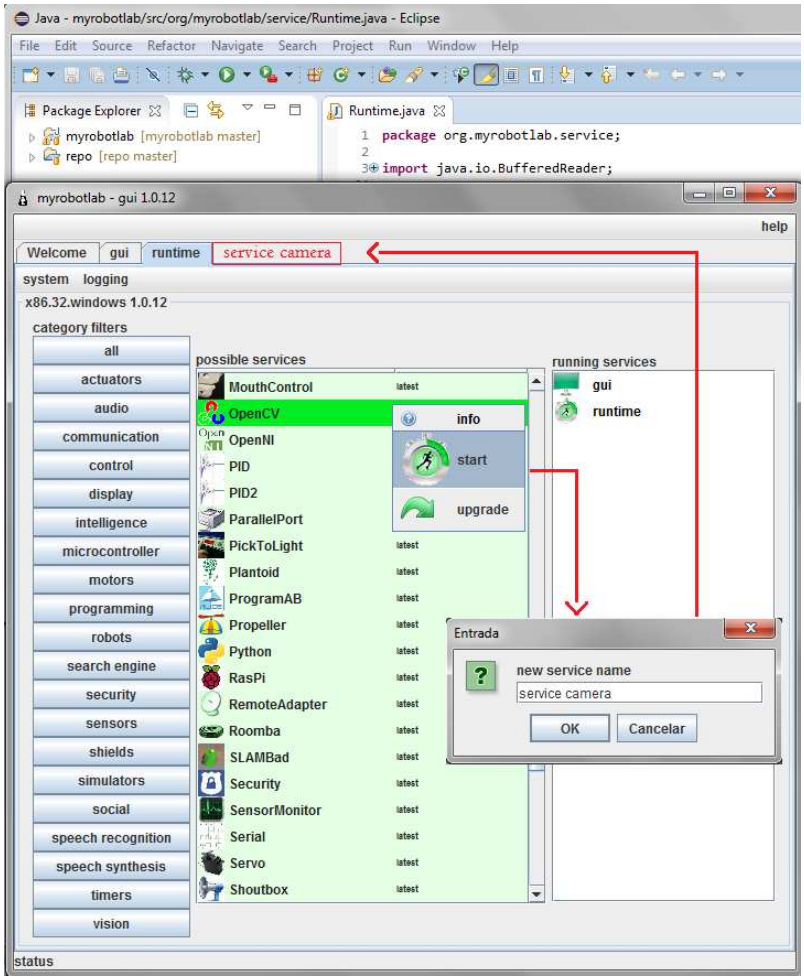


Figura 75: Iniciando um serviço do Frame "runtime". Fonte: Autor.

Para o primeiro exemplo, de um serviço não fornecido, é inicializado o serviço *Raspberry Pi*, a nova aba do *frame "service RasPi"* abre vazia. Para utilizar o serviço ou o que existe dele, é preciso ir ao site e ler as instruções para o serviço não fornecido (MYROBOTLAB, 2014). A Figura 76, apresenta o novo*frame* aberto após a inicialização do serviço.

Esses serviços não fornecidos, quando acessados apresentam um *frame* vazio. A usabilidade e nível de implementação dos serviços instalados do repositório *Myrobotlab*, estão descritos no site do projeto (MYROBOTLAB, 2014), portanto, dependendo do serviço requerido em *"runtime"*, o mesmo pode estar fornecido ou não fornecido, passando para o desenvolvedor a responsabilidade de continuar ou começar do zero a implementação do serviço requerido.

O serviço requerido, neste caso, é um serviço fornecido, com boa implementação e usabilidade no *framework* SOA *Myrobotlab*. O serviço trabalha com a API *Open Source Computer Vision Library (OpenCV)*, a qual possui módulos de processamento de imagens e video I/O, estrutura de dados, álgebra linear, GUI básica com sistema de janelas independentes, controle de mouse e teclado, além de muitos algoritmos de visão com-

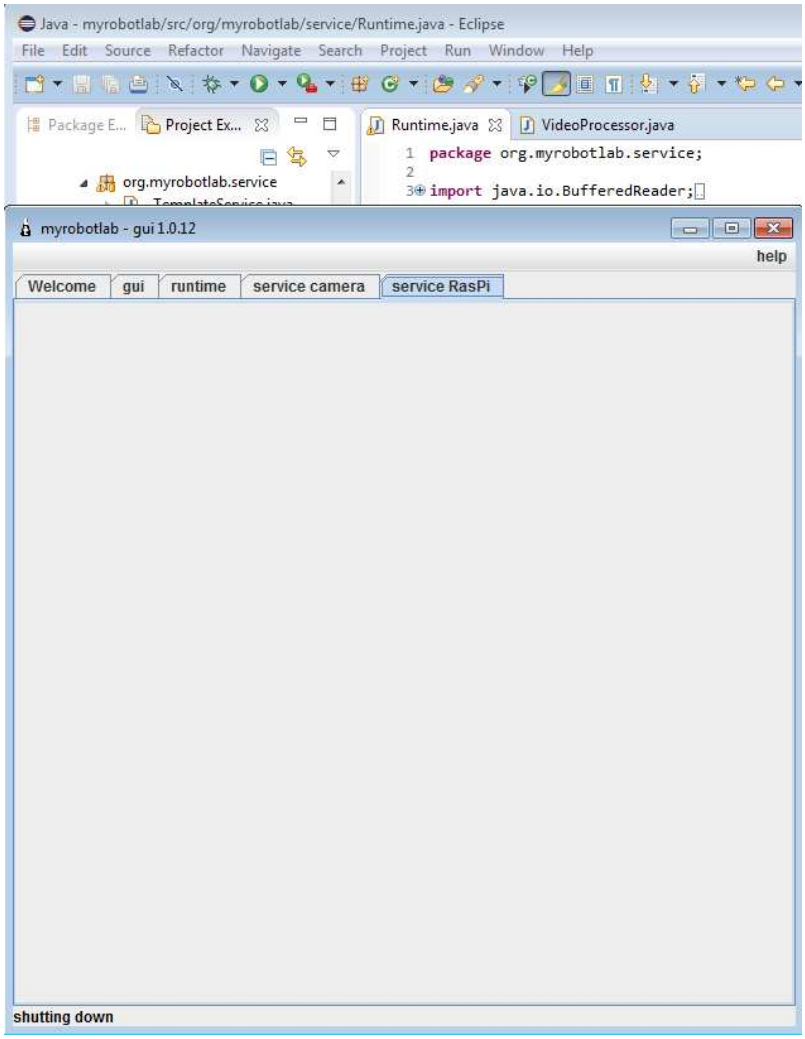


Figura 76: O Serviço "RasPi"tem o *frame* vazio. Fonte: Autor.

putacional como: filtros de imagem, calibração de câmera, reconhecimento de objetos, análise estrutural e outros. O seu processamento é em tempo real de imagens.

A Figura 77, apresenta o serviço inicializado na nova aba do *frame "service camera"*, posteriormente, é preciso fazer a definição da entrada da imagem do serviço *OpenCV*, que poderá ser carregada a partir de um caminho de diretório de um arquivo de imagem existente ou feito através de uma captura de câmera de video. A seguir, a opção (*"file"*) é definida como entrada, a imagem é carregada através do caminho de diretórios da imagem do arquivo fornecido.

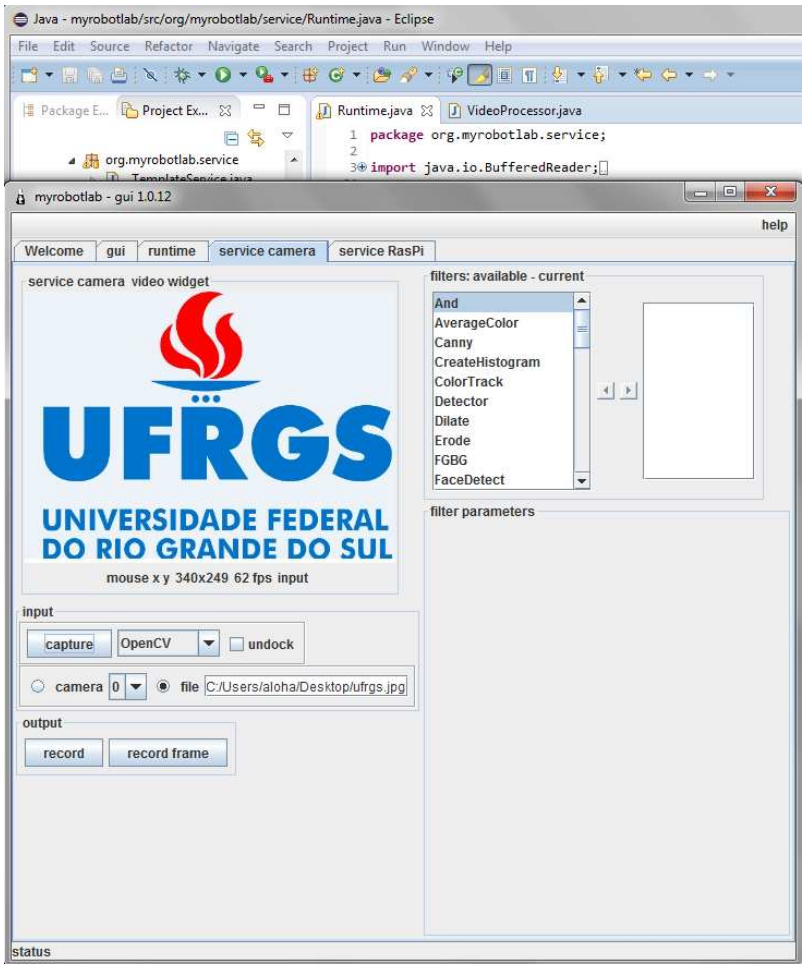


Figura 77: Serviço *OpenCV* executado em uma nova aba frame chamada *"service camera"*.

Na Figura 78, é possível visualizar o filtro *Canny* sendo aplicado na imagem JPEG carregada, *Canny* é um filtro de convolução que usa a primeira derivada, suaviza o ruído e localiza bordas, combinando um operador diferencial com um filtro Gaussiano.

A seguir, é escolhido no frame "service camera", a opção pela captura de uma câmera de video, o *driver* do periférico instala as dependências da API para abrir uma porta virtual de comunicação serial entre o dispositivo e o *framework* SOA *Myrobotlab*. A Figura 79, apresenta a captura da câmera, com o filtro para detecção de face. A face detectada faz parte de um repositório de imagens (FRISCHHOLZ, 2015).

Após iniciar dois serviços, *"service camera"* e *"service RasPi"*, podemos observar a aba do *frame* "gui", responsável por fornecer os serviços atualmente em execução, onde também podem ser passados parâmetros de entradas e saídas para os objetos. A Figura

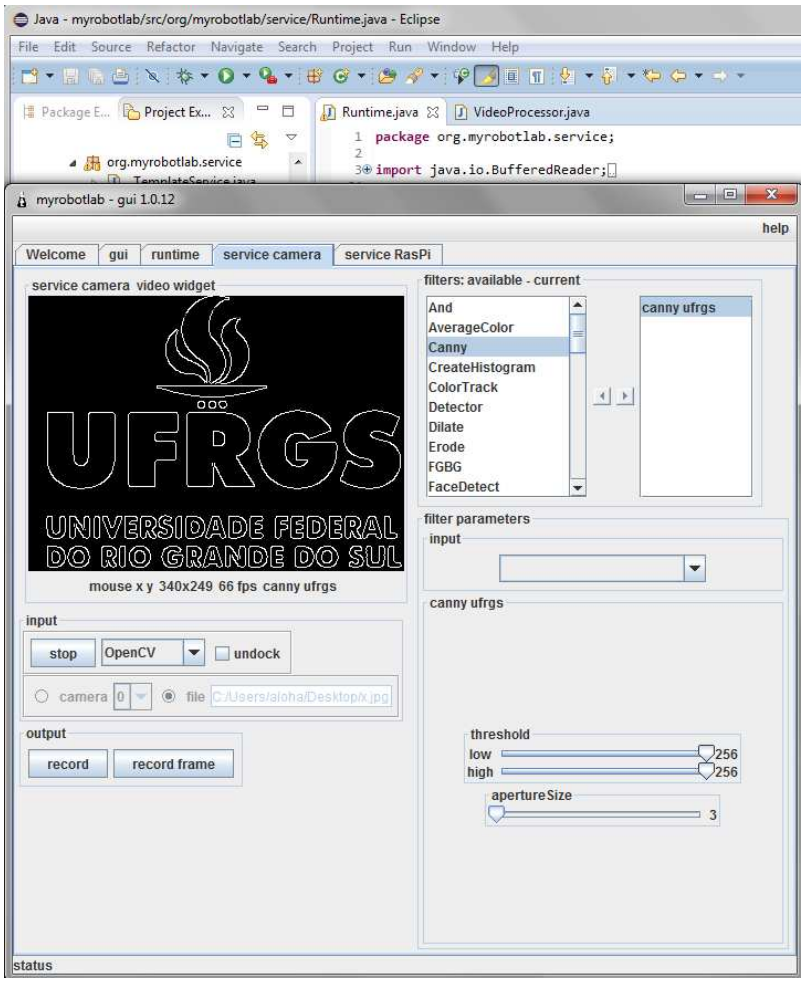


Figura 78: Aplicando o filtro canny no arquivo JPEG. Fonte: Autor.

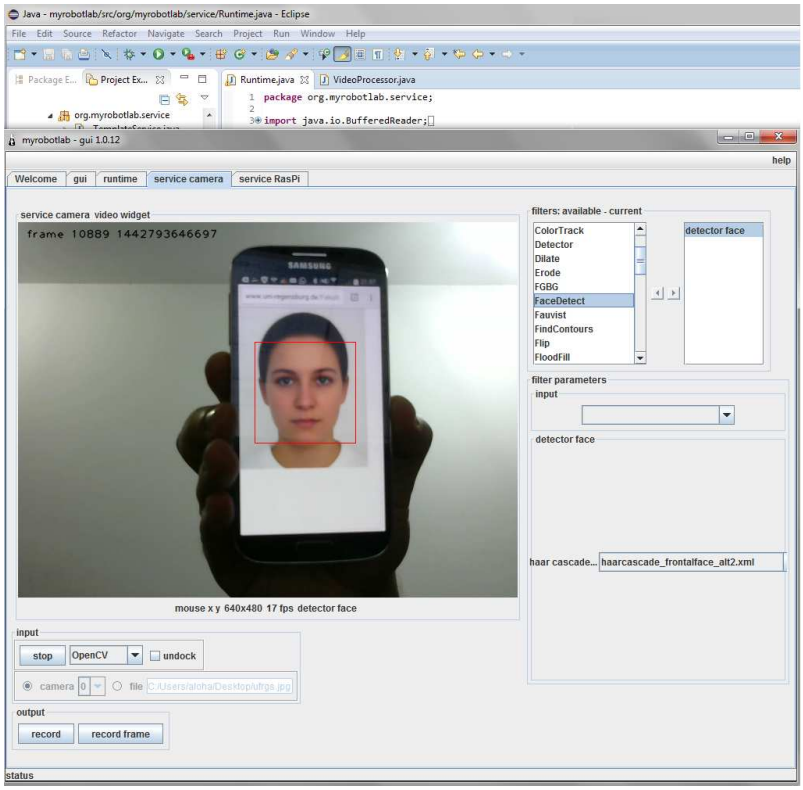


Figura 79: Filtro de detecção de face. Fonte: Autor.

80, apresenta os serviços atualmente executados na aba de controle "gui".

Todas os serviços do sistema *Myrobotlab* necessitam de diversas classes externas tanto do pacote *swing* como do *awt*. Na maioria dos exemplos, os serviços terão pelo menos três linhas com a diretiva *import* apontando para pacotes de classes externas, conforme as declarações seguintes:

- **importjava.awt.*:** Permite a utilização de diversas classes do pacote *awt*, além de possuir uma serie de constantes numéricas.
- **importjava.awt.event:** Usado para o processamento dos eventos que ocorrem na janela, tais como clique do mouse.
- **importjava.swing.*:** Permite a utilização de diversas classes do pacote swing.

Para os *frames* são utilizados à classe *JFrame* disponível no pacote *swing*, a qual terá uma janela com barra de titulo, bordas e pode ter outros componentes visuais (objetos) em seu interior.

Os serviços disponíveis na GUI do *framework* SOA , estão registrados no arquivo *serviceData.xml* do *Myrobotlab*.

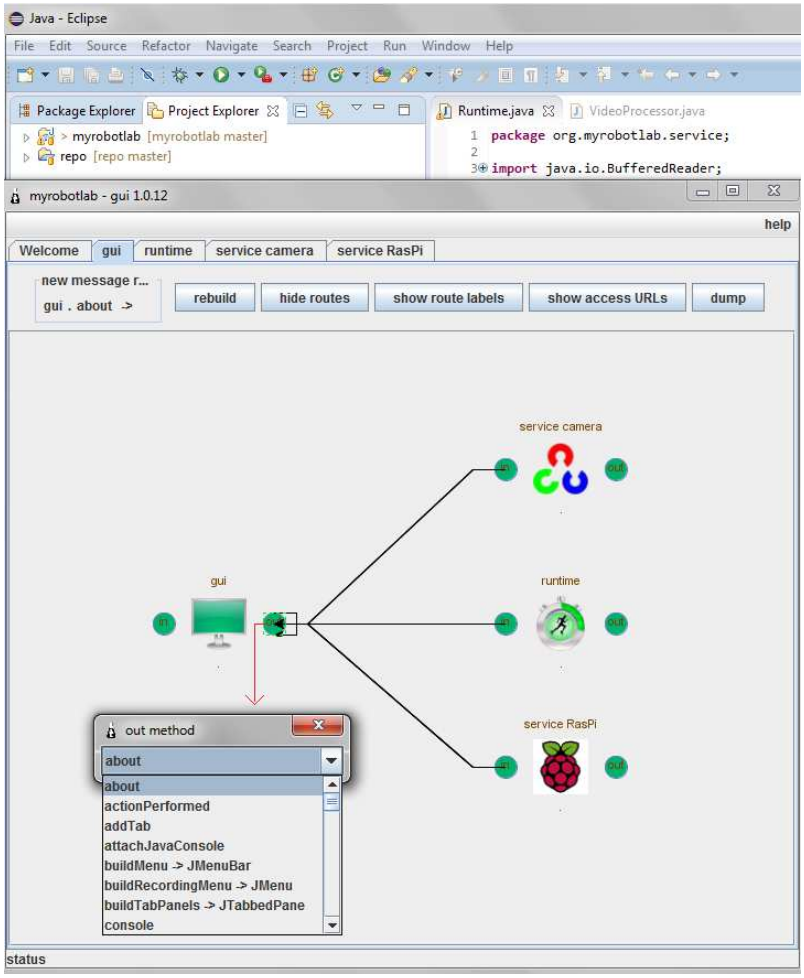


Figura 80: O *Frame* "gui"mostra os serviços atuais em execução. Fonte: Autor.

O XML é uma metalinguagem que define as regras para criar as linguagens de *"markup"* para codificar exemplos de documentos particulares ou tipos de mensagens. A especificação formal para qualquer linguagem "markup"definida utilizando XML ou SGML chama-se Document Type Definition (DTD). É uma maneira de organizar informações, os documentos XML podem ser facilmente compreendidos por programadores facilitando o desenvolvimento de aplicativos compatíveis. Todas as informações contidas no XML estão dentro de tags. O arquivo *serviceData.xml* tem uma importante característica no sistema: permite ao desenvolvedor ter acesso as definição dos serviços existentes ou criar seus próprios serviços, o qual devem ser implementados. A seguir pode ser observados as *tags* utilizadas para os serviços "OpenCV"e "RasPi"(plataforma de *hardware*) do *Myrobotlab*. Este arquivo é utilizado pela classe *"GUIService.java"*

serviceData.xml

```
1
2  <ServiceData>
3     <!-- n types of services -->
4     <serviceTypes>
5                :
6        <serviceType name="org.myrobotlab.service.OpenCV" description="
             The OpenCV Service is a library of vision functions">
7           <dependencies>
8                 <org>com.googlecode.javacv</org>
9                 <org>net.sourceforge.opencv</org>
10          </dependencies>
11       </serviceType>
12             :
13       <serviceType name="org.myrobotlab.service.RasPi" description="a
             service to allow access to the Raspberry Pi&amp;apos;s GPIO and
             I2C offered in Pi4J">
14          <dependencies>
15                <org>com.pi4j.pi4j</org>
16          </dependencies>
17       </serviceType>
18             :
19    </serviceTypes>
20    <!-- n categories -->
21    <categories>
22             :
23       <category name="vision">
24             <serviceTypes>org.myrobotlab.service.OpenCV</serviceTypes>
25       </category>
26             :
27       <category name="microcontroller">
28             <serviceTypes>org.myrobotlab.service.RasPi</serviceTypes>
29       </category>
30             :
31    </category>
32    <!-- n third party libs -->
33    <thirdPartyLibs>
34             :
35        <lib org="com.pi4j.pi4j" revision="0.0.5" resolved="false"
             released="true"/>
36          :
37        <lib org="net.sourceforge.opencv" revision="2.4.6" resolved="
             false" released="true"/>
38          :
```

```
39    </thirdPartyLibs>
40 </ServiceData>
```

É nesse arquivo que estão registrados todos os serviços existentes do sistema. Ele permite adicionar serviços existentes ou adicionar novos serviços. No caso de novos serviços é preciso desenvolver o pacote com os componentes, interfaces fornecidas e requeridas, bem como ajeitar as dependências ocorridas.

O sistema de execução da máquina virtual Java interage com o sistema operacional subjacente. Tais interações incluem executar outros programas, encerrar o sistema de execução, ler e escrever as propriedades do sistema que permitem a comunicação entre sistema operacional e o sistema de execução. Três classes principais em *java.lang*, pacote que contém as classes que constituem recursos básicos da linguagem, necessários à execução de qualquer programa Java, providenciam este acesso, descritos a seguir:

- A classe *System* - providencia métodos estáticos para manipular o estado do sistema. Ela permite a leitura e a escrita das propriedades do sistema, fornece os *streams* padrão de entrada e saída e fornece diversas funções utilitárias. Por conveniência, diversos métodos em *System* operam sobre o objeto *Runtime* corrente.

- A classe *Runtime* - providencia uma interface para o sistema de execução da máquina virtual que está sendo executada. O objeto *Runtime* corrente providencia acesso a funcionalidades de cada sistema de execução, tais como interagir com o coletor de lixo, executar outros programas e desligar o sistema de execução.

- A classe *Process* - representa um processo em execução que foi criado chamando *Runtime.exec* para executar um outro programa, ou através do uso direto de um objeto *ProcessBuilder*.

A API do *Myrobotlab*, não são compostas apenas de classes, métodos e o javadoc usado para documentá-lo. O javadoc é uma importante ferramenta que fornece uma boa visibilidade da API, porém ainda assim uma API é um termo muito mais amplo do que essa visibilidade poderia indicar. Uma API inclui muitos outros tipos de interfaces. A interface é um recurso muito utilizado em Java, bem como na maioria das linguagens orientadas a objeto, para "obrigar"a um determinado grupo de classes a ter métodos ou propriedades em comum para existir em um determinado contexto, contudo os métodos podem ser implementados em cada classe de uma maneira diferente. Pode-se dizer, a grosso modo, que uma interface é um contrato que quando assumido por uma classe deve ser implementado. Dentro das interfaces existem somente assinaturas de métodos e propriedades, cabendo à classe que a utilizará realizar a implementação das assinaturas, dando comportamentos práticos aos métodos. Para realizar a chamada ou referência a uma interface por uma determinada classe, é necessário adicionar a palavra-chave *implements* ao final da assinatura da classe que irá implementar a interface escolhida, com a seguinte sintaxe:

- *public class* nome_classe *implements* nome_interface

Onde nome_classe é o nome da classe a ser implementada e nome_Interface é o nome da interface a ser implementada pela classe. Diferentes classes podem usar classes interfaces, porém cada classe irá implementar seus métodos de maneira diferente. Ao contrário da herança que limita uma classe a herdar somente uma classe pai por vez, é possível que

uma classe implemente varias interfaces ao mesmo tempo. Uma classe interface no *Myrobotlab*, nada mais é que uma espécie de contrato de regras que uma classe deve seguir em um determinado contexto. Como em Java não existe herança múltipla, a interface passa a ser uma alternativa.

5.4.2 Integração do *Framework* SOA com o Robô móvel Acionamento Diferencial

Para integração entre o estudo de caso do robô móvel proposto com o *framework* SOA *Myrobotlab*, foi utilizado a arquitetura de camadas apresentada na subseção 6.1, o qual é útil em organizações que utilizam variados tipos de sistemas, com diferentes tecnologias e fornecedores, sendo imprescindível o atendimento a requisitos como escalabilidade, flexibilidade e interoperabilidade. Para usufruir dos benefícios desta arquitetura, é necessário investimento de tempo e aprendizado. Através do uso de SOA o entendimento entre os componentes de serviços. A Figura 81, apresenta a tela inicial do *framework* SOA, o qual vem sendo estendido e integrado com a plataforma de *hardware* do robô móvel para aplicações com estudantes de graduação de Engenharia da UFRGS.

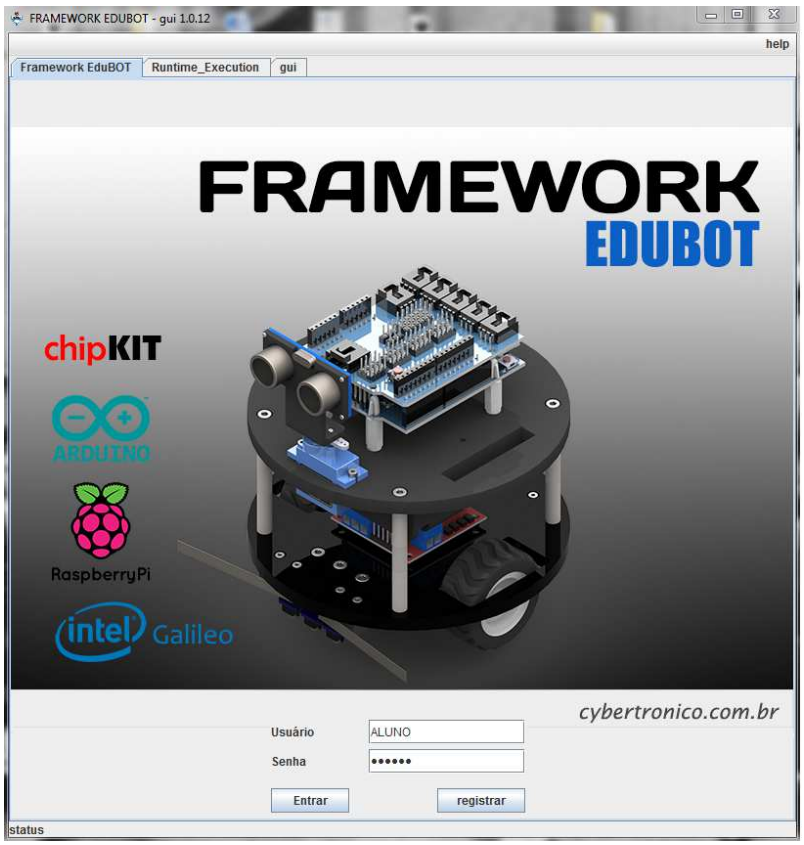


Figura 81: Tela inicial do novo Framework SOA estendido. Fonte: Autor.

A próxima tela, da Figura 82, apresenta os principais itens do SOA que são os serviços que servem para denominar o relacionamento entre um provedor e um consumidor, que possuem o objetivo de solucionar uma determinada atividade em comum. Estão sendo utilizados e implementados alguns serviços previamente escolhidos conforme os requisitos de projeto, cada serviço pode ser definido como uma atividade específica, devido a identificação dos serviços encontrados e escolhidos conforme a aplicação.

A seguir, é apresentado na Figura 83, o serviço que vem sendo criado para interação com a plataforma de *hardware* do robô móvel, nesse caso um serviço para o *"Intel Galileo*

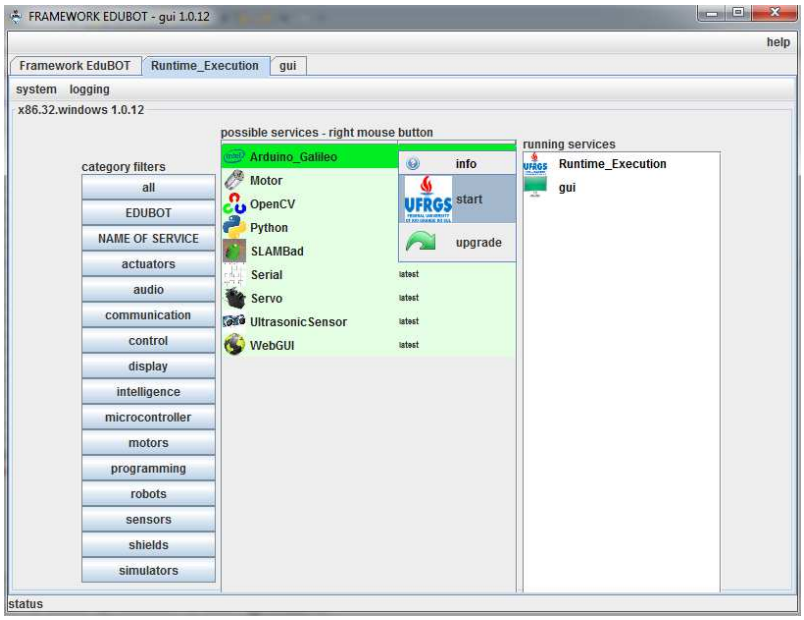


Figura 82: Tela de serviços implementados para utilizar com estudantes de graduação da UFRGS. Fonte: Autor.

Gen 2", a GUI do serviço possui três opções: "oscope", "pins"e "editor".

A opção osciloscópio permite a visualização de sinais periódicos tais como ondas quadradas e ondas seno. Já a opção pinos, permite a interação do usuário com os pinos do *hardware*. E por fim, a opção de edição, permite criar ou editar códigos, fazer compilação e *uploading* do *firmware* para o *hardware*, bem como acessar as bibliotecas ou API do microcontrolador via menu, as bibliotecas e o *"core"* do *framework* Arduino Intel fazem parte dos pacotes embarcados dentro do *framework* SOA *Myrobotlab* estendido. O *framework Wiring* C/C++ encapsula os registradores de baixo nível dos microntroladores ou SOC's para criar uma API de alto nível.

Importante ressaltar, que para utilizar a GUI do serviço Arduino com as opções osciloscópio e pinos, um *firmata* chamado "MRLComm.ino"precisa ser carregado no microcontrolador ou SOC. O código é um protocolo genérico que permite a comunicação do computador hospedeiro com o microcontrolador via software. A Figura 84, apresenta o diagrama de blocos que representa a conexão entre o computador hospedeiro com microcontrolador ou SOC. Com o *firmata* rodando no microcontrolador conectado ao *framework* SOA *Myrobotlab*, é possível utilizar as opções de controle dos pinos I/O e osciloscópio.

Atualmente, o serviço vem sendo implementado para fazer a adaptação e empacotamento do *"core"* do *framework* Arduino Intel. Para fazer a utilização da API de programação dos registradores do *hardware* e do compilador para o *Intel Galileo Gen 2*. Caso o *firmata* "MRLComm.ino"esteja embarcado no SOC ou microcontrolador, o mesmo se torna um *hardware slave*, assim o serviço Arduino "Intel Galileo"do *framework* SOA se comunica e controla o *hardware*. Caso contrario, o serviço Arduino "Intel Galileo"apenas edita, compila e carrega *firmwares* para a plataforma de *hardware*.

O *firmata* só tem de ser carregado uma vez no *hardware*. A única vez que teria de ser re-carregado é quando ele precisa ser atualizado. Depois de carregado o *firmata* no microcontrolador e inicializado o serviço Arduino no *Myrobotlab*, se estabelece-rá uma conexão entre o *software* e o *hardware*, o qual se dará através da porta de comunicação

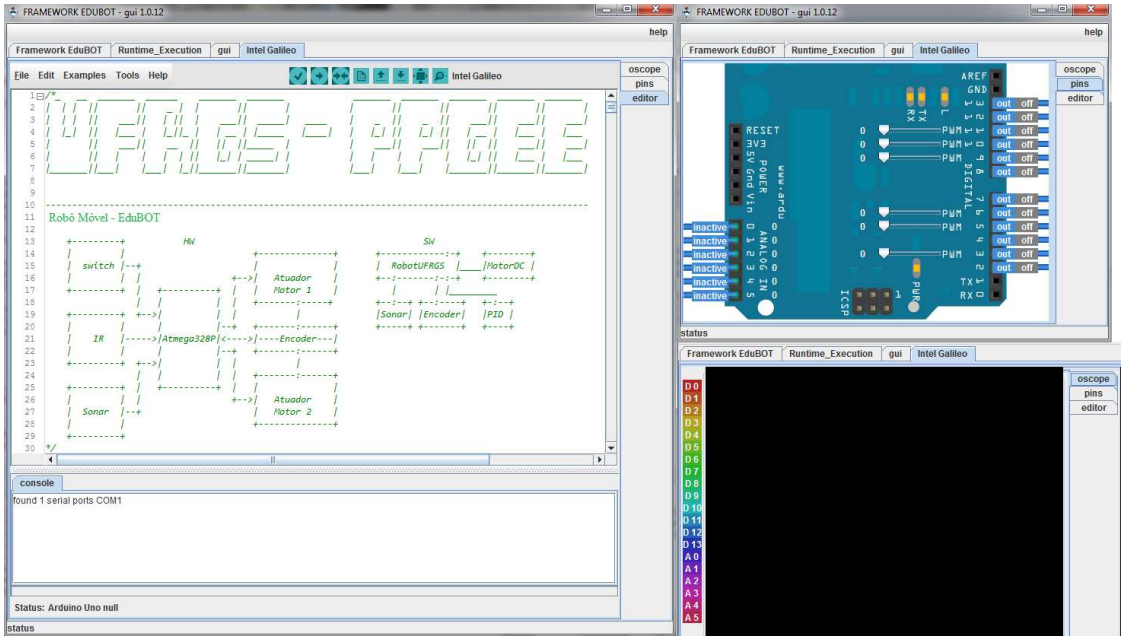


Figura 83: Tela do serviço para interação com a plataforma de *hardware*. Fonte: Autor.

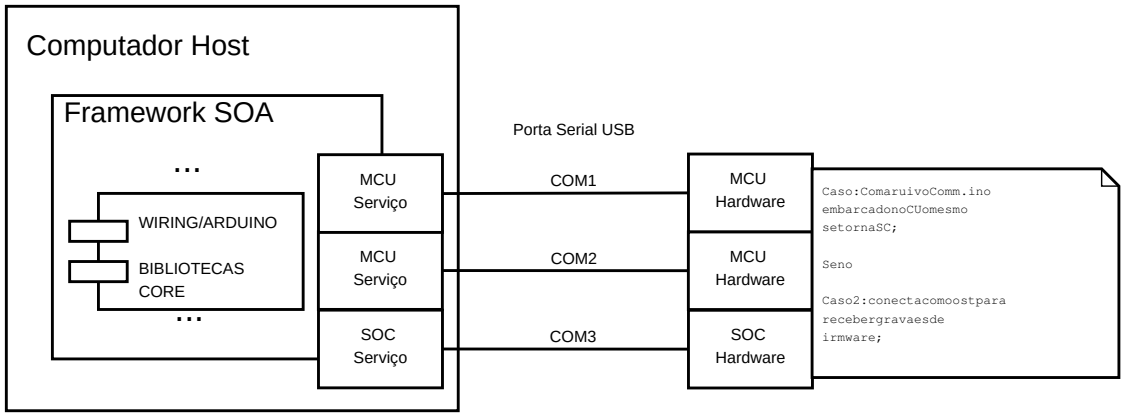


Figura 84: Diagrama de blocos com o computador hospedeiro rodando o *framework* SOA *Myrobotlab* com o serviço Arduino conectado com o *hardware*. Fonte: Autor.

serial apropriada. Se o *firmata* "MRLComm.ino"está atualizado, deve ser impresso a mensagem *"goodtimes"* na tela da parte inferior da GUI do serviço Arduino.

Desse modo é possível que comandos ou dados enviados a partir do *framework* SOA *MyRobotLab* sejam processados no *hardware slave* e também permite a aquisição dos dados do *hardware slave* para o *Myrobotlab*.

```
/**
* @author GroG (at) myrobotlab.org
* This file is part of MyRobotLab.
* MRLComm.ino
* -------------------
* Purpose: support servos, sensors, analog & digital polling
* oscope, motors, range sensors, pingdar & steppers.
* Requirements: MyRobotLab running on a computer & a serial connection
*/
#include <Servo.h>
#include<MRLComm.h>


MRLComm mrl = MRLComm();

void setup() {

    Serial.begin(57600);
    mrl.setup(&Serial);
}

void loop() {

  // process mrl commands
  // servo control, sensor, control send & recieve
  // oscope, analog polling, digital polling etc..
  // mrl messages
  mrl.process();

  // example how to
  // send 3 vars to mrl
  /*
  int ax = 28;
  int ay = 583;
  int az = 32767;

  mrl.startMsg();
  mrl.append(ax);
  mrl.append(ay);
  mrl.append(az);
  mrl.sendMsg();
  */

}
 }
```

Por sua vez, o robô móvel foi modelado em um software de desenho CAD, que baseia-se em computação paramétrica, criando formas tridimensionais a partir de formas geométricas elementares (SOLIDWORKS, 2014). No ambiente do programa, a criação de um

sólido ou superfície tipicamente começa com a definição de topologia em um esboço 2D ou 3D. Assim, vem sendo desenvolvido o desenho mecânico, eletromecânico e eletrônico do novo EduBOT v0.4. O projeto mecânico vem sendo desenvolvido e aperfeiçoado conforme as validações dos testes reais e de simulações, recentemente a plataforma robótica educacional vem sendo projetado para suportar a placa Intel Galileo Gen 2. Na Figura 85 é apresentado o desenho 3D do protótipo do robô móvel com direção diferencial. O desenho do chassi é mapeado para os elementos mecânicos e eletromecânicos como: motores,sensores, rodas, baterias, parafusos e demais componentes de *hardware* que são projetados e desenhados em software CAD de maneira modular e bem organizada. Para automatizar a fase de projeto, alguns dos componentes de hardware CAD (mecânicos e eletromecânicos) são projetados e alguns componentes reutilizados e adaptados, podendo-se tornar um projeto simples ou complexo dependendo dos requisitos necessários para a aplicação.

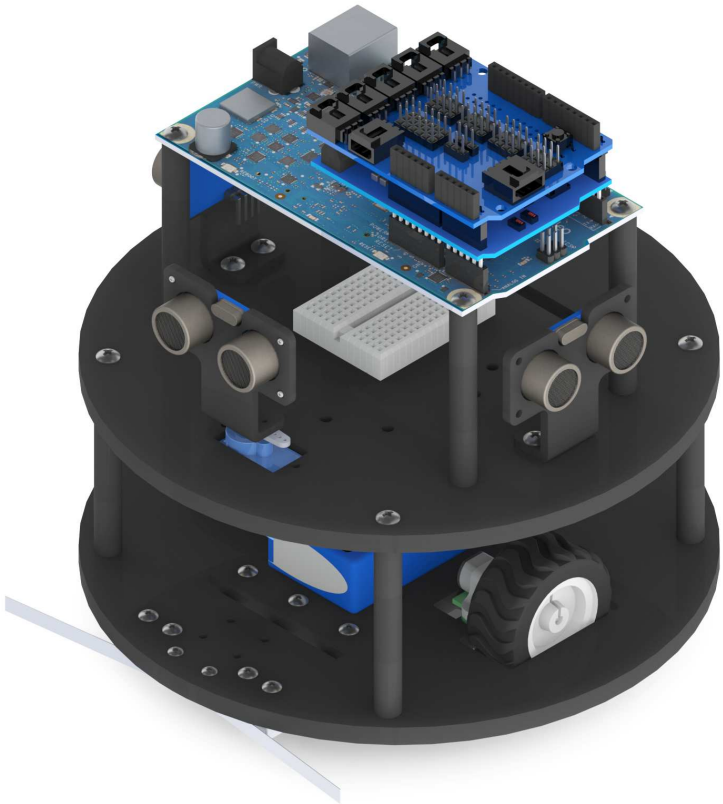


Figura 85: Plataforma Robótica Educacional EduRobot v0.4.

O projeto do *hardware* da plataforma EduBOT v0.4, vem sendo projetado para ter chassis com diâmetro 180mm de diâmetro, o que resulta em um robô móvel com tamanho flexível o suficiente para diferentes aplicações educacionais. A Figura 86, apresenta algumas das dimensões do desenvolvimento do projeto.

O projeto CAD faz o mapeamento dos componentes utilizados no desenho do projeto. Existem uma grande variedade de componentes mecânicos e eletromecânicos que podem ser utilizados para o protótipo. Vale ressaltar, que as escolhas dos componentes mecânicos e eletromecânicos utilizados no projeto, se deu através de pesquisas de diferentes tipos de componentes, priorizando aqueles que possuem qualidade, alta coesão, baixo custo e acoplamento. Para esta nova plataforma robótica estão sendo desenvolvido as bibliotecas em C++ para os componentes eletrônicos utilizados com o *hardware Intel Galileo Gen 2*,

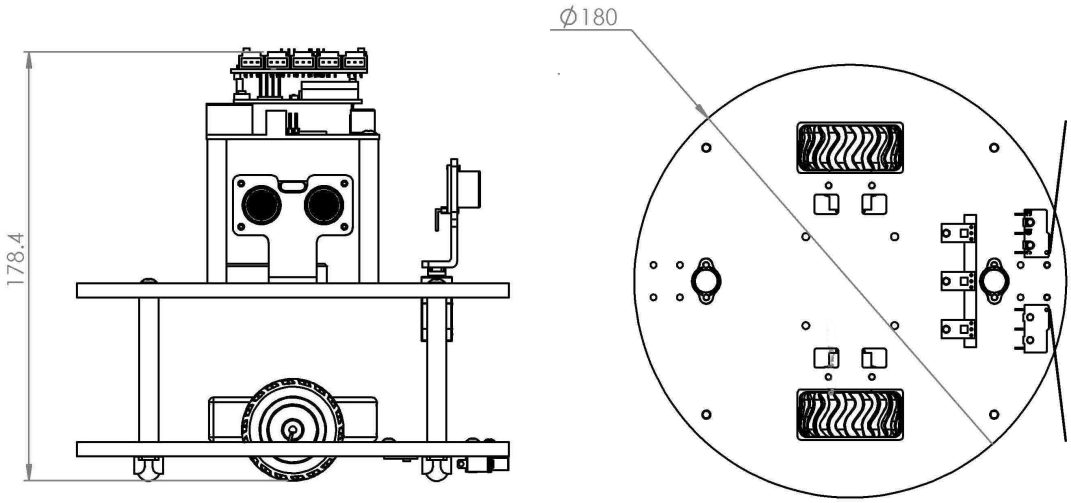


Figura 86: Principais dimensões do EduBOT v0.4.

inicialmente será apresentado a estrutura parcial do arquivo da classe "DCMotorBot.h".

```
/*
 @author Carlos Solon
 ------------------------*/
/*Biblioteca para os Motores DC - Intel Galileo Gen 2 - Driver L298N.
   Arquivo .h da classe DCMotorBot*/

#ifndef DCMotorBot_H
#define DCMotorBot_H

// Compatibility for Arduino 1.0

#if ARDUINO >= 100
    #include "Arduino.h"
#else
    #include "WProgram.h"
#endif

class DCMotorBot {

public:

    // construtores
    DCMotorBot(); // construtor vazio
    DCMotorBot(byte e1, byte e2, byte I1, byte I2, int delay, int
       pwmMotors); // Construtor completo

    void setEnablePins(byte e1, byte e2); //setar individual
    void setControlPins(byte I1, byte I2); // setar individual
    void setDelay(int delay); //
    void setPwmAll(int pwmMotors); //

    //Acionar motores de forma crescente e decrescente - PWM
    void fadein();
    void fadeout();

    // Funções para movimentos
    void start();
```

```
36      void moveForward();
37      void forw();
38      void moveBackward();
39      void back();
40      void turnLeft();
41      void turnRight();
42      void stop();
43  private:
44      int mDelay;
45      int mPwm; //pwm
46
47      // pinos para habilitar pwm
48      byte mE1;
49      byte mE2;
50
51      // Pinos de controle
52      byte mI1;
53      byte mI2;
54  };
55  #endif
56  }
```

Depois de implementados os arquivos "DCMotorBot.h"e "DCMotorBot.cpp"é preciso criar um arquivo *"main.ino"*, para instanciar a classe e usar os objetos concatenados com as funções e propriedades da classe dos motores DC para o *framework* Arduino Intel v1.6.x, as bibliotecas implementadas devem estar inseridas dentro do diretório *libraries*.

```
1  /*
2   @author Carlos Solon
3   ------------------------*/
4  /* Arquivo Main para instanciar e utilizar objetos da classe "
       DCMotorBot" */
5
6  #include <DCMotorBot.h>
7
8  //cria objeto
9  DCMotorBot bot;
10
11 void setup() {
12
13      // objeto inicializando os pinos
14      bot.setEnablePins(3, 11);
15      bot.setControlPins(12, 13);
16 }
17
18 void loop() {
19 //objeto acessando os métodos para movimentos
20      bot.moveForward();
21      delay(2000);
22      bot.moveBackward();
23      delay(2000);
24      bot.turnLeft();
25      delay(2000);
26      bot.turnRight();
27      delay(2000);
28 }
```

5.4.3 Integração do *Framework* SOA com a Macro e Micro Navegação da Bengala Eletrônica

Para integração entre o estudo de caso da bengala eletrônica proposto com o *framework* SOA *Myrobotlab*, foi utilizado a arquitetura de camadas apresentada na subseção 6.1. A Figura **??**, apresenta a tela inicial do *framework* SOA, o qual vem sendo estendido e integrado com a plataforma de *hardware* da bengala eletrônica para aplicações com pessoas deficientes visuais. Para os resultados do estudo de caso do projeto da tecnologia assistiva, estão sendo construídos um ambiente para localização de obstáculos e com um sistema de telemetria e telecontrole para geolocalização utilizando o serviço *"WebGUI"* do *framework*, 87.

Figura 87: Tela inicial do novo *framework* SOA estendido. Fonte: Autor.

O sistema de rastreamento global utiliza GPS para fazer aquisições de dados de localização no *framework*, A Figura 88, apresenta o serviço "WebGUI"inicializado na GUI do serviço *"runtime"*.

Os resultados das aquisições dos módulos, sensores e acionamento dos atuadores consistem em objetos instanciados e podem ser visualizados na tela que disponibiliza as informações em tempo real. O receptor GPS informa a latitude e longitude, os dados de posição global são passados para um mapa utilizando o GPS *Visualizer* e API *Google Maps* (GPS, 2014). O sonar informa a distancia de obstáculos em centímetros, caso o obstáculo estiver a uma distancia menor que o limite escolhido, é acionado o atuador de vibração por *Pulse-Width Modulation* (PWM) ou simplesmente modulação por largura de pulso, e por último, o LDR informa o nível de luminosidade do ambiente. O sistema embarcado foi testado na disciplina Sistemas de Automação do PGPEE, UFRGS. As classes

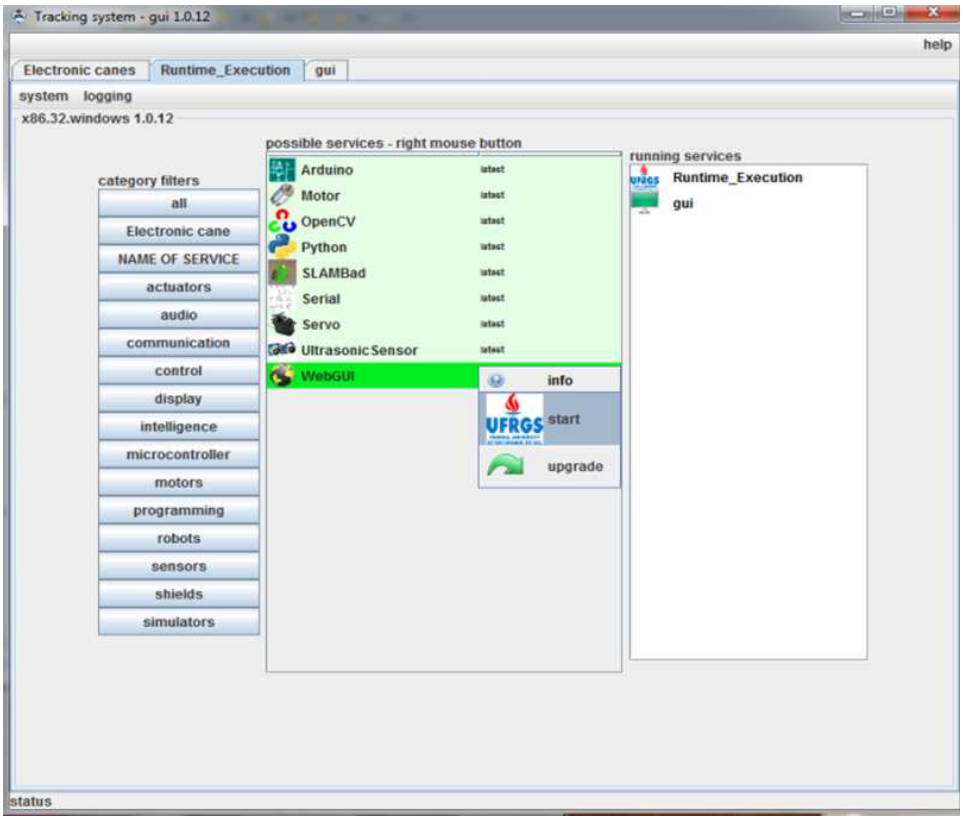


Figura 88: Tela de serviços implementados para utilizar com estudo de caso da bengala eletrônica. Fonte: Autor.

desenvolvidas consistem em bibliotecas em C/C++ e devem ser inseridas dentro da pasta *"libraries"* do *software* Arduino, para executar os testes no programa principal, as classes implementadas utilizam a API da linguagem *Wiring* para o seu desenvolvimento. As bibliotecas estão sendo testadas também nos ambientes de programação *Atmel Studio* e Eclipse. A comunicação serial entre o protótipo do *hardware* e o computador iniciam as amostras dos resultados em tempo real através do *"Serial Monitor"* do *software* que permite que dados simples de texto sejam enviados e recebidos, Figura 89.

O rastreador GPS recebe todas essas informações e as transmite, utilizando as tecnologias GSM e GPRS, padrões digitais de comunicação celular de voz e dados, respectivamente, para os servidores da central de rastreamento, que disponibiliza, via web, todas as informações em tempo real, para os clientes. A Figura 90, apresenta o localização global do sistema embarcado.

Para testes iniciais, foram desenvolvidas algumas bibliotecas referentes aos dispositivos utilizados no projeto. Cada classe é desenvolvida conforme os requisitos dos componentes eletrônicos. Uma classe principal (*main.ino*) deve ser criada para instanciar os objetos das classes implementadas. A seguir é apresentado o código do arquivo *main.ino* criado para testar as bibliotecas do software do projeto.

```
1 /*
2  @author Carlos Solon
3  ------------------------------*/
4 /* Arquivo Main criado para instanciar as classes e criar objetos das
     bibliotecas implementados em C++, do projeto Bengala Eletrônica */
5
6 #include "ControleBengala.h" // Biblioteca de controle do projeto
```

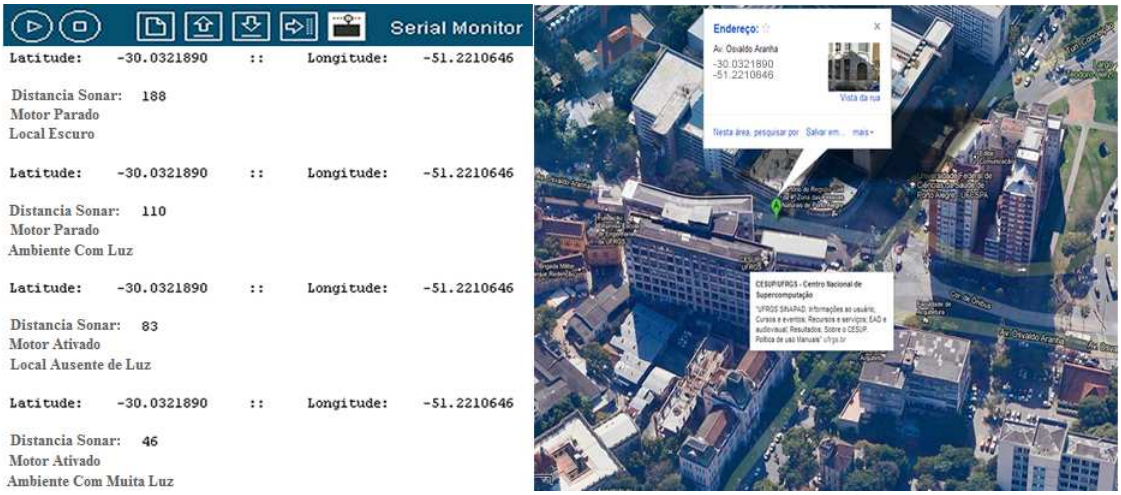


Figura 89: Bibliotecas C++ rodando com API de mapas. Fonte: Autor.

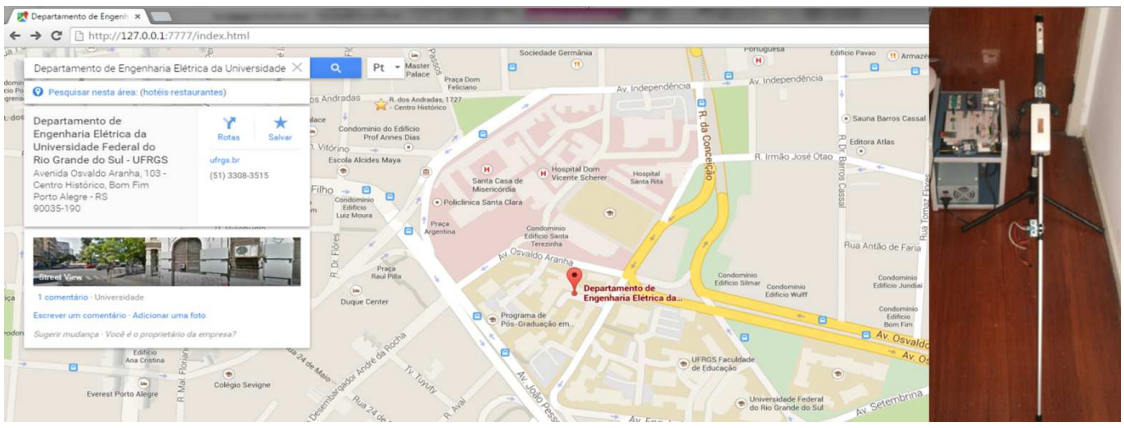


Figura 90: *Front End aberto após inicializar o serviço "WebGUI"* . Fonte: Autor.

```
7  #include "Sonar.h" //Biblioteca do Sensor de obstáculos do projeto
8  #include "Luz.h" //Biblioteca do Sensor de luz do projeto
9  #include "SkyGPS.h" //Biblioteca do receptor GPS do projeto
10 //#include "AUDIO.h"
11 #include <NewSoftSerial.h> //Biblioteca do arduino, cria interface
       virtual via software em qualquer par de pinos digitais da placa
       sendo um para Rx e o outro Tx. O RX e TX do uC do arduino esta
       sendo usado para comunicação serial entre a placa e o computador
       precisando simular um par de pinos para o RX e TX do GPS com o
       Arduino.
12 #include <cstdlib> //Biblioteca  cstandard library do c++, possui
       funções de alocação de memória, controle de processos, conversões,
       etc...
13 #include <iostream> //declara as quatro classes básicas de entrada e
       saída: ios, istream, ostream e iostream
14
15 #define ldrPin A0 // pino do Sensor de Luz
16
17 #define pinMOTOR 9 // pino pwm do uC Atmega328 do arduino para o motor
       de vibração;
18
19 // Pinos do Sonar, receptor e transmissor;
20 #define ECHO 7
21 #define TRIGGER 8
22
23 //GPS pinos
24 #define RX 3
25 #define TX 2
26
27 //#define pinSOM 6  pino para um buzzer;
28
29 #define velo 255 // valores entre 0 e 255 para controlar a tensão da
       bateria, pino pwm;
30
31 #define time 0 // variável para valor de Delay;
32
33 using namespace std;
34
35 String nome = "Nome Do Usuário";
36
37 long distancia;     //auxiliares
38 int ldr, ctrl, nivel;
39
40 unsigned float fix_age;   //variáveis para manipular o gps
41 float LAT=0, LON=0;
42
43 NewSoftSerial GPS(RX, TX);  //interface virtual para os pinos rx e tx
       do uC
44
45 void setup(void) { //Método da API Wiring
46   GPS.begin(9600); // inicia comunicação de serie do gps:baud rate de
         9600
47    Serial.begin(115200);//inicia comunicação de serie do arduino com o
         computador:baud rate de 115200
48 }
49
50 int main(int argc, char *argv[])
51 {
```

```
52   long lat, lon;
53   unsigned long fix_age, speed, course;
54   unsigned long chars;
55   unsigned short sentences, failed_checksum;
56
57   setup();
58
59 // objetos das classes que recebem como parâmetros em seus construtores
       os pinos de sensores e atuadores
60
61   Motor vibracao = Motor (pinMOTOR);
62   Sonar obstaculos = Sonar(ECHO, TRIGGER);
63   Luz intensidade = Luz(ldrPin);
64   ControleBengala controle = ControleBengala(nome); // recebe o nome do
         usuario como parametro
65   SkyGPS gps = SkyGPS(RX, TX);
66
67   while (1)
68   {
69
70 //Sensor sonar
71   obstaculos.distancia(); //objeto obstáculos concatena o metodo
       distancia() para acionando do transmissor do sonar e seu receptor
        retornando um valor em centímetro;
72   obstaculos.sonarPing(); // o valor da distancia do método distancia()
        é armazenada no método sonarPing(), com o método getDistancia é
       dado o retorno do valor armazenado em sonarPing();
73
74   distancia = obstaculos.getDistancia(); // Retorna a distancia em
       centímetros
75
76   Serial.print("Distancia Sonar: "); //escreve na tela
77   Serial.println(distancia); // a distancia dos obstáculos detectados
       em centímetros armazenados no método sonarPing;
78
79 //Sensor sonar
80   ctrl = controle.verificaColisao(distancia); // objeto controle
       concatena método verificaColisao()
81 //recebe como parâmetro a distancia dos obstáculos
82         if(ctrl==0)// leitura de erro na diagonal do sonar
83         {
84            vibracao.desligaMotor(); //desliga o motor de vibração
85            Serial.println("Motor Parado"); // escreve na tela
86         }
87         if(ctrl==1) //objeto próximo, menor que 50 centímetros a
              principio
88         {
89            vibracao.acionaMotor(velo,time);
90            Serial.println("Motor Ativado");
91         }
92         else // objeto afastado, fora do ambiente de ação
93         {
94            vibracao.desligaMotor();
95            Serial.println("Motor Parado");
96         }
97 //Sensor de Luz LDR
98   intensidade.luminosidade(); //objeto intensidade concatena o método
       luminosidade()  e faz a leitura do sensor LDR
```

```
99  // retornando valores da resistência do sensor;
100   intensidade.luzPing();// os valores lidos do método luminosidade é
        armazenada no método luzPing(), mais tarde é chamado o método
        getLuz que retorna o valor armazenado em "luzPing()";
101
102   nivel = intensidade.getLuz(); // objeto intensidade concatena método
        getLuz() que retorna valores da leitura do LDR
103
104 // objeto controle concatena método verificaLuminosidade() que recebe
      como parâmetro os valores de leitura da resistência do sensor no
      ambiente; verifica o nível de intensidade da luz com o retorno do
      método verificaLuminosidade()
105
106   ldr = controle.verificaLuminosidade(nivel);
107         if(ldr==1)
108                   Serial.println("\nAlerta: Local Ausente de Luz");
109         if(ldr==2)
110                   Serial.println("\nLocal Escuro");
111         if(ldr==3)
112                   Serial.println("\nAmbiente Com Luz");
113         if(ldr==4)
114                   Serial.println("\nAmbiente Com Muita Luz");
115         else
116                   Serial.println("\nAlerta: Ambiente Brilhante");
117
118 //receptor GPS -
119         getGPS(); //Faz chamada ao método get,
120         Serial.print("\n");
121         Serial.print("Latitude : "); // Escreve na tela
122         Serial.print(LAT/100000,7); //latitude
123         Serial.print(" :: Longitude : ");
124         Serial.println(LON/100000,7); //longitude
125         Serial.print("\n");
126
127         delay(500);
128   }
129 }
130 void getGPS(){
131   bool newdata = false;
132   unsigned long start = millis();
133   // A cada 1 segundo imprime uma atualização
134   while (millis() - start < 1000)
135   {
136     if (feedgps ()){
137       newdata = true;
138     }
139   }
140   if (newdata)
141   {
142     gpsdump(gps);
143   }
144 }
145 bool feedgps(){
146   while (GPS.available())
147   {
148     if (gps.encode(GPS.read()))
149       return true;
150   }
```

```
151   return 0;
152 }
153 void gpsdump(TinyGPS &gps)
154 {
155   // byte dia, mês, hora, minuto, segundo, centésimos;
156   gps.get_position(&lat, &lon);
157   LAT = lat;
158   LON = lon;
159   {
160     feedgps(); // Se não alimentar o gps durante esta longa rotina,
          podemos obter erros de checksum
161   }
162 }
```

Os resultados dos valores das aquisições dos sensores e acionamento dos atuadores consistem em objetos instanciados e podem ser visualizados em uma tela em texto que disponibiliza as informações de tempo real do teste. O receptor GPS informa a latitude e longitude, os dados de posição global são passados para um mapa utilizando o *"GPS Visualizer"* para testes iniciais. O sonar informa a distancia de obstáculos em centímetros, caso o obstáculo estiver a uma distancia menor que o limite escolhido, é acionado o micromotor CC, e por último, o LDR informa o nível de luminosidade do ambiente. O sistema embarcado foi testado no PGPEE - UFRGS.

6 CONCLUSÃO E TRABALHOS FUTUROS

Conforme a integração do *framework* SOA com os estudos de casos, foram analisadas questões como a gerenciamento das análises e equipamentos, bem como a escalabilidade da solução. Também foram testadas as entidades implementadas, verificando a interoperabilidade entre os diferentes componentes de *softwares* do projeto. Alguns experimentos visam comprovar as vantagens na utilização da proposta SOA definida neste estudo. Construir um *framework* SOA, assim como todo sistema grande e complexo, exige um planejamento e uma modelagem prévia. Esse planejamento leva à construção de uma arquitetura, definindo módulos, componentes e suas interações. Com este trabalho, espera-se facilitar o desenvolvimento de sistemas embarcados para a comunidade de robótica e criar um *framework* SOA que possa ser utilizado por professores e alunos em aulas da graduação de Engenharia da UFRGS. Fornecendo o manual e código sobre licença aberta, será dado a qualquer desenvolvedor o direito de implementar este *framework* em sua aplicação e, com isso, diminuindo pelo menos um pouco o retrabalho de construir um projeto de robótica e automação. Dividir o projeto em grupos menores ajudou a organizar o código, garantir que todas as funcionalidades estão desenvolvidas e diminuir o acoplamento entre as partes. Ao definir o que é visível em um módulo e como operá-lo (interface) e como os componentes se comunicam fica mais fácil testar, isolar funcionalidades e garantir o bom funcionamento de cada parte do sistema e dele como um todo. Os resultados iniciais deste projeto são as modelagens de projeto e a integração dos equipamentos mecânicos, eletromecânicos e de computação com o *framework* SOA, bem como as implementações e adaptações de bibliotecas, protocolos de comunicação, sensores, atuadores e arquiteturas de controle (PID - proporcional integral derivativo) que são fornecidos em forma de serviço no *Framework* SOA. Trabalhos futuros incluem continuar a implementação dos estudos de caso e do *framework* SOA. *Threads*, processamento de imagens, áudio e identificação por radio frequência também estão sendo investigados como futuros dispositivos para o projeto. Também esta sendo estudado a substituição do microprocessador ATmega328P pelo ARM Cortex-M3 (32-bit) para ter um melhor processamento nos sistemas embarcados. Com o sistema de rastreamento, o objetivo é implementar uma central de localização de bengalas eletrônicas, através dos dados enviados a um servidor central que irá converter os dados de localização em posições em um mapa, para criar mapas, rotas de condução, endereços ou coordenadas simples a partir de dados GPS. Os testes finais mostraram que ainda há espaço para melhorias no projeto em geral.

REFERÊNCIAS

ATMEL. **Datasheet ATmega328P**. Disponível em: <http://www.atmel.com/pt/br/devices/atmega328p.aspx>. Acesso em: 21 Jan. 2015.

BAZANELLA, A. S.; GOMES DA SILVA, J. M. **Sistemas de Controle**: princípios e métodos de projeto. Porto Alegre: UFRGS, 2005.

Da Silva, P. C. B. **Utilizando UML**: diagramas de implantação, comunicação e tempo. Disponível em: <http://www.devmedia.com.br/revista-sql-magazine-134/33353>. Acesso em: 14 Set. 2015.

EAGLE, C. **CadSoft EAGLE PCB Design Software**. Disponível em: <http://www.cadsoftusa.com/>. Acesso em: 12 Jul. 2014.

EDUBOT. **Protótipo de uma plataforma robótica livre para educação utilizando metareciclagem**. Disponível em: <https://uriedubot.wordpress.com>. Acesso em: 28 Ago. 2014.

ERL, T. **Service-oriented architecture**: concepts, technology, and design. Indianapolis, Indiana: Prentice Hall, 2005.

ERL, T. **SOA**: principles of service design. Indianapolis, Indiana: Prentice Hall, 2007.

FIGUEIREDO, L. C.; JOTA, F. G. Introdução ao Controle de Sistemas Não-Holonômicos. **Revista Controle Automação**, Campinas, São Paulo, v.15, n.3, p.243–268, Set. 2004.

FRISCHHOLZ, R. **Face Detection Datasets**. Disponível em: <https://facedetection.com/datasets/>. Acesso em: 10 Ago. 2015.

FUGITA, H. S. **MAPOS**: método de análise e projeto orientado a serviços. 2009. 175p. Dissertação (Mestrado em Engenharia da Computação e Sistemas Digitais) — Escola Politécnica da Universidade de São Paulo, São Paulo, 2009.

GARRO, R.; ORDINEZ, L.; SCASSO, M. Patrones de Diseño para Sistemas Ciber Físicos. In: CONGRESO MICROELECTRÓNICA APLICADA. 4., 2013, Bahía Blanca. **Anais...** Bahía Blanca: uEA, 2013. n.4, p.17–21.

GUIBOT. **Driver Motoruino**. Disponível em: <http://www.guibot.pt/>. Acesso em: 14 Nov. 2014.

GUIMARÃES, C.; HENRIQUES, R.; PEREIRA, C. Protótipo de um Sistema Embarcado para Auxílio à Macro e Micro Navegação de Deficientes Visuais. In: ENCUENTROS IBEROAMERICANOS: SEMINARIO CYTED DE LA RED TEMÁTICA IBERADA JORNADAS AITADIS DE TECNOLOGÍAS DE APOYO A LA DISCAPACIDAD. 5., 2012, Vitória. **Anais...** Vitória: GM, 2012. n.5, p.92–97.

GUIMARÃES, C.; HENRIQUES, R.; PEREIRA, C. Analysis and Design of an Embedded System to aid the navigation of the Visually Impaired. In: IEEE BIOSIGNALS AND BIOROBOTICS CONFERENCE. 4., 2013, Rio de Janeiro. **Anais...** Rio de Janeiro: IEEE, 2013. n.4, p.1–6.

GUIMARÃES, C.; PEREIRA, C.; HENRIQUES, R. Telemetria e Telecontrole de um Sistema Embarcado Aplicado na Macro e Micro Navegação de Deficientes Visuais. In: INTERNATIONAL CONFERENCE ON REMOTE ENGINEERING AND VIRTUAL INSTRUMENTATION. 11., 2014, Porto. **Anais...** Porto: IEEE, 2014. n.11, p.424–433.

GUIMARÃES, C. S. S. **EduBOT UFRGS - Algoritmo seguidor de parede**. Disponível em: <https://www.youtube.com/watch?v=Z8KsVS1BubI>. Acesso em: 12 Jun. 2015.

GUIMARÃES, C.; TAMAYO, J.; HENRIQUES, R. Robotica para los Procesos de Ensenanza de la Mecatronica: desarrollo del prototipo edubotv2 para la mejora de procesos de ingenieria de control. In: CONGRESO INTERNACIONAL SOCIEDAD DIGITAL: CIUDADANIA DIGITAL Y NUEVAS ALFABETIZACIONES. 3., 2014, Madrid. **Anais...** Madrid: Actas Icono14, 2014. n.16, p.349–376.

KUHNE, F. **Controle Preditivo de Robôs Móveis Não Holonômicos**. 2005. 155p. Dissertação (Mestrado em Engenharia da Elétrica) — Universidade Federal do Rio Grande do Sul, Porto Alegre, 2005.

KUROIWA, D. M. A. **Reflexões sobre a Arquitetura Orientada a Serviço e o Surgimento de uma Nova Disciplina, a Engenharia de Software de Serviço**. 2011. 30p. Trabalho de Conclusão (Graduação) — Faculdade de Tecnologia de São Paulo, Bom Retiro, SP - Brasil, 2011.

LAGES, W. F. **Controle e Estimação de Posição e Orientação de Robôs Móveis**. 1998. 180p. Tese (Doutorado em Engenharia Eletrônica e Computação) — Instituto Tecnológico de Aeronáutica, São José dos Campos, 1998.

LEE, E. A.; SESHIA, S. A. **Introduction to Embedded Systems**: a cyber physical systems approach. Berkeley: University of California, 2011.

MARTINS, M. d. S. **osTWDR - Open source two wheeled differential robot**. Disponível em: <http://msm.no.sapo.pt/osTWDR/>. Acesso em: 01 Nov. 2011.

MAXWELL. **Repositório institucional da Pontifícia Universidade Católica do Rio de Janeiro**: certificação digital 0410823/CA. Disponível em: <http://www.maxwell.vrac.puc-rio.br/8623/8623_3.PDF>. Acesso em: 08 Set. 2014.

MAXWELL. **Repositório institucional da Pontifícia Universidade Católica do Rio de Janeiro**: certificação digital 0210486/CA. Disponível em: <http://www.maxwell.vrac.puc-rio.br/5064/5064_3.PDF>. Acesso em: 05 Jun. 2014.

MIRANDA, L. C.; SAMPAIO, F. S.; BORGES, J. A. RoboFácil: especificação e implementação de um kit de robótica para a realidade educacional brasileira. **Revista Brasileira de Informática na Educação (RBIE)**, Porto Alegre, v.18, n.3, p.47–58, 2010.

MOBILEROBOTS, A. **Pioneer P3-DX**. Disponível em: <http://www.mobilerobots.com/researchRobots/PioneerP3DX.aspx>. Acesso em: 10 Sep. 2014.

MYROBOTLAB. **Open source Java service based framework for robotics**. Disponível em: <http://www.myrobotlab.org>. Acesso em: 16 Oct. 2014.

OASIS, W. **Web Services Business Process Execution Language**. Disponível em: <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>. Acesso em: 06 Out. 2013.

PAPAZOGLOU, M. Service-Oriented Computing: concepts, characteristics and directions. In: INTERNATIONAL CONFERENCE ON WEB INFORMATION SYSTEMS ENGINEERING. 4., 2003, Tilburg. **Anais...** Tilburg: IEEE, 2003. n.4, p.12.

POLOLU. **Pololu Robotics and Electronics**. Disponível em: <https://www.pololu.com/>. Acesso em: 05 Aug. 2014.

PRESSMAN, R. S. **Software Engineering**: a practitioner's approach. New York: McGraw-Hill, 2010.

RINCON, R. L. **Framework de Definição de Trajetória para Robôs Móveis**. 2014. 71p. Trabalho de Conclusão (Graduação) — Universidade de Brasília - UnB, Brasília, DF, 2014.

ROSA, P. R. M. **Bengala de apoio a cegos com detecção de buracos**. 2009. 81p. Dissertação (Mestre em Engenharia Electrónica e Telecomunicações) — Universidade de Aveiro, Aveiro, 2009.

SECCHI, H. **Uma Introdução aos Robôs Móveis**. 2008. 81p. Trabalho de Conclusão (Graduação) — Universidade Nacional de San Juan, San Juan - Argentina, 2008.

SERAFIM, G. CBD vs. SOA Serviços, processos de negócio e componentes que realizam os serviços. **Engenharia de Software**, São Paulo, v.2, p.243–268, Nov. 2009.

SIEGWART, R.; NOURBAKHSH, I. R. **Introduction to Autonomous Mobile Robots**: autonomous mobile robots. Cambridge, Massachusetts: MIT Press, 2004.

SILVA, A. F. da. **RoboEduc**: uma metodologia de aprendizado com robótica educacional. 2009. 115p. Tese (Doutorado em Engenharia da Computação) — Universidade Federal do Rio Grande do Norte, Lagoa Nova, 2009.

SILVA, R. P. e. **Suporte ao desenvolvimento e uso de frameworks e componentes**. 2000. 262p. Tese (Doutorado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul, Porto Alegre, 2000.

SOLIDWORKS. **3D CAD Design Software Solidworks**. Disponível em: <https://www.solidworks.com/>. Acesso em: 13 Jun. 2014.

SOMMERVILLE, I. **Software Engineering**. Massachusetts: Pearson, 2011.

WARD, P. A.; MELLOR, S. J. **Structured Development for Real-Time**: implementation modeling techniques. Upper Saddle River, New Jersey: Prentice Hall, 1986.

WEHRMEISTER, M. A. **Framework Orientado a Objetos para Projeto de Hardware e Software Embarcados para Sistemas Tempo-Real**. 2005. 104p. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul, Porto Alegre, 2005.