UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SARA^{MR}: UMA ARQUITETURA DE REFERÊNCIA BASEADA EM LOOPS DE CONTROLE PARA FACILITAR MANUTENÇÕES EM SOFTWARE ROBÓTICO AUTOADAPTATIVO

MARCOS HENRIQUE DE PAULA

ORIENTADORA: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos - SP
junho/2015

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

SARA^{MR}: UMA ARQUITETURA DE REFERÊNCIA BASEADA EM LOOPS DE CONTROLE PARA FACILITAR MANUTENÇÕES EM SOFTWARE ROBÓTICO AUTOADAPTATIVO

MARCOS HENRIQUE DE PAULA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software
Orientador: Prof. Dr. Valter Vieira de Camargo

São Carlos - SP
junho/2015

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária UFSCar
Processamento Técnico
com os dados fornecidos pelo(a) autor(a)

P324s

Paula, Marcos Henrique de
SARAMR: uma arquitetura de referência baseada em loops de controle para facilitar manutenções em software robótico autoadaptativo / Marcos Henrique de Paula. -- São Carlos : UFSCar, 2015.
153 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2015.

1. Arquitetura de software. 2. Arquitetura de referência. 3. Robô móvel autônomo. I. Título.

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Marcos Henrique de Paula, realizada em 08/06/2015:

Prof. Dr. Valter Vieira de Camargo
UFSCar

Prof. Dr. Glauco de Figueiredo Carneiro
UNIFACS

Profa. Dra. Elisa Yumi Nakagawa
USP

Dedico esse trabalho a meus pais.

AGRADECIMENTOS

Agradeço a Deus que, em Sua força nos concede a possibilidade de viver, continuar existindo e realizando.

Gostaria de agradecer a meu pai Alcebíades e minha mãe Luzia, pelo seu amor e carinho, por me ensinarem seus valores e por serem exemplos para a minha vida. Infelizmente, meu pai não está mais conosco, ele nos deixou durante o período de realização deste trabalho de mestrado.

Um agradecimento especial ao meu orientador Prof. Valter pelos seus ensinamentos, e que sem a sua atuação, este trabalho não teria se concretizado. Com ele, pude perceber o quanto pude aprender ao longo da realização deste trabalho.

Minha esposa Daniela e meu filho Arthur merecem também um agradecimento especial, pois foram parte dessa conquista e juntos comigo passaram pelos momentos difíceis e me apoiaram na realização deste trabalho.

Agradeço à Profa. Marilde pela sua atenção, que foi a primeira pessoa a me receber no Departamento de Computação da UFSCAR, ao Prof. Marcio Merino e ao Prof. Mario Liziér pelo apoio em momentos difíceis durante o mestrado.

Agradeço também a todos os professores e funcionários do departamento de pós graduação da UFSCAR - Universidade federal de São Carlos.

RESUMO

Robôs móveis autônomos fazem parte de uma categoria especial de robôs projetados para realizar tarefas sem a intervenção de seres humanos. Alguns robôs são projetados para realizar tarefas em ambientes completamente inóspitos à vida humana como no subsolo terrestre, nas profundezas de oceanos ou na exploração espacial. Para que um robô seja considerado autônomo, uma premissa fundamental é possuir capacidades de autoadaptação. Nos últimos anos, os avanços da tecnologia possibilitaram o desenvolvimento de sistemas robóticos autoadaptativos, que são capazes de gerenciarem a si próprios, se recuperarem de falhas e também de alterarem seu comportamento e estrutura com o objetivo de otimizar e/ou manter a qualidade do serviço (QoS) oferecido. Uma questão crítica para a concepção e construção de qualquer sistema de software é sua arquitetura, isto é, sua organização estrutural em um conjunto de componentes que interagem. Nesse contexto, a utilização de arquiteturas de referência é uma abordagem conhecida atualmente por combinar as melhores práticas, padrões e estratégias para a construção e padronização de sistemas de software para um determinado domínio. Atualmente, nota-se uma carência de estudos que apresentem arquiteturas de referência para estruturar o software de robôs móveis autoadaptativos de forma a facilitar atividades de manutenção nesses sistemas. Muitos estudos apontam que sistemas autoadaptativos são baseados na teoria do controle e mais especificamente na utilização de loops de controle em sua arquitetura para realizar as adaptações. Diante disso, este trabalho propõe a arquitetura de referência SARA^{MR}, uma arquitetura de referência baseada em loops de controle cujo objetivo é facilitar atividades de manutenção no software desses sistemas. A utilização da arquitetura divide o sistema em dois módulos: aplicação base e módulo de adaptação. O módulo de adaptação envolve os loops de controle e a aplicação base ainda é subdividida em três componentes: ambiente, comportamentos e a parte eletromecânica. SARA^{MR} foi avaliada de forma qualitativa mediante o desenvolvimento duas aplicações: um robô autoadaptativo seguidor de paredes e um outro convencional de patrulhamento. Depois disso, algumas manutenções evolutivas foram idealizadas no sentido de averiguar o esforço de aplicá-las. Constatou-se que a separação de interesses existente na arquitetura permite que novos componentes possam ser adicionados com impacto menor do que em sistemas que não usam essa arquitetura.

Palavras-chave: Arquitetura de software, Arquitetura de Referência, Robô Móvel Autônomo, Loop de Controle, Lego NXT, Lejos NXJ.

ABSTRACT

Autonomous mobile robots are a special category of robots designed for performing tasks without the intervention of human beings. Some robots are designed to perform tasks in completely inhospitable environments such as the earth´s subsurface, the ocean depths or spatial exploration. In order to consider a robot as autonomous, a fundamental premise is to have self-adaptation capabilities. Over the last years, the advances in technology allow the development of self-adaptive systems, which are able to manage themselves to recuperate from faults or even change their behavior and structure in order to improve the quality of the delivered service. A critical point when building any software is its architecture, i.e., its structural organization in a set of interacting components. In this context, reference architecture is a technique that is well known for combining the best practices, patterns and strategies for building and standardizing domain specific software. Nowadays, there is a lack of studies presenting reference architectures for structuring self-adaptive software of mobile robots in order to decrease maintenance efforts. A number of studies claim that self-adaptive systems are based on the control theory and, more specifically, on the use of control loops in their architecture to perform adaptations. Therefore, this master thesis proposes SARA^{MR}, a control loop-based reference architecture whose goal is to make maintenance activities a more productive task. The employment of the architecture divides the whole system in two modules; base application and adaptation module. The adaptation module encompasses the control loops and the base application is further divided into three other components: environment, behaviors and the electromechanical part. SARA^{MR} was qualitatively evaluated by means of the development of two applications: a self-adaptive wall follower mobile robot and another conventional one to performing monitoring in in-door environments. Next, some maintenance activities were created to investigate the effort of applying them. We have observed that the separation of concerns of our architecture allows new components to be added causing less impacts than in systems developed in an ad-hoc way.

Keywords: Software architecture, Reference Architecture, Autonomous Mobile Robots, Feedback Control Loop, Lego NXT, Lejos NXJ.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API - *Application Programming Interface*

AR - *Arquitetura de Referência*

ERP - *Enterprise Resource Planning*

IBM - *International Business Machines*

IDE - *Integrated Development Environment*

JVM - *Java Virtual Machine*

MAPEK - *Monitor, Analyzer, Planner, Executor and Knowledge*

MIAC - *Model Identification Adaptive Control*

MRAC - *Model Reference Adaptive Control*

MVC - *Model View Controller*

RMA - *Robôs Moveis Autônomos*

SOA - *Service-Oriented Architecture*

SOAP - *Simple Object Access Protocol*

XML - *Extensible Markup Language*

VANT - *Veículo Aereo não Tripulado*

PID - *Proporcional Integral Derivativo*

Capítulo 1

INTRODUÇÃO

1.1 Contextualização

Robôs móveis autônomos (RMA) são dispositivos eletromecânicos com capacidade de realizar atividades com pouca ou nenhuma intervenção de seres humanos. Atualmente, os RMAs estão ganhando cada vez mais espaço no cotidiano das pessoas com a realização de diversos tipos de tarefas (WOLF *et al.*, 2009). O crescente interesse por RMAs deve-se à grande diversidade de tarefas que podem ser realizadas por eles e também à popularização do hardware de baixo custo. O desenvolvimento de aplicações robóticas envolve várias áreas de conhecimento: um RMA deve ter capacidade de mobilidade, percepção e interação com o ambiente. Para se deslocar adequadamente no ambiente, um RMA geralmente é composto de vários atuadores e sensores dependendo do tipo de tarefa que deve realizar. Os atuadores dão capacidade de movimento. As informações adquiridas por diversos tipos de sensores são utilizadas para "perceber" o ambiente. Assim, tanto atuadores quanto sensores podem ser adicionados em qualquer parte do robô. Esses são aspectos que envolvem o projeto físico de um RMA.

O software que controla as funcionalidades de um RMA tem sido reconhecido como uma das partes mais desafiadoras para a construção de um sistema robótico (BRÄUNL, 2008, BROTEN *et al.*, 2006). Cabe ao software instalado no robô a tarefa de interpretar dados dos sensores e gerar comandos para os atuadores. Geralmente, as aplicações de software desenvolvidas para RMAs são grandes bloco de código construídos para implementar muitas funcionalidades, não existindo uma separação clara de funcionalidades. Assim o código que implementa os sensores e atuadores estão misturados aos códigos que controlam os

comportamentos do robô, ou mesmo a outros códigos utilizados para a locomoção e localização do robô no ambiente.

A partir da década de 90, arquitetura de software começou a receber muita atenção e passou a ser considerada como uma importante disciplina da área de Engenharia de Software. De acordo com Shaw e Garlan (1994), a arquitetura de software ajuda a reconhecer princípios comuns e relações de alto nível entre sistemas. A utilização de arquiteturas de software é crucial para o sucesso na produção de sistemas, pois a compreensão detalhada da arquitetura do software permite ao engenheiro fazer escolhas baseadas em princípios e alternativas de projeto. Ainda nessa mesma década, surgiram os primeiros estudos, produzidos por Mettala e Graham (1992), indicando o uso de arquiteturas de software para domínios específicos. Esse tipo de arquitetura passou a ser conhecida como arquitetura de referência. Uma arquitetura de referência é um tipo especial de arquitetura de software que captura a essência das arquiteturas de uma coleção de sistemas de um mesmo domínio. Um dos propósitos de uma arquitetura de referência é principalmente prover suporte para o desenvolvimento, padronização e evolução das arquiteturas de sistemas de software (NAKAGAWA, 2012).

Em outro contexto, mais voltado ao desempenho, nas últimas décadas, os avanços da tecnologia possibilitaram o desenvolvimento de sistemas robóticos capazes de gerenciar a si próprios, se recuperar de falhas e também de alterar seu comportamento e estrutura com o objetivo de otimizar o seu desempenho e/ou manter níveis aceitáveis da qualidade do serviço (QoS) oferecido. O termo geralmente utilizado para designar tais sistemas é "sistema autoadaptativo" (SAA). No campo da robótica, também é comum o emprego do termo "robô autônomo". Vale ressaltar que embora o termo tenha surgido mais recentemente na computação (IBM, 2006), SAAs não são novidade. Outros tipos de sistemas já pesquisados anteriormente possuem relação direta com esse termo, como por exemplo, sistemas multiagentes, sistemas tolerantes a falhas e computação ubíqua adaptabilidade (MACÍAS-ESCRIVÁ, 2013; CHENG *et al*., 2009).

O desenvolvimento de SAAs tem sido considerado mais desafiador do que sistemas tradicionais em consequência de sua complexidade (CHENG *et al*., 2009). Muitos estudos demonstram que SAAs são baseados na teoria do controle (*Control Theory*) e intrinsecamente usam loops de controle (*Feedback Control Loops*) em sua arquitetura para realizar as adaptações (CHENG *et al*. 2005, DOBSON *et al*. 2006, BRUN *et al*. 2009). Em consequência disso, é possível identificar na arquitetura lógica de um SAA duas partes distintas: um

subsistema controlado e um subsistema controlador (que são os loops de controle). Entretanto, essas partes usualmente não se encontram divididas de forma clara no código fonte desses sistemas. Um loop de controle é um elemento responsável por: (i) monitorar um determinado processo; (ii) capturar a saída desse processo; (iii) comparar com um valor de referência; e (iv) realizar adaptações na entrada de forma que a saída se torne mais próxima da referência (WEYNS *et al*., 2013a; KOKAR *et al*., 1999; TRUXAL, 1961). Geralmente esses loops são compostos pelos seguintes elementos básicos: monitor, analisador, planejador e executor, sendo o MAPE-K (*Monitor*, *Analyzer*, *Planner*, *Executor – Knowledge-base*), proposto pela IBM (2006), um dos modelos de loop de controle mais utilizados (HURTADO *et al*., 2011).

Autores também apontam que uma das formas de reduzir a complexidade e elevar os níveis de manutenibilidade de SAAs é modularizando/externalizando os loops de controle, já que diversas atividades de manutenção e evolução são voltadas para os componentes desses loops (WEYNS *et al*., 2013b; CHENG *et al*., 2009; GARLAN *et al*., 2004). Isso significa que esses loops devem ser implementados de forma mais modular e evidente nesses sistemas ao invés de seu código fonte ficar espalhado e entrelaçado com os outros módulos da aplicação. Embora alguns autores tenham apresentado propostas de arquitetura para sistemas robóticos, ou realizado uma comparação entre essas arquiteturas, a maioria não considerou a existência de SAAs e assim, não incluíram partes específicas para separar as preocupações de funcionalidades do sistema das preocupações da autoadaptação com os loops de controle (BARCHANSKI 2005, TOWLE e NICOLESCU 2012, KUMBASAR e HAGRAS 2013, SHEIKH *et al*. 2014).

1.2 Objetivos e Motivação

Este projeto de mestrado tem como objetivo mais geral fazer com que atividades de manutenção no software de sistemas robóticos autoadaptativos sejam feitas de forma mais sistemática e produtiva. Embora vários trabalhos reconhecem que o software de sistemas robóticos autoadaptativos são complexos e que modificações podem demandar grande esforço (BERGER *et al.* 1997; BRUGALI, 2007; BAKER *et al.* 2011), nenhum deles concentra-se em fornecer uma solução para facilitar essa tarefa.

Para atingir esse objetivo propõem-se uma arquitetura de referência chamada SARA^{MR} que instrui o engenheiro de software a estruturar o software robótico de forma que essas atividades sejam mais sistemáticas e diminuam o número de locais do software que devem ser modificados. Isto é, vislumbra-se que o emprego da arquitetura faz com que atividades de inclusão, alteração e remoção de elementos relacionados a sua estrutura sejam realizadas de forma mais concentrada e controlada.

No contexto deste trabalho, uma atividade de manutenção concentrada é aquela cuja edição de código-fonte ocorre em um ou poucos locais do sistema. Uma atividade controlada e aquela que as consequências da edição são poucas e previsíveis.

SARA^{MR} é o acrônimo para as palavras *Self-Adaptation*, *Reference Architecture* e *Mobile Robots*. A arquitetura de referência possui os seguintes módulos: i) módulo para a implementação da parte física que compõe o robô; ii) módulo para implementação do ambiente e navegação; iii) módulo para a implementação de comportamentos, e iv) módulo de autoadaptação baseada em loops de controle.

A principal motivação para o desenvolvimento deste projeto é que, assim como sistemas tradicionais, sistemas robóticos autoadaptativos também necessitam de manutenções, muitas vezes, no sentido de evoluir o software que controla o robô. Entretanto, apesar de que vários trabalhos reconhecerem que manutenções nesse contexto são, muitas vezes, mais desafiadoras do que em sistemas tradicionais (BRUGALI, *and* SALVANESCHI, 2006; BRUGALI, 2007; EDWARDS *et al.*, 2009; GEORGAS *and* TAYLOR, 2008), poucos deles concentram-se em facilitar manutenções nesse contexto. Há alguns que apresentam arquiteturas de software (OREZY *et al*., 1999; ALBUS, 2002; BROOKS, 2005, CLARK, 2005), outros propõem frameworks (GARLAN *et al.*, 2004; OREIZY *et al*., 2008; FIERRO *et al*., 2002), outros apresentam ainda formas alternativas de estruturar sistemas robóticos (MOONZOO, 2005; BRUGALI, 2007; LEE, 2010), mas nenhum deles concentra-se em facilitar manutenções em software de robôs autoadaptativos.

Evoluções no contexto de sistemas robóticos autoadaptativos podem ser necessárias por vários motivos, entre eles: i) modificações no ambiente em que o robô atua; ii) necessidades de otimização para melhorar o desempenho do robô; iii) necessidades de inclusão/alteração dos componentes físicos do robô, e iv) necessidade de inclusão/alteração dos comportamentos do robô. Dessa forma, a adequada separação dos principais módulos de

software do robô influencia na produtividade das tarefas de evolução do sistema, isso é, a arquitetura de software escolhida impacta na qualidade dos processos de evolução.

Um ponto importante da arquitetura proposta é o reconhecimento de que loops de controle são componentes essenciais de sistemas autoadaptativos (CHENG *et al*. 2005, DOBSON *et al*. 2006, BRUN *et al*. 2009) e que muitas manutenções/evoluções são voltadas aos componentes desses loops. Assim, a arquitetura proposta eleva os loops de controle a componentes de primeira ordem, deixando-os evidentes na forma de componentes de software para possam ser mais facilmente modificados (WEYNS *et al*., 2013b; CHENG *et al*., 2009; GARLAN *et al*., 2004).

1.3 Organização da Dissertação

No Capítulo 2, os conceitos sobre robôs móveis, de teoria do controle, *feedback control loops* e sistemas autoadaptativos que são conceitos fundamentais para o entendimento da dissertação são apresentados. No Capítulo 3 são apresentados conceitos de arquitetura de software e arquitetura de referência. No Capítulo 4 são apresentados trabalhos relacionados com a pesquisa realizada. O objetivo destes três capítulos é mostrar um panorama atual das áreas de pesquisa relacionados com esta dissertação.

No Capítulo 5, o trabalho segue com a identificação e especificação dos requisitos de software para a arquitetura de referência utilizando como metodologia a exploração de cenários de evolução que ocorrem no domínio de robôs móveis autônomos e também sistemas autoadaptativos. Uma vez especificados os requisitos, ainda nesse capítulo, são mostradas soluções de projeto que serviram de base e inspiração para estabelecer a arquitetura de referência e justificar o trabalho proposto.

No Capítulo 6, é apresentada a arquitetura de referência SARA^{RM} juntamente com o seu detalhamento e explicações de como especializá-la para produzir sistemas de software no domínio de RMAs.

No Capítulo 7, são apresentados dois exemplos de aplicação, com o objetivo de explicar a utilização e, também avaliar a arquitetura de referência demonstrando como a arquitetura atende a todos os requisitos identificados nos cenários de evolução.

No Capítulo 8, é apresentada uma avaliação da arquitetura de referência com o objetivo de demonstrar que os sistemas para RMA desenvolvidos com o seu apoio da realmente possuem facilidade nas atividades que envolvem manutenções evolutivas. A abordagem escolhida para a avaliação baseia-se em exemplos de código referentes aos cenários de evolução identificados no domínio de RMAs e considerados para o desenvolvimento da arquitetura de referência.

As contribuições da dissertação, as dificuldades e limitações encontradas, e algumas propostas de continuidade das pesquisas aqui conduzidas são apresentadas no Capítulo 9.

Capítulo 2

SISTEMAS ROBÓTICOS E SISTEMAS AUTOADAPTATIVOS

Este capítulo tem o objetivo de esclarecer os conceitos que abrangem o conhecimento sobre robôs móveis autônomos e sistemas autoadaptativos. Todos esses conceitos são necessários para entender a solução proposta e estão diretamente relacionados com o tema principal deste trabalho.

2.1 Robôs Móveis e Sistemas Embarcados

Um Robô Móvel Autônomo (RMA) é um sistema mecatrônico que incorpora tecnologias de várias disciplinas nos domínios da engenharia mecânica, eletrônica, engenharia de software, engenharia computacional, inteligência artificial, etc. (DAYANG *et al*., 2007). O estudo da robótica móvel é um tema bastante relevante e atual e apresentou um grande salto em seu desenvolvimento nas últimas duas décadas. Atualmente, existem várias aplicações práticas para os RMAs em nossa sociedade. Por exemplo, em aplicações industriais (veículos de carga autônomos, transporte automatizados), domésticas (aspiradores de pó e cortadores de grama robóticos), urbanas (transporte público, cadeiras de rodas robotizadas), segurança, defesa civil e militar (controle e patrulhamento de ambientes, resgate e exploração em ambientes hostis). O aumento significativo na produção dessas aplicações robóticas e as preocupações em relação ao seu desenvolvimento vêm demonstrando o crescimento dos interesses econômicos envolvidos e o quão promissor é o futuro dessa área (Wolf *et al*., 2009).

O projeto de um sistema robótico envolve a especificação e seleção de diferentes componentes. Os principais dispositivos de hardware de um robô são seus sensores e atuadores. Alguns exemplos famosos de robôs móveis, resultantes da pesquisa e desenvolvimento que vem ocorrendo nessa área são mostrados na Figura 2.1.

Figura 2.1: Exemplos de Robôs Móveis Autônomos, Asimo (Honda 2009), Automower (Huskvarna 2009), Scooba (iRobot 2009), Aibo (Sony 2009), Pioneer 2 e Mars Rover Curiosity (NASA 2008).

ASIMO e AIBO são robôs de entretenimento, movimentam-se por meio de um conjunto de pernas. AutoMower e Scooba são robôs para tarefas domésticas. Pioneer 2 é um robô de uso acadêmico e o Curiosity é um robô de exploração espacial. Como pode ser observado nos exemplos, esses tipos de robôs são adequados à locomoção em terra, alguns deles mais indicados para ambientes fechados e outros podem ser utilizados em ambiente aberto. A forma de locomoção também pode variar, alguns são semelhantes a veículos e usam rodas para se locomover, outros se assemelham à forma humana ou de animais usando pernas para a sua locomoção. Dependendo da função e das tarefas para as quais são projetados, esses sistema robóticos possuem diferentes configurações de dispositivos de hardware embarcados. Basicamente, um RMA é composto dos seguintes dispositivos:

- **Micro Controlador**: É a parte principal de um robô, dotada de um microprocessador e memória para execução de seus programas, além de portas de entrada e saída de dados para interligação de outros dispositivos;

- **Sensores**: São dispositivos que permitem ao robô ter percepção do ambiente em que se encontra. Existem vários tipos de sensores com níveis de precisão e qualidade variados, são eles, sensores ultrassônicos, infravermelho, de temperatura, de pressão, de rotação e ainda outros que produzem diversos tipos de dados que são convertidos em informação de entrada para o controlador;

- **Atuadores**: São dispositivos que dão mobilidade ao robô e seus manipuladores, podem ser motores mecânicos, elétricos, hidráulicos ou pneumáticos com níveis de precisão e qualidade variados;

- **Manipuladores**: são dispositivos que permitem ao robô interagir e transformar o ambiente em que se encontra. Os manipuladores podem ser membros como braços e garras, com eles uma variedade de movimentos pode ser realizada. Normalmente um robô possui um ou mais atuadores em sua estrutura;

- **Comunicação**: dispositivos que permitem ao robô se comunicar com um operador humano, ou mesmo com outros dispositivos robóticos.

Os RMAs possuem, como características fundamentais, as capacidades de locomoção e de operação por modo semi ou completamente autônomo. Maiores níveis de autonomia são alcançados à medida que o robô receba alguns aspectos como a capacidade de percepção (sensores que conseguem “ler” o ambiente), capacidade de agir (atuadores e motores capazes de produzir ações, tais como o deslocamento do robô no ambiente), robustez e inteligência (capacidade de lidar com as mais diversas situações, de modo a resolver e executar tarefas por mais complexas que sejam). Para realizar tarefas mais complexas como autolocalização, mapeamento e navegação autônoma, um RMA vai necessitar de técnicas de planejamento e controle conhecidas pelo termo "Controle Robótico Inteligente" (WOLF *et al*., 2009).

O controle inteligente citado é realizado pelo software do RMA. Cabe ao software instalado no robô a tarefa de interpretar dados dos sensores e gerar comandos para os atuadores. O software de controle de um RMA tem sido reconhecido como uma das partes mais desafiadoras para um sistema robótico (BRÄUNL, 2008, BROTEN *et al*., 2006). Um dos grandes desafios da robótica é justamente como interpretar as informações vindas dos sensores, de modo a gerar comandos e controlar os diferentes dispositivos de atuação do robô, garantindo que a tarefa seja executada corretamente e sem colocar em risco o robô ou aqueles que o cercam (WOLF *et al*., 2009).

Um sistema embarcado é um sistema computacional com uma função dedicada instalado dentro de um sistema mecânico ou elétrico maior. Ele é incorporado como parte de um dispositivo completo, muitas vezes, incluindo hardware e peças mecânicas (HEALT, 2003; BARR *et al*., 2006). O software que controla um RMA é um sistema embarcado, um programa com um conjunto de instruções que controlam ações do robô.

A grande complexidade existente no desenvolvimento de software para robótica se deve ao fato de que os sistemas de software dependem muito do tipo de hardware utilizado no robô. Escrever software para robôs é difícil, principalmente porque a escala e escopo da robótica continuam a crescer. Diferentes tipos de robôs podem ter hardware extremamente diferentes, tornando a reutilização de código uma tarefa pouco trivial (QUIGLEY *et al*., 2009). É importante destacar que o projeto do sistema de controle de um robô deve levar em conta alguns fatores que aumentam a complexidades do projeto como por exemplo o tipo e precisão dos sensores e atuadores embarcados. Os sensores individualmente fornecem apenas uma "percepção", parcial, incompleta e sujeita a erros, sendo papel do sistema de controle adquirir, unificar e tratar essas informações de forma robusta e inteligente. Além disso, os comandos de atuação podem não ser executados de forma precisa pelos atuadores (WOLF *et al*., 2009).

De acordo com Bräunl *et al*. (2003), localização e navegação são as tarefas mais importantes para os RMAs. Wolf *et al*. (2009) destaca que o projeto do software de controle de um robô deve usualmente, através da adoção de uma arquitetura específica de controle, ser capaz de realizar algumas, ou todas das seguintes tarefas:

- **Fusão de sensores**: adquirir e integrar os diversos dados recebidos a partir dos sensores que integram o sistema robótico;
- **Desviar de obstáculos**: detectar obstáculos e poder evitar a colisão, preservando a integridade do robô e dos elementos externos;
- **Auto localização**: determinar a localização do robô no ambiente (posição e orientação) para poder planejar e executar o deslocamento seguindo uma determinada trajetória;
- **Mapeamento do ambiente**: exploração e construção de um mapa do ambiente, onde o robô é capaz de identificar onde existem paredes e obstáculos.
- **Planejamento de trajetórias**: utilizando um mapa é possível planejar ações para definir previamente uma trajetória a ser executada pelo robô, especificando as ações elementares a serem realizadas de modo a se deslocar de uma posição origem até uma

posição destino;

- **Planejamento de ações**: estabelecer um plano de ações, que pode ser composto da execução de ações mais elementares, como por exemplo, explorar um mapa do ambiente, seguir uma parede ou corredor até o seu final, desviar de obstáculos;

- **Navegação robótica**: capacidade de executar o deslocamento de uma posição origem até uma posição destino, realizando os devidos ajustes necessários durante o deslocamento;

- **Interação e Comunicação**: capacidade de emitir mensagens, interagir e se comunicar com outros agentes.

A capacidade de realizar essas tarefas é o objetivo principal do software que controla um RMA.

Kramer e Scheutz (2006) Afirmam que o aumento da capacidade de aplicações robóticas requer amplo suporte com a expansão do desenvolvimento ferramentas e técnicas que serão utilizados não só para o desenvolvimento e depuração do software robótico, mas também para sua execução e manutenção como parte da implementação do aplicativo. A escolha de uma arquitetura para um RMA será baseada no apoio ao desenvolvimento e nos recursos oferecidos para operações a longo prazo das aplicações robóticas.

2.2 Lego MindStorms NXT

O projeto de mestrado apresentado neste documento foi realizado com a utilização do kit Lego MindStorms NXT (LEGO, 2014) como ferramenta de apoio ao desenvolvimento de aplicações para robôs móveis em ambientes fechados. A primeira vista, o kit Lego MindStorms pode parecer simples, entretanto, ele torna-se interessante e adequado à condução de pesquisas quando utilizado em conjunto com a plataforma LeJOS NXJ.

O kit Lego MindStorms é o resultado de uma parceria entre o Media Lab do MIT (*Massachusetts Institute of Technology*) e o LEGO Group, é composto por um conjunto de 612 peças para montar (tijolos vazados, placas, rodas, eixos, engrenagens, polias e correntes etc.), acrescido de sensores atuadores e controlado por um processador programável, o módulo NXT 2.0

Um benefício do kit é que as peças de montagem propiciam uma estruturação mecânica reconfigurável para projetos de robótica. Na necessidade de realizar qualquer readaptação de projeto em pouco tempo, com as diversas combinações possíveis, o robô poderá assumir diferentes formas melhorando o aproveitamento. Na Figura 2.2 são mostrados os principais componentes eletromecânicos do kit.

Figura 2.2: Principais componentes do Kit Lego MindStorms NXT (LEGO, 2014).

1. **Micro processador:** ARM7 de 32 bits, com sete portas de comunicação, quatro para *input*, três para *output*, além de comunicação via Bluetooth ou USB.

2. **Atuadores** (motores): São três unidades e possuem *encoder* com precisão de um pulso por grau de rotação, portanto, possuem boa precisão para aplicações móveis, O *encoder* é importante para a navegação, pois gera informações de odometria. Possui um bom torque para acionamento das rodas que dependendo da velocidade e sentido de rotação, permitem ao robô realizar tanto curvas acentuadas, quanto rotacionar em torno de seu próprio eixo.

3. **Sensor ultrassônico**: Utilizado para identificar obstáculos, medir distâncias e detectar movimentos. A leitura pode ser feita em polegadas ou centímetros, com o sensor estando no máximo a 255 centímetros de distância do objeto a precisão obtida é de +/- 3 cm.

4. **Sensor de toque**: Possui um botão sensível ao toque, é utilizado para estabelecer limites aos movimentos do robô.

5. **Sensor de luminosidade e cor RGB**: Pode distinguir a intensidade luminosa numa escala entre branco e preto. Pode identificar até seis cores diferentes.

6. **Sensor infravermelho:** É capaz de detectar fontes de luz infravermelha e determinar sua direção e força relativa, possui cinco detectores infravermelhos dispostos em intervalos de 60°.

As interligações entre esses componentes podem ser vistas na Figura 2.3, são quatro portas para sensores e três portas para atuadores.

Figura 2.3: As conexões entre os componentes eletromecânicos (LEGO, 2014).

A programação nativa para o robô Lego é feita por meio de blocos programáveis, de forma totalmente visual, com a ferramenta NXT-G que vem junto ao kit. São vários tipos de blocos e cada um é usado para uma ação diferente. Para desenvolver programas básicos como controlar os atuadores e fazer leitura dos sensores, a NXT-G é muito simples de usar, mas para programas mais avançados, o formato visual da ferramenta deixa os recursos de programação um pouco limitados. Para melhorar essa situação, uma boa alternativa, muito utilizada no meio acadêmico, é a utilização da plataforma *open source* LeJOS NXJ.

2.3 Plataforma LeJOS NXJ

A plataforma LeJOS NXJ foi idealizada com base na plataforma Java e é composta de uma máquina virtual e uma API (*Application Programming Interface*), para desenvolvimento de aplicações. A LeJOS NXJ JVM (*Java Virtual Machine*) é uma máquina virtual usada como plataforma para executar programas escritos na linguagem Java e a API LeJOS. A API é uma biblioteca de classes Java que fornece muitos recursos para o desenvolvimento de software para robôs móveis.

Para utilizar os recursos da API, é necessário realizar uma substituição do firmware original Lego MindStorms pelo novo firmware LeJOS NXJ. Uma vez instalada a nova JVM, o próximo passo é escolher uma ferramenta IDE (ambientes integrados de desenvolvimento) como o Eclipse ou o NetBeans, importar a API (classes.jar) e começar a desenvolver os programas. O programa executável desenvolvido é transferido para o microcontrolador NXT via ligação com cabo USB ou via conexão Bluetooth por meio de comandos das próprias ferramentas IDE. A API LeJOS foi projetada para executar sobre a LeJOS JVM e possui os seguintes recursos:

- Linguagem Java orientada a objetos;
- Permite a utilização de IDE ambientes integrados de desenvolvimento, Eclipse e Netbeans;
- Suporte a várias plataformas, Windows, Linux e Mac OSX;
- Suporte completo a protocolos Bluetooth, USB, I2C e RS485;
- Suporte a navegação avançada;
- Suporte a localização, incluindo algoritmos de Monte Carlo (MCL);
- Suporte a algoritmos probabilísticos de robótica, como filtros de Kalman;
- Suporte a controle baseado em comportamentos (arquitetura de subsunção) para facilitar a programação de comportamentos robóticos complexos;
- Suporte a execução remota via PC através de Bluetooth ou USB;
- Fornece cálculo de ponto flutuante, trigonometria e outras funções matemáticas;
- *Mult-threading* (tarefas simultâneas);
- Recursão;
- Sincronização;
- Tratamento a exceções;
- Classes básicas da linguagem Java: java.lang, java.util e java.io.

É importante destacar que a API LeJOS possui muitas classes com diversos recursos para o desenvolvimento de software orientado a objetos no domínio da robótica móvel, inclusive pacotes de classes como o lejos.robotics.subsumption criado para atender a requisitos de arquitetura baseada em comportamentos, um fator que contribuiu muito para os estudos realizados nesse trabalho. Um outro fator que contribuiu para a realização desse trabalho é a existência de uma grande quantidade de códigos fonte de aplicações robóticas móveis utilizando a API LeJOS (GOOGLE CODE, 2013). Os códigos fonte encontrados foram usados como fonte de aprendizado e posteriormente também colaboraram para uma análise de domínio para esse tipo de sistema. A API LeJOS, com todos os seus recursor, se tornou uma grande fonte de informação e os estudos realizados sobre ela produziram uma grande fonte de conhecimento no domínio de RMAs.

O objetivo dessa seção foi apresentar uma introdução aos conhecimentos que envolvem a robótica móvel. Neste ponto, destaca-se a referência ao conceito de arquitetura de controle que remete ao tema principal do presente trabalho. Arquiteturas de software e arquiteturas de referência, assim como alguns exemplos de arquiteturas de software para robótica são assuntos que serão abordados com maior profundidade no próximo capítulo.

2.4 Teoria do Controle e Sistemas Autoadaptativos

Como foi visto nos anteriormente, um RMA funciona com pouca ou nenhuma intervenção humana. O sistema de controle de um RMA integra componentes mecânicos, eletrônicos e software. O software produzido para o controle de RMAs, muitas vezes, terá que abordar problemas como a imprecisão das informações obtidas dos sensores ou também com a imprecisão dos atuadores na execução de comandos. Alguns trabalhos de pesquisa como Ceccarelli *et al*. (2011) e Delafosse *et al*. (2005) apontam esse tipo de problema que é inerente ao domínio de RMAs. Além disso, as características dinâmicas do ambiente causam situações que podem gerar a necessidade de adaptações no software de controle do robô. Esse cenário aponta para a necessidade de sistemas de software com mecanismos para monitorar eventos, com condições e estratégias para adaptar-se a essas situações.

Dois termos importantes que emergem nesse contexto são "sistemas adaptativos" e "sistemas autoadaptativos". De acordo com Oreizy *et al*., (1999), um sistema é considerado adaptativo quando seu comportamento é alterado com base em mudanças de contexto ou necessidades do usuário. Mudanças de contexto caracterizam-se como situações que o usuário passa em seu dia a dia. Por exemplo, reuniões, viagens, passeios, etc. Já um sistema é autoadaptativo quando ele reage a mudanças em seu ambiente operacional, por exemplo: enfraquecimento de sinais de GPS, ausência ou sobrecarga de rede, diminuição da carga da bateria, ausência de espaço de armazenamento, sobrecarga de processamento do processador, etc. Em geral, o objetivo da autoadaptação é manter a mesma qualidade do serviço oferecido ou até mesmo melhorá-la.

A autoadaptabilidade tem sido estudada nas diferentes áreas de pesquisa da engenharia de software, incluindo a de robôs móveis autônomos, e tem sido proposta como uma abordagem eficaz para enfrentar a crescente complexidade do gerenciamento de software de sistemas modernos. Autoadaptação faz com que um sistema de software tenha a capacidade de lidar de forma autônoma com dinâmica interna, bem como a dinâmica do ambiente (CHENG *et al*., 2009; BRUN *et al*. 2009).

Sistemas autoadaptativos é um paradigma de controle de alto nível e teve sua origem na teoria do controle por meio de um conceito conhecido como *control loop* ou *feedback control loop*. Teoria de controle é um ramo interdisciplinar de engenharia e matemática que lida com o comportamento de sistemas dinâmicos. Muitos autores (KEPHART and CHESS 2003; CHENG *et al*., 2009; BRUN *et al*. 2009; ABEYWICKRAMA, 2012, VROMANT, 2011) concordam que os sistemas autoadaptativos incorporam intrinsecamente *feedback loops*. Ao longo dos últimos 40 anos, na teoria de controle, desenvolveu-se uma arquitetura bastante simples (Figura 2.4). É uma arquitetura para um *fedback loop* para gerenciar um sistema ajudando-o a alcançar um objetivo desejado. O componente que manipula o sistema de gerenciado é o controlador (DIAO *et al*. 2005). Os elementos essenciais dessa arquitetura de referência para um *feedback loop* são mostrados na Figura 2.6.

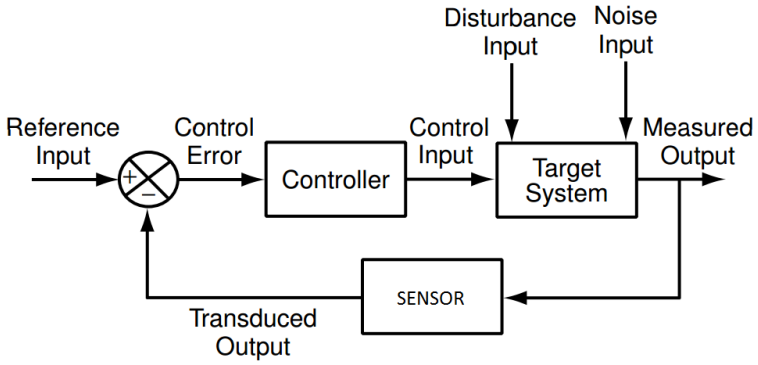

Figura 2.4: Diagrama de blocos de um feedback loop, adaptada de DIAO et al. (2005).

- ***Target System***: é o sistema gerenciado;
- ***Measured output***: é uma característica representada por um valor mensurável coletado do sistema gerenciado;
- ***Transduced output***: é o valor filtrado coletado pelo sensor;
- ***Reference input***: é o valor desejado que será comparado com *measured output;*
- ***Control Error***: é a diferença entre *measured output* e *reference input;*
- ***Controller***: o controlador calcula os valores da entrada com base em valores atuais e passados e determina uma configuração necessária (*control output*) para alterar o comportamento de sistema gerenciado com base no valor de *control error*;
- ***Control input***: é o parâmetro que afeta o comportamento do sistema gerenciado e pode ser ajustado dinamicamente;
- ***Disturbance input****:* é qualquer mudança que afete a forma em que a *control input* influencie *measured output*;
- ***Noise input***: é qualquer efeito afeta o valor *measured output*, é também chamado de ruído do sensor ou ruído de medição.

O fluxo de dados em um *feedback loop* forma um ciclo genérico, dados de saída são monitorados e analisados contra um modelo de referência, em seguida, um mecanismo de ajuste atua de volta no sistema aplicando políticas de adaptação ajudando o sistema a atingir seu objetivo. No caso de RMAs, os sensores fornecem um modo para obter e monitorar os dados do ambiente, e os atuadores fornecem um meio para alterar o comportamento do sistema gerenciado.

Outros dois trabalhos apresentam modelos de referência para sistemas autoadaptativos com feedback loops, *Model Identification Adaptive Control* (MIAC) (SODERSTROM e STOICA, 1988) e *Model Reference Adaptive Control* (MRAC) (ASTROM e

WITTENMARK, 1995). Como é mostrado na Figura 2.11, o fluxo de dados é semelhante à arquitetura de referência anterior. A estratégia modelo MIAC é observar o sistema gerenciado, usando input *U* e output *Yp*, e construir um modelo de referência dinâmico. Esse processo, coleta dados do sistema gerenciado por um determinado tempo, e gera um elemento de identificação do sistema. Então, o elemento de identificação fornece as características identificadas do sistema ao mecanismo de ajuste que, em seguida, ajusta em conformidade, definindo os parâmetros do controlador (BRUN *et al*. 2009).

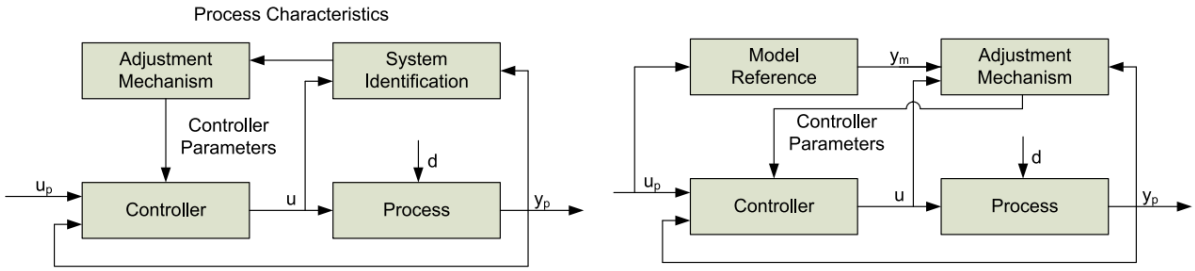

Figura 2.5: Modelos de referência MIAC e MRAC (BRUN *et al*. 2009).

A estratégia do modelo MRAC baseia-se em um modelo de referência predefinido que inclui entradas de referência. A solução do modelo MRAC que foi originalmente proposta para o problema de sistemas de controle de voo, é adequado para situações em que o processo controlado tem que seguir um comportamento previamente elaborado e descrito pelo modelo de referência. O algoritmo adaptativo compara a saída *Yp* do processo que resulta no valor de controle *U*, o mecanismo de ajuste compara os valores de acordo com o objetivo no modelo de referência, depois ajusta o modelo definindo parâmetros do controlador para melhorar o ajuste no futuro. Esse modelo proporciona um sistema de adaptação mais instantâneo pois já possui um modelo de referência. Já a estratégia de adaptação do modelo MIAC tende a ser mais lento, pois tem de levar em conta que distúrbios d podem afetar o comportamento do processo e, portanto, geralmente tem que observar o processo para vários ciclos de controle antes de iniciar ajustes nos parâmetros do controlador (BRUN *et al*., 2009). Na Figura 2.12 é mostrado o ciclo genérico de um *feedback loop* que geralmente envolve quatro atividades principais: coletar, analisar, decidir e agir. Sensores monitoram o sistema em execução e coletam dados sobre o seu estado atual.

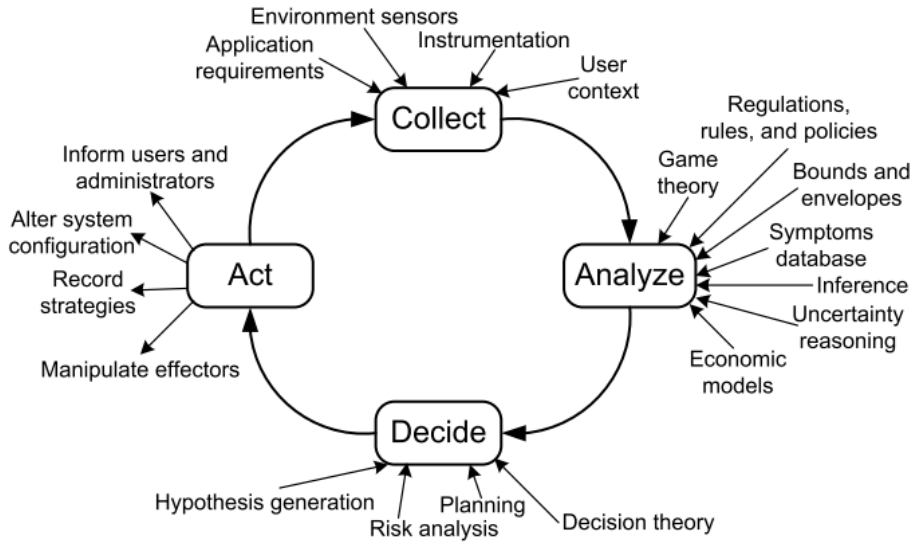

Figura 2.6: Ciclo genérico de um feedbackloop (DOBSON et al. 2006).

- ***Collet***: Coleta os dados do sistema em execução e o contexto de seu estado atual. Os dados devem ser tratados, filtrados e finalmente guardados como referência futura para retratar um modelo preciso de estados atuais e passados.
- ***Analyze***: Analisa os dados para inferir tendências e identificar e diagnosticar os sintomas.
- ***Decide***: Tenta prever o futuro, monta um plano para decidir sobre a forma de agir sobre o sistema em execução e seu contexto..
- ***Act***: Recebe um plano de ação e o executa por meio dos atuadores

Os sensores monitoram o sistema em busca de situações predeterminadas que recebem o nome de "eventos". Um valor ou um conjunto de valores específicos podem caracterizar um evento que seja de interesse para a coleta de dados. Assim os dados são filtrados e selecionados para coleta de acordo com critérios específicos. A atividade de analisar envolve mecanismos que correlacionam os eventos coletados com situações complexas de acordo com modelos de adaptação predefinidos. A atividade de decidir abrange mecanismos que constroem as ações necessárias para atingir metas e objetivos de adaptação. A atividade de agir consiste em executar grupos de funções dos mecanismos que controlam a execução de um plano de adaptação para atualizações dinâmicas em tempo de execução (BRUN et al., 2009).

Nesta seção foram apresentados os conceitos que abrangem o conhecimento sobre robôs móveis autônomos e teoria do controle com sistemas autoadaptativos e loops de

controle. No próximo capítulo, serão apresentados os conceitos que envolvem arquiteturas de software e arquiteturas de referência.

Capítulo 3

ARQUITETURA DE SOFTWARE E ARQUITETURA DE REFERÊNCIA

O objetivo deste capítulo é introduzir conceitos de engenharia de software que envolvem Arquitetura de Software, Arquitetura de Referência, Requisitos Arquiteturais e métodos de avaliação de arquiteturas de software. Todos esses conceitos são importantes para o entendimento deste projeto de mestrado. Em adicional será apresentado um exemplo de arquitetura de referência para sistemas autoadaptativos.

3.1 Arquitetura de Software

Um aspecto importante no desenvolvimento de software é a escolha do modelo arquitetural que vai definir a forma como o software será estruturado. Na fase de desenvolvimento de sistemas, os projetistas utilizam uma arquitetura de software para determinar quais serão os componentes do sistema, como serão estruturados e como serão os relacionamentos entre esses componentes. A arquitetura de software tem importância fundamental na produção de um software de qualidade. Empregar uma arquitetura de *software* adequada vai incorporar características de reusabilidade de software que facilitam o processo de manutenção e evolução do sistema (GARLAN; SHAW, 1994; GARLAN, 2000).

Os pesquisadores Mary Shawn e David Garlan, da Universidade Carnegie Mellon, foram os primeiros a tratar a arquitetura de software como uma disciplina da Engenharia de Software. Na definição de Shawn e Garlan (1996), uma arquitetura de software define o que é o sistema em termos de componentes computacionais e os relacionamentos entre esses componentes, os padrões que guiam a sua composição e restrições. Para Bass *et al*. (2003), a

arquitetura de software de um sistema computacional é definida como a estrutura ou estruturas do sistema, que incluem elementos de software, propriedades visíveis externamente desses elementos e os relacionamentos entre eles. O padrão internacional ANSI/IEEE 1471-2000 define arquitetura de software como a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu projeto e evolução.

Embora existam inúmeras definições de arquitetura de software na literatura, pode-se dizer que elaborar a arquitetura de um sistema significa organizá-lo de maneira abstrata em componentes e conectores estabelecendo uma topologia que mostre como esses elementos se relacionam uns com os outros. Os componentes são blocos de alto nível que descrevem a arquitetura, armazenam algum dado do sistema e realizam algum tipo de computação. Cada componente possui uma funcionalidade bem definida, e assim é capaz de oferecer modularidade e separação de conceitos. Geralmente os componentes publicam uma ou mais interfaces que constituem sua identificação. Essas interfaces expõem as propriedades visíveis externamente que possibilitam fazer a conexão entre os componentes na arquitetura (GARLAN, 2000).

Junto às várias definições apresentadas, surgiu também na área de arquitetura de software alguns conceitos como o de padrão arquitetura, estilo arquitetural, modelo de referência, arquitetura de referência e arquitetura concreta. Esses conceitos são importantes, pois têm a finalidade de estabelecer um vocabulário comum entre os engenheiros de software, além disso, estabelecem regras e restrições envolvidas na combinação entre os componentes da arquitetura.

Um padrão arquitetural define uma família de elementos em termos de padrão organizacional e estrutural, incluído um conjunto de restrições sobre eles. Exemplos de estilos arquiteturais são os *pipers and filters*, *cliente-server*, *peer to peer*, *blackboard*, *event-base*, etc (GARLAN e SHAW, 1994; BASS *et al*., 2003).

Modelo de referência é a divisão de funcionalidades em elementos juntamente com o fluxo de dados entre eles. É a decomposição de um problema conhecido em partes que cooperam entre si para resolver o problema (BASS *et al*., 2003).

Arquiteturas concretas são projetadas com base em uma arquitetura abstrata por meio da aplicação de várias transformações, ou seja, uma arquitetura concreta é obtida como

resultante de uma arquitetura abstrata e, dessa forma, irá preservar todas as propriedades do projeto abstrato referencial. A arquitetura de software produzida incorpora um conjunto de especificações que guiam no desenvolvimento de cada parte do sistema a ser desenvolvido. Nesse contexto, Garlan e Shaw (1994) enfatizam a existência de um grande interesse pelo desenvolvimento de arquitetura de software de domínio específico, também chamada de arquiteturas de referência.

3.2 Arquitetura de Referência

Uma arquitetura de referência é um tipo especial de arquitetura de software que captura a essência das arquiteturas de uma coleção de sistemas que compartilham o mesmo domínio. O propósito de uma arquitetura de referência é principalmente prover suporte para o desenvolvimento, padronização e evolução das arquiteturas de sistemas de software (NAKAGAWA, 2012). Outros autores definem arquitetura de referência como um modelo de referência mapeado em elementos de software e os fluxos de dados entre eles. Esses elementos de software cooperativamente implementam as funcionalidade definidas no modelo de referência (BASS *et al*., 2003; ANGELOV *et al*., 2009).

Sendo especializada para um domínio específico, a arquitetura de referência possibilita aumentar o poder descritivo das estruturas de componentes influenciando diretamente na qualidade e no projeto de arquiteturas concretas e sistemas derivados dessas arquiteturas (ANGELOV *et al*., 2008). Essas arquiteturas têm impactado importantes aspectos do desenvolvimento de sistemas de software, tais como a produtividade e a qualidade desses sistemas (NAKAGAWA, 2012). BASS *et al*. (2003) afirmam que uma arquitetura de referência consiste de um repositório de conhecimento de um domínio específico que promove o reúso arquitetural e apóia no desenvolvimento de sistemas.

BASS *et al*. (2003) ressaltam que modelo de referência, estilo arquitetural e arquitetura de referência não são arquiteturas, mas são conceitos úteis que capturam elementos de uma arquitetura. Entretanto, modelo de referência e arquitetura de referência, bem como arquitetura concreta são conceitos relacionados e são diferenciados entre si pelo nível de abstração. O relacionamento entre esses conceitos é mostrado da Figura 2.4.

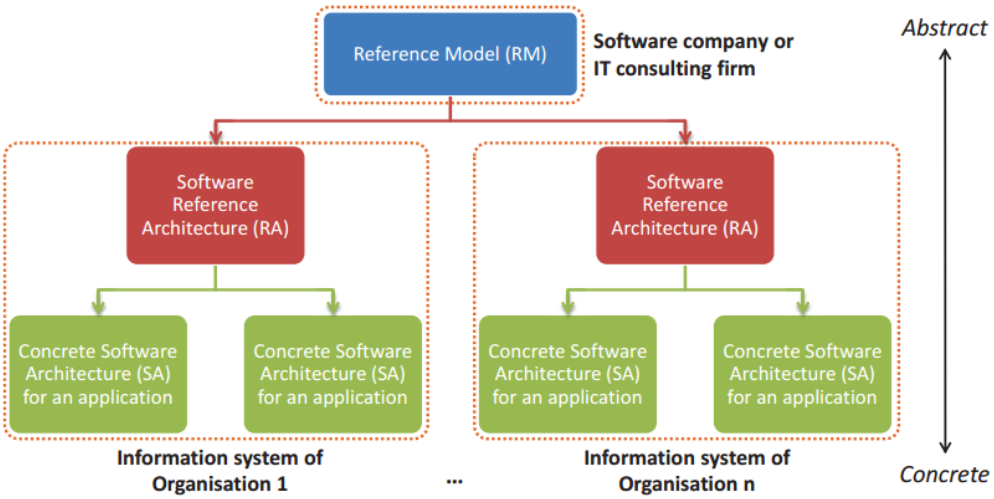

Figura 3.1: O relacionamento entre os conceitos presentes em arquitetura de software (MARTÍNEZ-FERNÁNDEZ *et al.*, 2013).

Na próxima subseção serão apresentados dois exemplos de arquiteturas de referência com o objetivo de ilustrar e melhorar o entendimento sobre esse conceito.

3.3 Arquitetura de Referência RefASSET

Com o objetivo de esclarecer melhor os conceitos que envolvem arquiteturas de referência, será apresentado a seguir, um exemplo desse tipo de arquitetura.

A RefASSET (*Reference Architecture for Software Engineering Tools*) (NAKAGAWA, 2006) é uma arquitetura de referência para ambientes de engenharia de software e foi construída para apoiar o desenvolvimento de ferramentas deste contexto de desenvolvimento. A arquitetura RefASSET considera a utilização da plataforma *Web* seguindo o conceito da arquitetura Cliente-Servidor, dividida em três camadas, e o padrão arquitetural MVC (*Model-View-Controller*). A camada de apresentação estabelece e gerencia a interface utilizada pelo usuário. A camada de aplicação possui as regras de negócio, ou seja, as funcionalidades principais que o sistema deve prover. A camada de persistência armazena e gerencia os dados que precisam ser persistidos.

Essa arquitetura serve de base para o estabelecimento de arquiteturas de referência específicas de ferramentas que automatizam atividades de Engenharia de Software. Por exemplo: ferramentas para gerência de requisitos, ferramentas de teste de software, ferramentas de análise e projeto, etc.

Na Figura 2.6 é apresentada uma visão geral da arquitetura RefASSET.

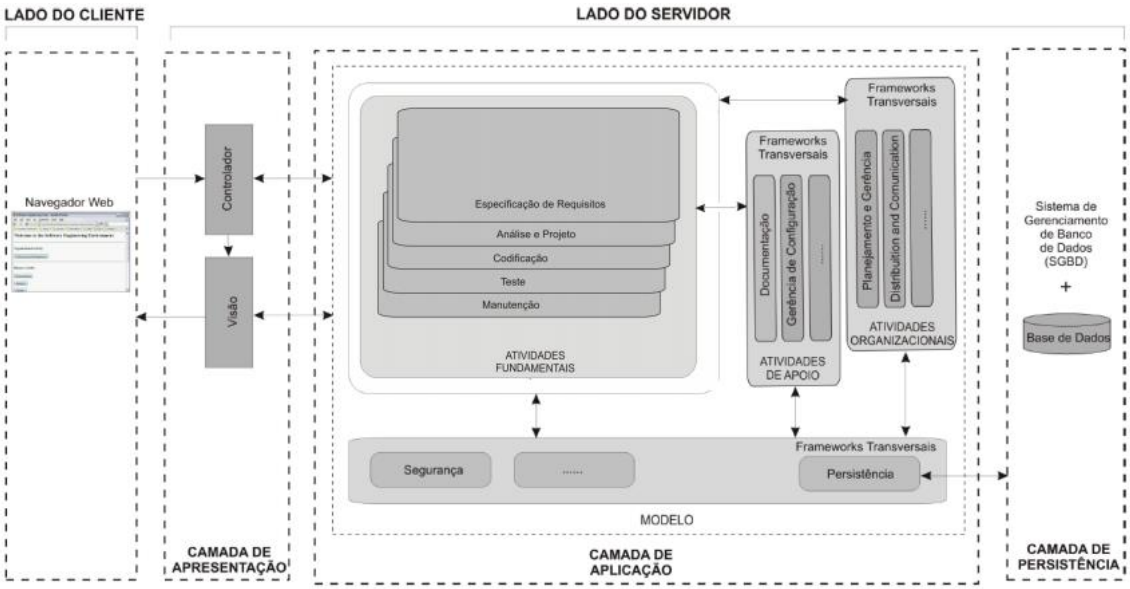

Figura 3.2: Visão geral da arquitetura de referência RefASSET (NAKAGAWA, 2006).

Os módulos que compõem a camada de aplicação foram projetados com base na norma ISO/EIC 12207 e correspondem ao conjunto de atividades básicas de desenvolvimento de software. A seguir, são apresentados os módulos que compõem a camada de aplicação:

Atividades Fundamentais:

- Gerência de Requisitos;
- Análise e Projeto;
- Implementação;
- Testes;
- Manutenção.

Atividades de Apoio: (melhoram a qualidade das atividades fundamentais)

- Documentação;
- Gerencia de configuração;
- Validação;
- Auditoria;
- Garantia da Qualidade.

Atividades Organizacionais: (interesses da organização para o planejamento do processo de software).

- Planejamento e Gerência;
- Distribuição e Comercialização;
- Infraestrutura;
- Treinamento.

Na Figura 2.7 é apresentado o processo de especialização e instanciação da RefASSET para o domínio de teste de software. Na etapa de especialização é necessário o conhecimento do domínio de teste de software. Com isso é obtida a arquitetura de referência RefTEST, que é mais adequada como mecanismo de apoio à contrução de ferramentas para teste de software.

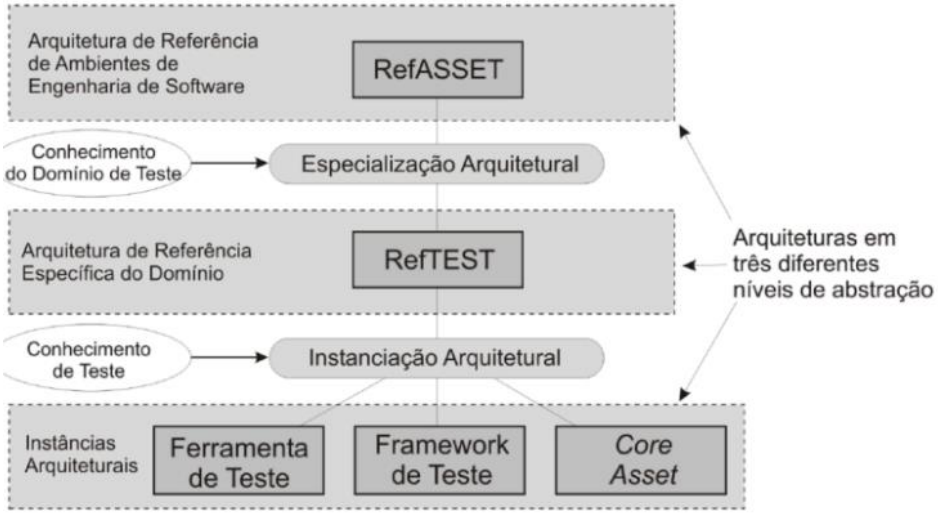

Figura 3.3: O processo de especialização da arquitetura RefASSET (NAKAGAWA, 2006).

Na etapa de especialização arquitetural é requerido um conhecimento no domínio de teste de software, uma vez que é necessário fazer um estudo sobre as ferramentas existentes e suas arquiteturas ou pesquisar arquiteturas de referência para do domínio.

Como pode ser observado na Figura 2.8, as alterações ocorreram no módulo “Atividades Fundamentais”. O módulo “Atividades Fundamentais” foi especializado com componentes para o domínio de teste de software. Com isso, é obtida uma arquitetura de ferramentas de teste que é adequada como mecanismo de apoio à construção de ferramentas desse domínio (NAKAGAWA, 2006). Uma vez que se tem a arquitetura de referência específica para o domínio de teste de software, podem-se criar instâncias arquiteturais para o desenvolvimento de ferramentas de teste, possibilitando, assim, o reuso do projeto arquitetural.

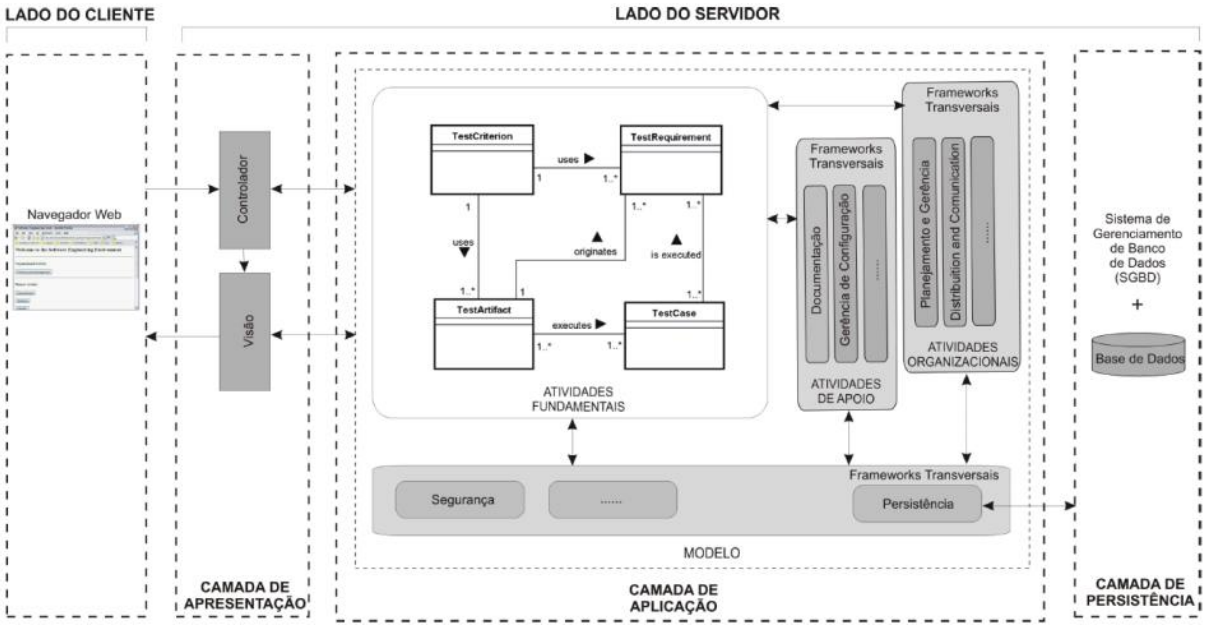

Figura 3.4: Arquitetura RefTest para ferramentas de teste de software.

Arquiteturas de software e arquiteturas de referência têm tido papel importante na determinação da qualidade de sistemas de software. Em particular, arquiteturas de referência, pois são consideradas como um conceito que agrega conhecimento de um domínio específico, promovendo o reúso desse conhecimento e dando apoio ao desenvolvimento de sistemas de software (NAKAGAWA, 2006).

Para apresentar graficamente arquiteturas de referência, um elemento muito importante são os artefatos que documentam a representação de uma arquitetura de software. Na próxima seção serão abordadas formas de representar e documentar arquiteturas de software encontradas na literatura.

3.4 Representação de Arquiteturas de Software

De acordo com Shaw e Garland (1996), uma arquitetura de software deve conter: a definição dos elementos de projeto que compõem o software; a descrição das interações entre estes elementos; os padrões de composição dos elementos; e um conjunto de restrições sobre estes padrões.

O conhecimento de uma arquitetura de software vem de sua representação por meio de documentos que apresentem os aspectos em termos estruturais. A documentação de uma

arquitetura consiste na representação de múltiplas visões arquiteturais. Uma visão arquitetural é uma abstração do sistema feita a partir de um conjunto de regras estabelecidas em um determinado ponto de vista. Ponto de vista é a perspectiva através da qual uma dada visão do sistema é construída.

A combinação de várias visões arquiteturais forma um modelo multidimensional em que cada visão pode atender a necessidades particulares de um ou mais participantes do sistema. A representação gráfica da arquitetura é útil para prover uma visão geral do sistema a ser desenvolvido, encontra-se na literatura vários esforços para descrever modelos de documentação, como o modelo "4+1" (Visão Lógica, Visão de Processos, Visão de Desenvolvimento e Visão Física) proposto por Kruchten, (1995), o modelo proposto por Hofmeister *et al*. (2000) (Visão Conceitual, Visão de Módulo, Visão de Execução, Visão de Código), inclusive uma normatização internacional regulamentada pelo padrão ISO/IEC 42010-2007.

O padrão ANSI/IEEE 1471-2000 surgiu como uma prática recomendada para descrição de arquiteturas de software. Em 2007, este padrão foi adotado pelo comitê técnico ISO/IEC JTC1/SC7 e denominado como ISO/IEC 42010-2007 um padrão internacional para engenharia de software nas práticas recomendadas para a descrição e documentação de arquitetura de software e sistemas. De acordo com o padrão, uma descrição arquitetural pode ser usada para:

- Expressão do sistema e a sua evolução;
- A comunicação entre os participantes do sistema;
- Avaliação e comparação de arquiteturas de forma consistente;
- Planejamento, gerenciamento e execução das atividades de desenvolvimento de sistemas;
- Expressão das características e dos princípios de um sistema para orientar a evolução;
- A verificação do cumprimento de restrições e relacionamentos na implementação do sistema.

O padrão ISO/IEC 42010-2007 apresenta um glossário definindo as seguintes terminologias para engenharia de software:

- **Arquiteto** (*architect*): a pessoa, equipe ou organização responsável por projetar a arquitetura de sistemas;
- **Descrição arquitetural** (*architectural description*): uma coleção de artefatos para

documentar uma arquitetura.

- **Arquitetura** (*architecture*): A organização fundamental de um sistema, compostas por seus componentes e relacionamentos, o ambiente e os princípios que direcionam o projeto e a evolução;
- **Sistema** (*system*): uma coleção de componentes organizados para realizar uma função ou conjunto de funções específicas;
- **Participantes** do sistema (*stakeholders*): Um indivíduo, equipe ou organização com interesses ou preocupações relativas a um sistema;
- **Visão** (*view*) : uma representação de todo um sistema a partir da perspectiva de um conjunto relacionado de preocupações;
- **Ponto de vista** (*viewpoint*): uma especificação das convenções para a construção e utilização de um ponto de vista. Um padrão ou modelo a partir do qual se desenvolvem visões individuais, estabelecendo os propósitos e interesses dos participantes e as técnicas para a sua criação e análise.

O padrão ISO/IEC 42010-2007 estabelece um *framework* conceitual para a descrição a arquitetura e define o conteúdo de uma descrição arquitetural.

Um outro aspecto muito importante para estabelecer arquiteturas de software são os requisitos arquiteturais. Esses requisitos devem ser identificados e especificados para direcionar as atividades no processo de produção da arquitetura. É importante destacar que muitos desses requisitos estão diretamente relacionados a atributos de qualidade. Na próxima seção serão apresentados conceitos referentes a requisitos arquiteturais e atributos de qualidade.

3.5 Requisitos Arquiteturais

Pesquisas na área de Engenharia de Software têm focado atenção nos processos, métodos, técnicas e ferramentas para o levantamento e gerenciamento de requisitos de software, e têm considerado arquiteturas de software como parte essencial dos sistemas de software. O estabelecimento de requisitos arquiteturais é certamente indispensável e determinante para a qualidade dos sistemas de softwares resultantes dessas arquiteturas (NAKAGAWA e MALDONADO, 2008; WASSERMAN, 1996).

Uma parte importante desse trabalho consiste na identificação e especificação de requisitos que sejam de relevância para estabelecer uma arquitetura de referência adequada ao domínio de RMAs. Estudos realizados sobre métodos e técnicas para atividades de identificação de requisitos contribuíram para esclarecer que quanto maior for o conhecimento acerca do domínio, mais eficaz será a tarefa de especificação de requisitos, e a qualidade do software está diretamente relacionada à conformidade deste com os requisitos especificados.

Um requisito descreve uma condição ou capacidade com a qual um sistema deve estar de acordo. Os requisitos são derivados diretamente das necessidades dos participantes do sistema e especificados formalmente em um documento ou contrato (KRUCHTEN, 2004).

De acordo com Pressman (2002), a qualidade do software depende dos padrões que especificam o desenvolvimento e orientam a maneira pela qual o software é produzido. Um conjunto de critérios de desenvolvimento deve ser seguido e o projeto deverá estar em conformidade com os requisitos identificados.

Na Engenharia de Software os requisitos são classificados em requisitos funcionais e requisitos não funcionais. Além dessa classificação, de acordo com Eeles (2005) existe também a classificação denominada “FURPS +” que considera os requisitos em categorias de atributos de qualidade do software. A sigla FURPS é um acrônimo para as seguintes categorias:

- Funcionalidade (*Funcionality*);
- Usabilidade (*Usability*);
- Confiabilidade (*Reliability*);
- Desempenho (*Performance*);
- Suportabilidade (*Supportability*).

O "+" em FURPS + estende ainda mais as categorias de requisitos adicionando os requisitos de projeto, requisitos de implementação, Portabilidade, Extensibilidade e Reusabilidade.

- **Funcionalidade**: reúne todos os requisitos funcionais. Estes requisitos geralmente representam as principais características do produto;
- **Usabilidade** está relacionada com:
  - Acessibilidade: Facilidade com que se utilizam as diferentes interfaces do sistema;
  - Estética: Qualidade estética das interfaces do sistema;

    - Coerência: O uso consistente de mecanismos empregados na interface do usuário.
- **Confiabilidade** está relacionada aos atributos de disponibilidade, precisão e recuperação de falhas;
- **Desempenho** está relacionada a rendimento, tempo de resposta, tempo de recuperação, tempo de inicialização e tempo de desligamento;
- **Suportabilidade** está relacionada aos atributos de escalabilidade, adaptabilidade, facilidade de manutenção, compatibilidade, facilidade de instalação e testes.

Especificar os requisitos arquiteturais de acordo com essas categorias é uma abordagem que vai contribuir para destacar os atributos de qualidade de uma arquitetura de software. Nesse contexto é importante observar que no domínio de RMAs, Orebäck e Christensen (2003) destacam que as características de extensibilidade e escalabilidade significam o apoio à adição de novos módulos de software, bem como novos dispositivos eletromecânicos de hardware. Esse é um aspecto muito importante, uma vez que sistemas robóticos tendem a evoluir tanto em termos de hardware como em software.

A adição de novos sensores é praticamente uma atividade padrão no desenvolvimento de sistemas de software para RMAs. Por outro lado, em se tratando de evolução de software em sistemas de controle baseado em comportamentos, a adição de novos comportamentos também é uma prática comum.

Outra forma de elicitar requisitos encontrada na literatura (LADDAGA, 2000; SALEHIE e TAHVILDARI, 2009; AFFONSO e NAKAGAWA, 2013), são as seis questões, conhecidas na área de Qualidade como modelo 5W1H. Essas questões formam um conjunto metódico de perguntas cujas respostas são consideradas essenciais para coleta e seleção de informações de forma criteriosa e objetiva.

No contexto de uma investigação para a captura de requisitos arquiteturais referentes à característica de evolução em uma arquitetura de referência para um projeto de software no domínio de robôs móveis autônomos, as perguntas elaboradas segundo o modelo 5W1H ficariam da seguinte forma:

Tabela 3.1: O modelo 5W1H, adaptado de AFFONSO e NAKAGAWA (2013).

5W1H	Questões	Escopo
What?	O que deverá evoluir?	Sensores, atuadores, comportamentos, etc
Where?	Onde a evolução deverá ocorrer?	Componentes arquiteturais, hardware, etc.
Who?	Quem deve realizar a evolução?	Sistemas autônomos, humanos, ambos, etc.
When?	Quando a evolução deverá ser aplicada?	Em tempo de execução, quantas vezes, etc.
Why?	Porque deve evoluir?	Erros no projeto, novos requisitos, etc.
How?	Como a evolução será realizada?	Planos de adaptação, políticas de adaptação, etc.

Nas atividades de engenharia de software, o desenvolvimento de um determinado sistema de software para áreas administrativa e comercial como no caso de um ERP, pode-se contar com elementos típicos do domínio, tais como clientes, documentos, entre outros, para a atividade de extração dos requisitos. No caso de software para RMAs, quando se trata da extração dos requisitos para o desenvolvimento de arquiteturas de referência visando aspectos de qualidade e evolução, outras fontes de informação são requeridas, inclusive fontes mais abrangentes, uma vez que a arquitetura de referência é base para o desenvolvimento de um conjunto de sistemas de software de um determinado domínio.

Para a identificação e especificação dos requisitos arquiteturais deste trabalho, foram investigadas diversas fontes de informação e selecionadas as mais relevantes que contribuíssem para extração dos requisitos no domínio de RMAs, entre elas, exemplos de código fonte (GOOGLE CODE, 2013), a API LeJOS (LeJOS, 2013) além de diversos livros e artigos publicados na área. O modelo 5W1H foi utilizado neste trabalho e ajudou a identificar cenários que ocorrem no domínio de RMAs. Esses cenários serão abordados e detalhados de forma mais profunda mais adiante em outra seção.

Apesar de haver métodos conhecidos na Engenharia de Software para especificação de requisitos, inclusive métodos específicos para requisitos arquiteturais em arquiteturas de referência como o ProSA-RA apresentado por (NAKAGAWA e MALDONADO, 2013), neste trabalho como o objetivo principal é a especificação de requisitos arquiteturais que habilitem características de evolução à arquitetura de referência aqui proposta, optou-se por uma metodologia de especificação de requisitos baseada em cenários de evolução.

3.6 Avaliação de Arquiteturas de Software

Um aspecto importante quando se trata de arquiteturas de software são os requisitos de qualidade que a arquitetura escolhida deve satisfazer. Em geral, a escolha de uma arquitetura é guiada por esses requisitos que devem ser atendidos. Por exemplo, se escalabilidade é importante para o sistema, então a arquitetura escolhida deve ser flexível a ponto de entregar a mesma qualidade do serviço mesmo quando o número de requisições aumentar bastante. Muitas vezes, o atendimento a um determinado requisito de qualidade compromete outros, assim deve-se cuidar para obter um bom balanceamento entre eles.

Uma forma de avaliar arquiteturas de software é o método SAAM. O objetivo desse métodos é indicar se uma determinada arquitetura estabelecida para um sistema atenderá os requisitos de qualidade esperados.

3.6.1 Método de Avaliação SAAM

O método de análise de arquiteturas de software SAAM (*Scenario-Based Architecture Analysis Method*) foi desenvolvido por Kazman *et al.* (1994). Esse método utiliza a descrição e investigação de cenários para conduzir avaliações de arquiteturas de software quanto a aspectos de qualidade. O objetivo deste método é fornecer meios para caracterizar o quão bem um projeto arquitetural atende a atributos de qualidade específicos colocados por um conjunto de cenários. Um cenário é uma sequência específica de etapas que envolvem o uso ou a modificação do sistema. As principais etapas do SAAM são:

- **Desenvolvimento dos Cenários**: Um cenário é definido como uma descrição resumida de um determinado uso pretendido de um sistema. Diferentes tipos de cenários de avaliação devem ser desenvolvidos nesta fase. Cada um tem de ser definido com o propósito explícito de revelar como o sistema se comporta em relação a um determinado tipo de requisito. Os cenários são normalmente desenvolvidos pelas partes interessadas (usuários, administrador de sistemas, desenvolvedores, etc.).

- **Descrição das Arquiteturas Candidatas**: Nessa fase devem ser apresentadas todas as arquiteturas, quando houver mais de uma, candidatas a atender aos propósitos do sistema em questão. As arquiteturas devem ser bem descritas, de forma bem simples

utilizando-se uma notação onde a estrutura de componentes e relacionamentos fique bem compreendida a todos os participantes interessados.

- **Classificação dos cenários**: Os cenários podem ser de dois tipos: (i) cenários diretos, envolvem situações de uso suportadas pela arquitetura e (ii) cenários indiretos, envolvem situações que requerem ajuste na arquitetura para que possam ser suportados. O cenário direto é adequado para avaliações de comportamento e de desempenho da arquitetura, enquanto que o cenário indireto vai dar um passo adiante no sentido de avaliar impactos de mudança. Portanto, essa segunda categoria indica a necessidade e a extensão das modificações.
- **Avaliação dos cenários indiretos**: as mudanças necessárias para que a arquitetura possa apoiar cada cenário indireto são especificadas e ponderadas em relação a dificuldade e o custo de execução.
- **Avaliação da interação dos cenário**s: Uma interação de dois cenários indiretos ocorre quando eles exigem uma mudança no mesmo componente da arquitetura. Levando-se em conta a relação semântica entre os cenários, uma exploração de sua interação é indicada para revelar a complexidade estrutural da arquitetura, e o acoplamento/coesão de seus componentes.
- **Avaliação global**: Nesta etapa é produzido um *ranking* das arquiteturas candidatas. Este deve refletir o comportamento global da arquitetura no que diz respeito a todos os cenários (atributos de qualidade). Os avaliadores devem realizar a pontuação final de acordo com a importância de cada cenário.

3.6.2 Método de Avaliação ATAM

O método de análise de arquiteturas de software ATAM foi desenvolvido por Kazman *et al.* (2000). O ATAM é claramente uma especialização e evolução do método SAAM e seu o objetivo é avaliar as consequências das decisões arquiteturais frente aos atributos de qualidade dos requisitos arquiteturais. Em arquiteturas de software, ao se modificar uma característica de qualidade outras também são alteradas. Em virtude disso, é necessária uma analise para se encontrar a melhor alternativa que minimize os efeitos das consequências de uma decisão, ou seja produzir o melhor *tradeoff* das características relevantes ao contexto em que a arquitetura de software está inserida.

O método ATAM se baseia em três conceitos-chave:

- **Atributos baseados em estilo arquitetural.** O ATAM utiliza um estilo arquitetural chamado estilo arquitetural baseado em atributo (ABAS). O ABAS é um método para elaboração de arquiteturas de software baseado em estilos arquiteturais. O foco principal está nos tipos de componentes e padrões de interação que são particularmente relevantes para os atributos de qualidade, tais como o desempenho, a confiabilidade, a segurança ou a disponibilidade. O método consiste de um framework conceitual para realizar uma análise qualitativa e quantitativa sobre estilos arquiteturais relacionados a atributos de qualidade. Estilos arquiteturais podem ser usados como blocos de construção para projetar e analisar as arquiteturas de software. O enfoque é tornar mais explícitos os fatores que influenciam um projeto arquitetural. A composição de estruturas por meio de blocos vai oferecer a base para um raciocínio mais preciso e eficiente para a tomada de decisões sobre um determinado projeto arquitetural (KLEIN, 1999).
- **Caracterização de atributos de qualidade**, divididos em três categorias:
  - Estímulos externos
  - Decisões arquiteturais
  - Respostas
- **Cenários**:
  - Cenários de caso de uso: Envolvem o uso de sistemas existentes e são usados para obter informações de elicitação;
  - Cenários de crescimento: Abrangem mudanças previstas para a arquitetura;
  - Cenários exploratórios: Abrangem alterações extremas onde se espera um "estresse" do sistema.

A realização do método ATAM é composta de quatro fases, divididas em nove etapas, conforme é apresentado a seguir:

Fase 1: Apresentação

1. **Apresentação**: O método deve ser apresentado à equipe que irá utilizá-lo (equipe de arquitetos de software, desenvolvedores, administradores, gerentes, testadores, etc).
2. **Apresentação dos objetivos de negocio:** O gerente de projetos descreve os objetivos de negócio que estão motivando o esforço de desenvolvimento e, portanto, quais serão as diretivas primárias de arquitetura, por exemplo, de alta disponibilidade ou tempo de comercialização ou de alta segurança.

3. **Apresentação da arquitetura atual**. O arquiteto irá descrever a arquitetura proposta, centrada na forma como ele trata as diretivas de negócios. Investigação e Análise.
4. **Identificação das abordagens arquiteturais**. As abordagens arquiteturais são identificadas pelo arquiteto, mas não são analisadas.

Fase 2: Investigação e Análise

5. **Gerar a árvore de atributos qualidade**: A árvores de atributos de qualidade fornece um mecanismo *top-down* para traduzir diretamente e de forma eficiente as diretivas de negócio de um sistema em cenários concretos de atributos de qualidade. Nessa etapa, a árvore qualidade é criada e os fatores de qualidade (desempenho, disponibilidade, segurança, modificabilidade, etc) são extraídos e caracterizados como: estímulos, decisão ou de respostas e, por fim são priorizados.
6. **Analisar as abordagens arquitetônicas**: Com base nos fatores de prioridade identificadas na etapa 5, as abordagens de arquitetura que suportam esses fatores são extraídas e analisadas (por exemplo, uma abordagem arquitetônica que visa cumprimento de metas de desempenho será submetida a uma análise de desempenho). Durante esta etapa, os riscos de arquitetura, pontos de sensibilidade e pontos de *trade-off* são identificados.

Fade 3: Testes

7. ***Brainstorm* de cenários**. Baseado nos cenários de exemplo gerados na etapa 4 da árvore de qualidade, um conjunto maior de cenários é induzida a partir de todo a equipe envolvida. Este conjunto de cenários é priorizado novamente por meio de um processo de votação que envolve todo o grupo.

8. **Analisar as abordagens arquitetônicas**. Este passo reafirma a etapa 6, mas aqui os cenários priorizados na Etapa 7 são considerados casos de teste para a análise das abordagens arquiteturais determinadas até o momento. Com esses cenários de teste pode-se descobrir novas abordagens arquitetônicas, riscos, pontos de sensibilidade e pontos de *trade-off*, que deverão documentados.

Fase 4: Reportar os Resultados

9. **Apresentação dos resultados**. Com base nas informações colhidas nos ATAM (estilos, cenários, questões específicas de atributos, a árvore de qualidade, riscos, pontos de sensibilidade, compensações) a equipe ATAM apresenta os resultados para as partes interessadas e, escreve um relatório que detalha essas informações, juntamente com a proposta e todas as estratégias da solução.

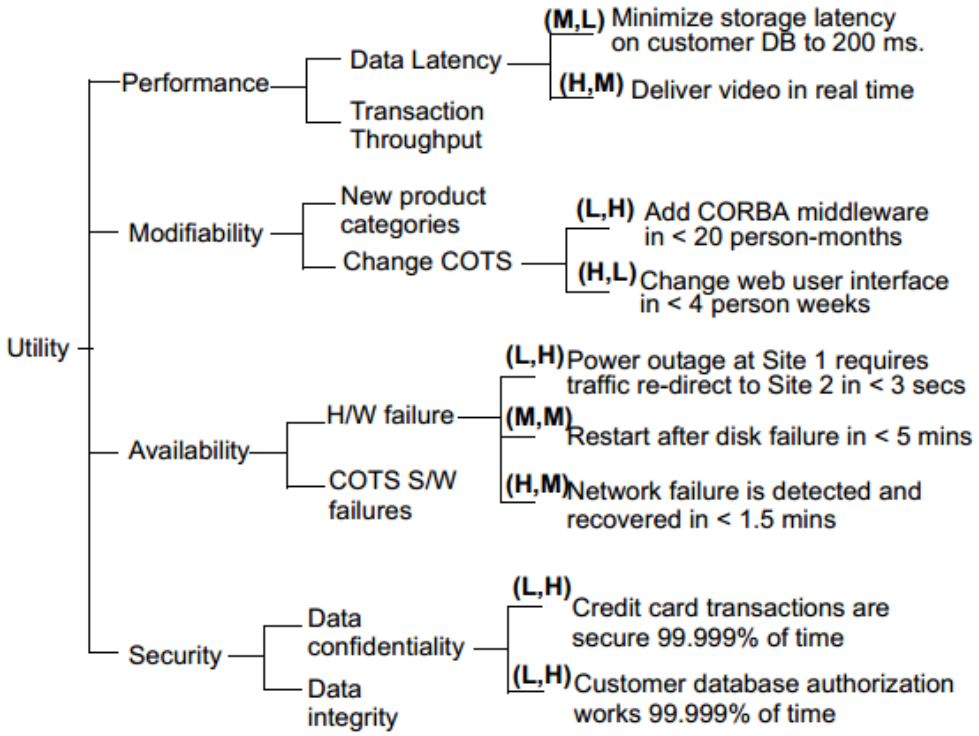

Figura 3.5: Arvore de atributos de qualidade, Kazman *et al.* (2000).

Conforme é mostrado na Figura 2.9, cada nó folha recebe um fator de prioridade, normalmente a abordagem mais comum é atribuir os referenciais Alto, Médio e Baixo (*High*, *Medium*, *Low*). A priorização da árvore de atributos de qualidade é feita em duas dimensões, (i) pela importância que o atributo representa a arquitetura e (ii) e pelo grau de risco que envolve a incorporação do atributo de qualidade a arquitetura, ou seja, a complexidade de modificação necessária para incorporar o atributo de qualidade a arquitetura. Por exemplo, o cenário de desempenho: "*minimize storage latency on customer"* recebeu a prioridade (*M,L*) médio, baixo, o que significa que esse cenário é de importância média para o projeto e de baixo risco para incluir na arquitetura. Já o cenário: "*deliver video in real time"* recebeu a prioridade (H,M) alta, média, o que significa que esse cenário é de alta importância para o projeto e de risco médio para sua inclusão na arquitetura.

Nesse capítulo foram abordados conceitos de engenharia de software muito importantes para o estabelecimento de Arquiteturas de Referência. Uma parte da arquitetura de referência proposta neste trabalho consiste em propor uma estrutura com os elementos necessários para prover as funcionalidades de autoadaptação aos RMAs. No próximo capítulo será apresentado o conceito de Teoria do Controle que, mais especificamente, aborda as técnicas utilizadas em sistema autoadaptativos.

Capítulo 4

TRABALHOS RELACIONADOS

Com o objetivo de entender melhor o papel de arquiteturas de referência na produção de software para sistemas autoadaptativos, foi realizada uma pesquisa buscando propostas que utilizam loops de controle como abordagem para realizar autoadaptação em tempo de execução. Foram encontradas duas propostas, uma no domínio da robótica móvel a 4D/RCS, e outra no domínio de sistemas para Web, a RA4SaS.

4.1 Arquitetura 4D/RCS

A arquitetura 4D/RCS (ALBUS, 2002) fornece um modelo de referência para veículos militares não tripulados. É uma arquitetura híbrida que apoia os princípios da arquitetura deliberativa com capacidade de planejamento e ação, e também suporta os princípios da arquitetura reativa com capacidade autonômica para responder e reagir aos estímulos do ambiente. É uma proposta para o desenvolvimento de sistemas móveis autônomos para locomoção em ambientes desconhecidos. A metodologia consiste na decomposição e análise do ambiente para que o controle e locomoção do veículo possa ser planejado e executado de forma autônoma e em tempo real.

Como é mostrado na Figura 3.1, a4D/RCS é uma arquitetura dividida em camadas e organizada de forma hierárquica. Cada camada possui vários loops de controle que neste trabalho são chamados de nós computacionais. Cada nó contém elementos de processamento sensorial (SP), modelagem do ambiente (WM), juízo de valor (VJ) e geração de comportamento (BG). Nos níveis mais baixos, o tempo de resposta é menor e a resolução é alta com os elementos gerando comportamento reativo na busca de objetivos. Nos níveis mais

altos da arquitetura são formulados conceitos mais abstratos, com isso o tempo de resposta nesses níveis é maior para geração de comportamento deliberativo e para definição de metas.

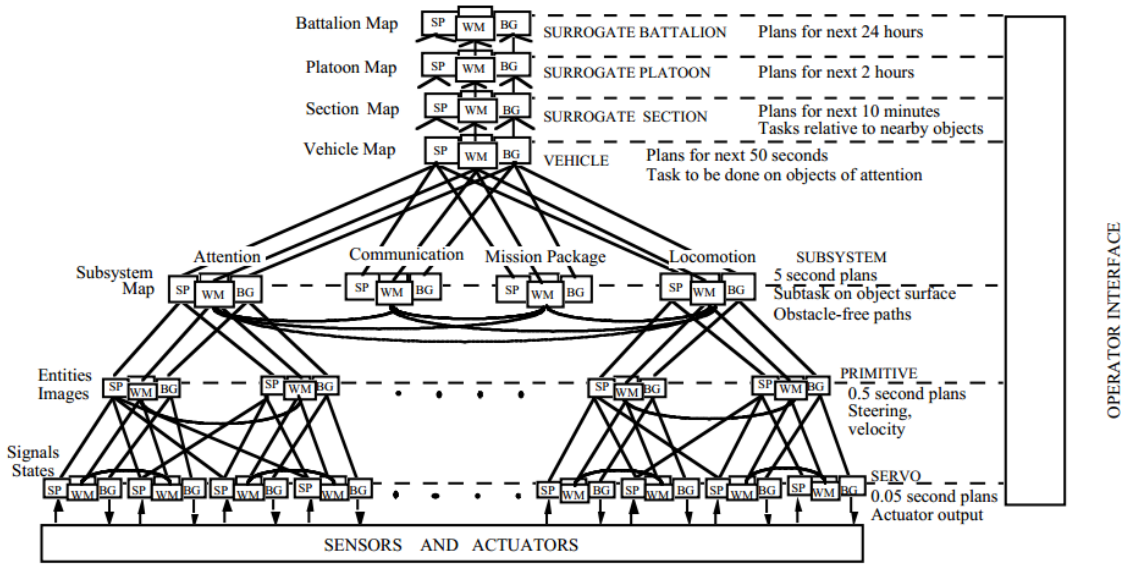

Figura 4.1: As camadas da arquitetura 4D/RCS.

Os níveis de hierarquia nas camadas da arquitetura 4D/RCS:

- ***Servo***: Grupos de comandos são decompostos em sinais de controle para os atuadores.
- ***Primitive***: Vários grupos de atuadores são coordenados e as interações dinâmicas entre esses grupos são consideradas.
- ***Subsystem***: Todos os componentes dentro de um subsistema estão sendo coordenados e o planejamento leva em consideração questões como desvio de obstáculos e controle de velocidade.
- ***Vehicle***: Todos os subsistemas dentro de um veículo ão coordenados para gerar comportamentos táticos.
- ***Section***: Vários veículos são coordenados para gerar comportamentos táticos em conjunto.
- ***Platoon***: várias seções que contêm um total de 10 ou mais veículos de diferentes tipos são coordenadas para gerar táticas de pelotão.
- ***Battalion***: Vários pelotões contendo um total de 40 ou mais veículos de diferentes tipos são coordenados para gerar táticas de batalhão.

A arquitetura fornece um modelo que estabelece como os componentes de software devem ser identificados, organizados e como devem interagir. Nos níveis mais baixos estes elementos geram comportamento reativo em busca de objetivos. Em níveis mais altos, ela permite a definição de objetivos por meio do comportamento deliberativo. A estrutura interna de um nó é mostrada na Figura 3.2.

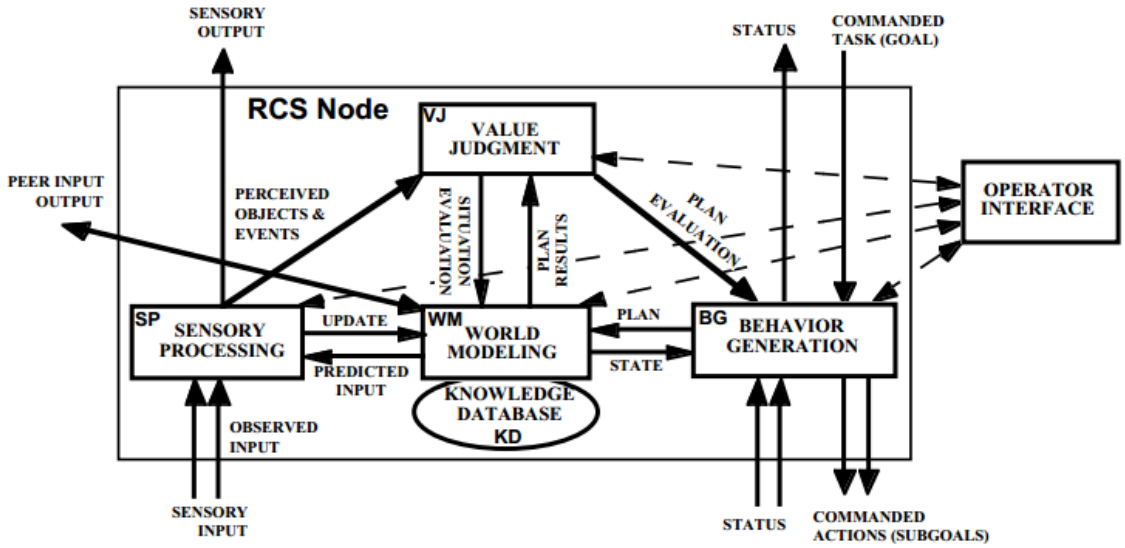

Figura 4.2: Estrutura interna de um nó na arquitetura 4D/RCS.

Um loop de controle pode ter uma interface de operador. A interface permite a um operador humano enviar comandos de entrada, substituir ou modificar o comportamento do sistema, observar os valores das variáveis de estado, imagens, mapas. A interface também pode ser utilizada para programação, depuração e manutenção.

De acordo com os autores a arquitetura 4D/RCS tem as seguintes propriedades:

1. Define os elementos funcionais, subsistemas, interfaces, entidades, relacionamentos e unidades de informação em sistemas para veículos inteligentes.

2. Apoia a seleção de metas, estabelecimento de prioridades e regras de engajamento, geração de planos, decomposição de tarefas, agendamento de atividades. Fornece um retorno para ser incorporado aos processos de controle de modo que ambos os comportamentos deliberativos e reativos possam ser combinados em um único sistema integrado.

3. Apoia o processamento de sinais dos sensores no conhecimento de situações e relacionamentos. Fornece armazenamento do conhecimento em formas de

representação que podem apoiar o raciocínio, a tomada de decisão e controle inteligente.

4. Fornece informação estática (de longo prazo) e dinâmica (a curto prazo) para representar o conhecimento necessário para descrever o ambiente e o estado do campo de batalha e os sistemas de veículos inteligentes operando nele.

5. Apoia a transformação de informações de sinais de sensores em representações simbólicas e descritivas de objetos, eventos e situações, incluindo relações semânticas,. Suporta transformações a partir de formas visuais para descritivas e vice-versa.

6. Apoia a aquisição (ou aprendizagem) de novas informações e a integração e consolidação dos conhecimentos para uma base de dados.

7. Prevê o cálculo de custos e benefícios, a avaliação de incerteza e risco, a avaliação dos planos e os resultados comportamentais, e a otimização de leis de controle.

Os autores destacam que apenas os níveis mais baixos foram totalmente implementados. Como pode ser observado os elementos são organizados na estrutura de um *feedback loop*. Cada nó funciona como um loop de controle, lendo dados dos sensores e enviando comandos aos atuadores. Não fica claro no trabalho como é a interface de comunicação entre os loops de controle, uma vez que os loops podem estar conectados tanto à camada de sensores e atuadores como a outros loops. Não foi demonstrado como é o processo de geração de comportamento, e também não foi especificado quais as técnicas de representação do ambiente.

4.2 Arquitetura de Referência RA4SaS

Arquitetura RA4SaS (AFFONSO E NAKAGAWA, 2013) é uma arquitetura de referência baseada no recurso de reflexão para inspeção e modificação de entidades de software em tempo de execução. Apesar de não ser uma arquitetura para o domínio de RMAs, essa proposta é interessante para o contexto deste trabalho, pois a arquitetura RA4SaS utiliza loops de controle para realizar a autoadaptação. Outro ponto importante é que o contexto de adaptação da RA4SaS ocorre diretamente na estrutura das entidades de software o que

diferencia um pouco do método de adaptação apresentado na proposta deste trabalho, criando uma perspectiva interessante de comparação. A RA4SaS contém duas funções principais: software de monitoramento e autoadaptação de entidades de software em tempo de execução.

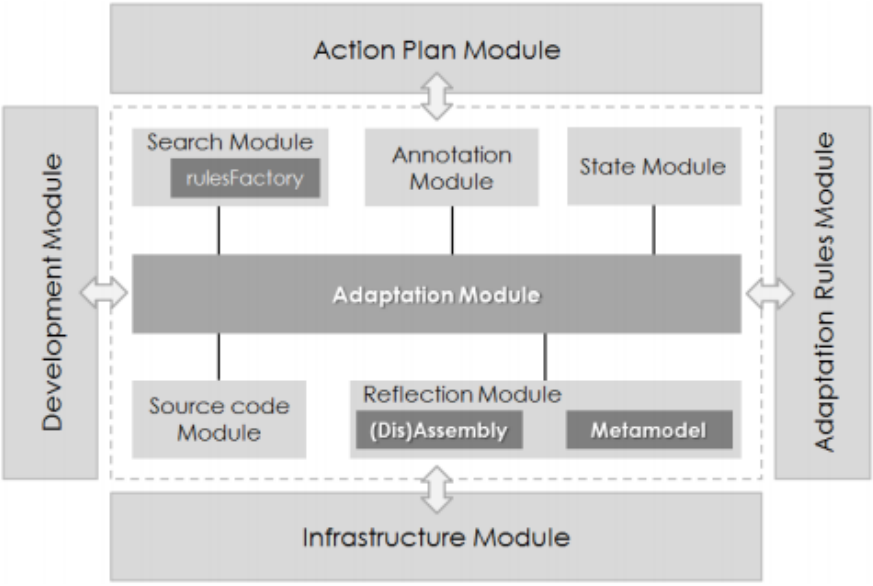

Figura 4.3: A arquitetura de referência RA4SaS (AFFONSO e NAKAGAWA, 2013).

A arquitetura de referência RA4SaS é composta dos seguintes módulos:

- ***Development Module*:** Esse módulo fornece um conjunto de diretrizes para o desenvolvimento de software que atuam nas fases de análise de requisitos, design, implementação e adaptação. Para gerenciar a adaptação são utilizadas métrica que permitem avaliar a complexidade e o comportamento das entidades. Essas métricas visam controlar a coesão e a granularidade de entidades, de modo que eles não perdem a sua flexibilidade e capacidade de adaptação.
- ***Action Plan Module***: Esse módulo é responsável por criar um plano de ação que estabelece como será adaptado e como aplicar as alterações na entidade de software para alcançar o melhor resultado. O processo de execução deve fornecer mecanismos para execução do plano de ação estabelecido.
- ***Adaptattion Rules Module***: Esse módulo é responsável por extrair automaticamente as regras de adaptação da entidade de software.
- ***Infrastructure Module***: Esse módulo fornece apoio para a adaptação de entidades de software em tempo de execução. Para isso, dispõe de um conjunto de mecanismos

como a compilação dinâmica e carregamento dinâmico, diagnóstico de problemas e correção de erros.

- ***Search Module***: Esse módulo auxilia na busca por entidades de software no ambiente de execução quando uma atividade de adaptação é invocada.
- ***Annotation Module***. Esse módulo auxilia o engenheiro de software na definição do nível de adaptação das entidades de software.
- ***State Module***: Esse módulo tem o objetivo de preservar o estado atual da entidade de software em execução.
- Source Code Module:
- ***Reflection Module***. Esse módulo auxilia o processo de "desmontagem" e "montagem" da entidade de software. O processo de desmontagem é realizado com as informações estruturais e comportamentais (atributos e métodos) recuperados através da reflexão e inserido no submodulo Metamodelo.

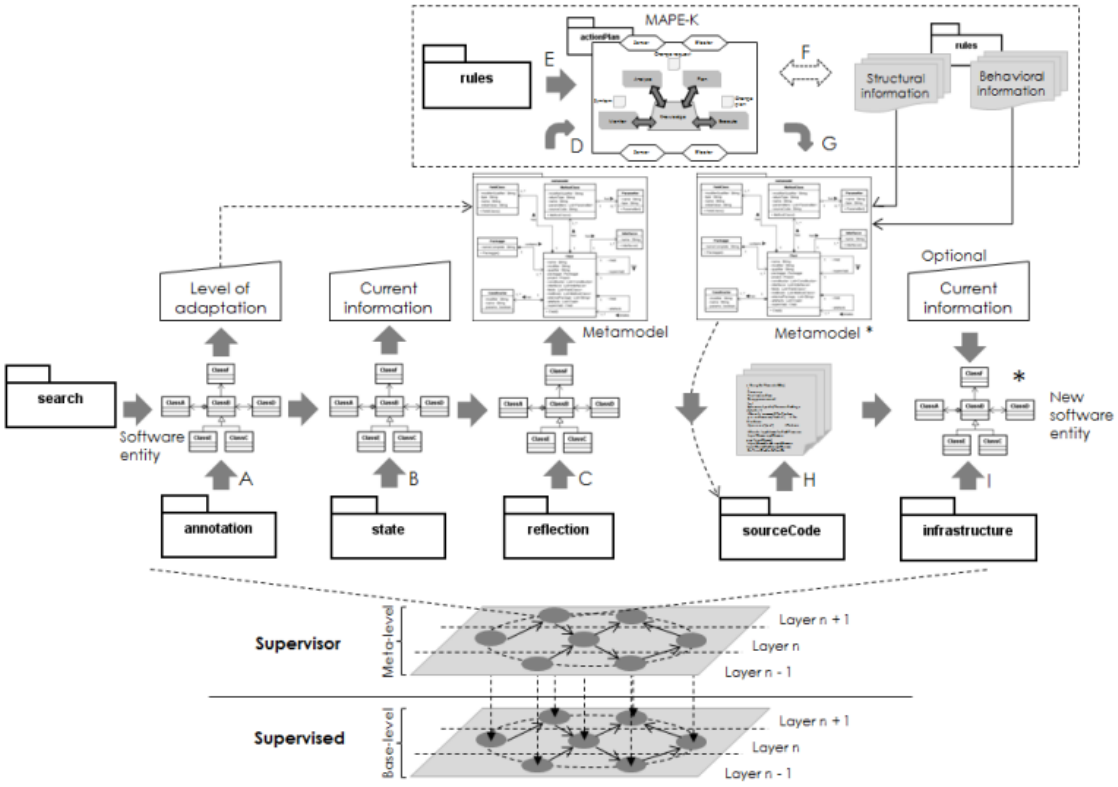

Figura 4.4: Processo de adaptação da arquitetura RA4SaS (AFFONSO e NAKAGAWA, 2013).

Um ponto importante a destacar, para o contexto deste trabalho, é que na arquitetura RA4SaS o processo de adaptação é bem detalhado. Na Figura 4.4 é ilustrado como ocorre o processo de autoadaptação.

O processo de adaptação na arquitetura de referência RA4SaS é organizado em nove etapas:

- **Etapa A**: o nível de adaptação suportado pela entidade de software é verificado, uma vez que irá ser utilizado na produção do metamodelo (etapa C).
- **Etapa B**: o estado de execução atual da entidade de software naquele momento é preservado.
- **Etapa C**: a entidade de software é "desmontada" e um metamodelo é instanciado. Este metamodelo tem a informação estrutural e comportamental da entidade de software, e também seu nível de adaptação obtido na etapa A.
- **Etapa D**: é estabelecido um plano de ação para realizar a adaptação entidade.
- **Etapa E**: O plano de ação deve estabelecer critérios de adaptação com base de regras que fornecem orientações e metas.
- **Etapa F**: O plano de ação também deve estar de acordo com requisitos estruturais e comportamentais.
- **Etapa G**: Com base nesses processos e nas informações de adaptação, um novo metamodelo é gerado, e transferido para o módulo de código fonte.
- **Etapa H**: o código fonte de entidade de software gerado e compilado de forma que ele pode ser inserido novamente no ambiente de execução.
- **Etapa I**: Nesta última etapa as informações preservadas na etapa B são reinseridas na nova entidade para substituir a antiga no ambiente de execução.

As arquiteturas de referencia, 4D/RCS e RA4SaS apresentadas, assim como a arquitetura proposta neste trabalho, oferecem funcionalidades de autoadaptação em tempo de execução, com enfoque em loops de controle. A abordagem de adaptação utilizada pela arquitetura 4D/RCS consiste perceber, planejar e atuar adaptando-se às mudanças do ambiente, alterando o comportamento do sistema em tempo de execução. No entanto, a abordagem utilizada pela arquitetura RA4SaS é um pouco diferente, pois, inicialmente a adaptação é aplicada diretamente às entidades de software que compõem o sistema. Seu estado original é capturado, e a alteração é efetuada na estrutura básica da entidade, com adição de novos atributos e novos métodos. Posteriormente, as entidades alteradas precisam

ser compiladas, o módulo de adaptação deve restaurar seu estado original e retomar o ponto de execução do sistema. Essa abordagem é interessante pois apresenta a autoadaptação em um outro nível, modificando não somente comportamento e configuração, mas a estrutura de entidades que compõem o sistema. Esse tipo de abordagem requer um maior tempo de resposta do sistema devido às fases inerentes ao processo de adaptação. Isso contrasta um pouco com as aplicações de RMAs nas quais o desempenho é um atributo imprescindível na maioria das situações. Essas e outras comparações entre as duas arquiteturas de referência apresentadas neste capítulo, inclusive a proposta neste trabalho, estão reunidas na tabela 3.1.

Tabela 4.1: Tabela comparativa entre as arquiteturas de referência.

Características	SARA-^{MR}	RA4SaS	4D/RCS
Destinada a Facilitar Manutenções/Evoluções	Sim	NI	NI
Emprego de Padrões de Projeto	Sim	NI	NI
Específico de Robôs Móveis Autônomos	Sim	Não	Sim
Suporte a Processamento de Sensores	Sim	Não	Sim
Uso da arquitetura MIAC	Sim	Não	Apenas na primeira e segunda camada
Uso da arquitetura MRAC	Sim	Sim	Sim
Suporte a Loop de Controle Individual	Sim	Sim	Sim
Suporte a Múltiplos Loops	Sim	Não	Sim
Emprego de Metas e padrões	Sim	Sim	Sim
Políticas ou Regras de adaptação	Sim	Sim	Não
Plano de ação	Sim	Sim	Sim
Tipo de Adaptação	Comportamento	Entidades de software e comportamento	Comportamento
Interface com usuário	Não	Sim	Sim
Tempo real	Sim	Sim	Sim
Mais de um Domínio	Não	Sim	Não
Implementada	Sim	Parte	Parte
Permite alto Desempenho	Não	Não	Nas Camadas inferiores

As arquiteturas RA4SaS e 4D/RCS não evidenciam um atributo de qualidade específico, como por exemplo, desempenho ou evolução/manutenção. Considerando que arquiteturas são criadas com vistas a atender determinados atributos de qualidade, não ficam evidentes os principais benefícios do uso dessas arquiteturas.

A característica de permitir múltiplos loops se refere à capacidade de implementação de mais de um loop de controle funcionando simultaneamente. Essa categoria não é mencionada na arquitetura RA4SaS, mas é uma funcionalidade apresentada pela arquitetura 4D/RCS, bem como pela arquitetura proposta neste trabalho de mestrado.

A característica de "Metas e padrões" se refere ao estabelecimento de prioridades e regras de adaptação, a geração de planos e, também a programação de atividades. As três arquiteturas apresentam referências a essas características.

A característica de "Alto desempenho" se refere a capacidade da arquitetura de reagir com um tempo curto de resposta na atividade de adaptação. Manter a taxa de latência em níveis baixos, com curtos intervalos de tempo para completar o ciclo do loop de controle. No caso da arquitetura de referência proposta neste trabalho, como será mostrado no capítulo de estudo de caso, e também nos níveis mais baixos da arquitetura 4D/RCS, o tempo de respostas fica na casa dos milissegundos. No caso da arquitetura RA4SaS, apesar de não haver referência ao tempo de conclusão do ciclo de adaptação, o "não" para essa característica foi estimada com base nas fases do processo de adaptação, que envolve a interrupção do ponto de execução do sistema, captura do estado da entidade, modificação das características estruturais da entidade, compilação, devolução do estado original e retomada do ponto de execução do sistema. Essas atividades, não são triviais, variam conforme a complexidade estrutural da entidade a ser adaptada, e por isso recebeu uma estimativa com tempo de reposta maior que um segundo.

Nesse capítulo foram apresentadas duas propostas de arquiteturas de referência para sistemas autoadaptativos, assim como uma comparação entre elas. Nos próximos capítulos, a arquitetura de referência proposta neste trabalho de mestrado será apresentada em uma sequência que descreve o projeto de estabelecimento da arquitetura por meio de cenários de evolução, a apresentação detalhada da arquitetura de referência e, por fim, um estudo de caso apresentando dois ensaios envolvendo aplicações para RMAs.

Capítulo 5

CENÁRIOS DE EVOLUÇÃO NO DOMÍNIO DE RMAS

O conceito de arquitetura de software sempre é atrelado com o objetivo de fornecer uma estrutura que faça com que os sistemas desenvolvidos com seu apoio satisfaçam determinados atributos externos e internos de qualidade. O foco principal da arquitetura de referência que foi desenvolvida neste trabalho é facilitar atividades de manutenção. Para que fosse possível elaborar a arquitetura com esse objetivo, foi necessário identificar primeiramente os cenários de evolução/manutenção mais típicos em aplicações robóticas autoadaptativas. Com base nesses cenários é que a arquitetura foi então desenvolvida.

Dessa forma, neste capítulo apresentam-se os cenários de evolução estudados e considerados para o desenvolvimento da arquitetura. Apresentam-se também, para cada cenário de evolução identificado, as soluções de projeto adotadas e que compõem a arquitetura SARAMR, que é então, apresentada completamente no Capítulo 6.

5.1 Cenários de Evolução

Com base nos estudos realizados, os cenários de evolução mais típicos no domínio de robôs móveis autônomos podem ser classificados em quatro tipos: Evoluções Eletromecânicas, Evoluções de Ambiente, Evoluções do Controle do Robô e Evoluções de Desempenho e Confiabilidade. Essas quatro categorias de cenários de evolução surgiram após a utilização do modelo 5W1H como base para atividades de investigação com o objetivo de facilitar a especificação de requisitos de software no domínio de RMAs. As perguntas foram elaboradas seguindo o modelo 5W1H. Por exemplo:

1. **O que deve evoluir no sistema robótico?** Resposta: Adicionar, remover ou substituir componentes de hardware. Poder operar com mobilidade em diferentes configurações de ambientes fechados. Alterar, aperfeiçoar os comportamentos do robô, etc.

2. **Porque é necessário evoluir?** Resposta: Para adicionar novas funcionalidades. O robô deve se adaptar a diferentes tipos de ambiente. Novos comportamentos podem ser necessários às tarefas que o robô deverá realizar.

Outras perguntas foram elaboradas seguindo a mesma linha de investigação e sempre mantendo o foco nas características de evolução. Com o andamento do trabalho, foi observada a existência de uma relação entre as respostas obtidas, evidenciando as categorias de cenários de evolução. Assim, foi proposta uma matriz de relacionamentos com as perguntas organizadas nas linhas e as respostas relacionadas nas colunas. Essa atividade de investigação revelou os quatro cenários mostrados na Tabela 5.1.

Tabela 5.1: Cenários de evolução do sistema para RMAs

Evolução	Eletromecânica	Representação do Ambiente	Controle	Desempenho
O que deve evoluir?	Parte física do robô, adicionando novos componentes de hardware.	A representação do ambiente, adicionando ou removendo elementos ao ambiente.	Os comportamentos do robô.	O conhecimento de energia do robô.
Por que evoluir?	Para adicionar novas funcionalidades.	O robô deve se adaptar a novas características do ambiente.	Um comportamento pode precisar de modificações. Novos comportamentos podem ser necessários.	Para otimizar o desempenho e garantir o funcionamento do robô.
Quando evoluir?	Quando a configuração atual de hardware do robô não é adequada para determinada tarefa..	Quando o robô deve se movimentar por um ambiente diferente do que estava previamente configurado.	Quando for necessário modificar, adicionar ou remover novos comportamentos ao controle do robô.	Quando for necessário otimizar de alguma forma, melhorando o desempenho do robô na realização de suas tarefas.
Onde será a evolução?	Nos componentes de hardware e na implementação dos componentes de software.	Na implementação ou na manutenção dos componentes de software que representam o ambiente.	Na implementação dos componentes de software relacionados ao comportamento.	Na implementação dos componentes de hardware e de software.
Como será a evolução?	Adicionando novos componentes como atuadores e sensores e implementando suas funcionalidades.	Adicionando ou atualizando os componentes de software que representam o ambiente.	Alterando componentes de software ou adicionando novos componentes relacionados ao comportamento.	Otimizando comportamentos, introduzindo meios para otimização do uso da energia.

Os cenários de evolução apresentados neste capítulo foram organizados em quatro categorias, eletromecânica, controle do robô, representação do ambiente e desempenho. Nas

próximas seções, são mostradas as quatro categorias de evoluções mais recorrentes nesse domínio. Para cada categoria, comenta-se sobre a causa e o impacto da evolução. Causas são razões que exigem/disparam a necessidade de alguma alteração no projeto do sistema. O impacto consiste nas modificações que são necessárias nas classes/relacionamentos e estrutura do sistema. Além disso, para cada categoria mostra-se também a solução em termos de classes de projeto que foi adotada para compor a arquitetura final.

5.2 Evoluções Eletromecânicas do Robô

Apesar das diferenças na estrutura mecânica de diferentes robôs, muitas semelhanças podem ser identificadas na forma como as aplicações são implementadas, por exemplo, na aquisição de dados dos sensores ou na transmissão de comandos para os atuadores. Obrigatoriamente, essa tarefa de adquirir dados ou enviar comandos sempre precisará fazer referência a uma instância do objeto que representa esses dispositivos. Uma das razões que impedem uma aplicação de migrar facilmente para diferentes plataformas robóticas é que o software que controla o robô depende fortemente do tipo de hardware utilizado para realizar uma determinada tarefa, ou seja, a estrutura mecânica do robô influencia fortemente os requisitos da aplicação do software (BRUGALI, 2007). Geralmente, essa dependência ocorre em decorrência do uso de APIs (*Application Program Interfaces*) específicas. Por exemplo, o desenvolvimento de aplicações para robôs Lego devem usar obrigatoriamente a API Lejos, a qual dá acesso aos componentes de hardware específicos desse tipo de robô.

Com base nos estudos realizados, as principais evoluções eletromecânicas que podem ocorrer em um robô móvel para monitoramento de ambientes fechados são:

- Inclusão, remoção ou alteração de sensores e atuadores;
- Alteração do número de rodas, tipo e tamanho delas;

Com base nesse tipo de evolução, foram especificados cinco requisitos para a arquitetura de referência, que podem ser vistos na Tabela 5.2. A primeira coluna refere-se à identificação e classificação do requisito. REE significa Requisito de Evolução Eletromecânica.

Tabela 5.2: Requisitos arquiteturais de evolução eletromecânica.

Identificação	Descrição
REE-1	A AR deve permitir a inclusão e remoção de sensores/atuadores na estrutura física do robô impactando em apenas um local do código-fonte da aplicação.
REE-2	A AR deve permitir alterações no tipo e tamanho das rodas do robô impactando em apenas um local do código-fonte da aplicação.
REE-3	A AR deve permitir alterações de tamanho e peso do robô.
REE-4	A AR deve permitir a interpretação de sensores de forma individual ou coletiva.
REE-5	A AR deve facilitar a utilização de APIs de desenvolvimento.

Com relação ao requisito REE-1, foi observado durante a fase de desenvolvimento que as chamadas aos métodos para a aquisição dos dados dos sensores e aos métodos que enviam comando para os atuadores são constantemente requisitadas em diversas funcionalidades do robô. Com isso, objetos que representam os sensores e atuadores podem estar sendo instanciados de maneira inapropriada e espalhados em diversos componentes por toda a aplicação. Para se instanciar sensores ou atuadores, é necessário referenciar a porta lógica onde esses dispositivos estão conectados de acordo com a conexão física de hardware. Por exemplo, o sensor ultrassônico está conectado a porta P1 enquanto os motores A e B estão conectados as portas P3 e P4 respectivamente. Caso essas ligações físicas sejam modificadas por algum motivo na reconfiguração física do robô, será necessário rastrear as linhas de código desses objetos por todos os componentes da aplicação onde foram instanciados, e realizar a devida alteração. O impacto disso é que existe uma grande probabilidade de se informar incorretamente a porta onde o dispositivo está realmente conectado e como consequência a propagação de erros pela aplicação. Esse cenário aponta para a uma difícil manutenção futura no código da aplicação.

A arquitetura proposta deverá atender a esse requisito indicando ao arquiteto de software, como evitar os problemas encontrados nesse cenário. A solução estrutural deve ser uma referência para componentes de software que, ainda na fase de projeto, antecipe as condições para uma melhor evolução e manutenção do software.

Com relação ao requisito REE-2, para se obter melhor deslocamento, dependendo do tipo de terreno por onde o robô irá se locomover, poderá ser necessário alterar o tipo ou tamanho das rodas. Geralmente, esses parâmetros de configuração são informados no momento de instanciação dos objetos que representam os atuadores. Assim como no requisito anterior,caso os atuadores estejam sendo instanciados de forma inadequada, ou seja,

instâncias espalhadas pelos componentes da aplicação, será necessário rastrear as linhas de código desses objetos por todos os componentes onde foram instanciados, e realizar a devida alteração. O impacto disso é que existe uma grande probabilidade de que a manutenção do código seja uma tarefa difícil e demorada de se realizar.

Com relação ao requisito REE-3, a inclusão ou remoção de dispositivos eletromecânicos no sistema robótico, geralmente tem como efeito a alteração de seu tamanho e peso. Esse tipo de alteração não deverá causar nenhum impacto estrutural nos componentes da aplicação ou na forma como os componentes se relacionam. Alterações no código referentes a tamanho e peso deverão ser realizadas em um único local causando mínimo impacto na manutenção ou evolução da aplicação.

Com relação ao requisito REE-4, foi observado que os comandos enviados aos atuadores são definidos de acordo com os valores lidos dos sensores. Em determinadas situações pode ser necessário obter informações de diversos sensores de forma instantânea e coletiva, combinar esses valores para conseguir uma melhor interpretação das situação do robô no ambiente, e então definir o comando mais adequado à situação. Por exemplo o robô pode estar equipado com um conjunto de 12 sensores na parte da frente, 12 sensores na parte de traz e mais seis sensores divididos em igual número nas laterais, direita e esquerda.

Para que a aplicação possa obter a leitura desses sensores de forma coletiva, sem a necessidade de muitas linhas de código, é importante que os objetos que representam esses sensores estejam organizados em um componente de software que seja uma coleção lógica. Assim, para realizar a leitura de forma coletiva basta realizar uma iteração sobre a coleção de sensores, obtendo a identificação do sensor e o valor lido. Por outro lada, se for necessário realizar a leitura de um único sensor, basta referenciar o índice na coleção para acessar o objeto e acionar o método de leitura.

Com relação ao requisito REE-5, geralmente, os fabricantes de kits de robótica disponibilizam uma biblioteca de componentes de software para os dispositivos eletromecânicos. No caso deste trabalho, foi utilizado o kit Lego Mindstorms e a API LeJOS. Esses componentes, assim como os componentes de outras API, são modulares e específicos, representando cada componente eletromecânico de forma individual. No caso desse requisito, a arquitetura de referência deverá envolver, estruturar e organizar os relacionamentos entre os componentes da API.

Todos esses requisitos têm o objetivo de estabelecer uma arquitetura de referência que auxilie o arquiteto de software na produção de um bom projeto considerando o cenário de evolução eletromecânica para o domínio de robôs móveis de segurança para ambientes fechados. O projeto físico de um RMA usualmente tem início na definição do tipo de ambiente onde este robô irá atuar e na especificação do conjunto de tarefas que este robô deverá executar. Por exemplo, um robô de segurança e vigilância para ambientes fechados deve ter sensores e atuadores que permitam a ele se localizar e se deslocar no ambiente, detectar e desviar de obstáculos estáticos, e também deve possuir câmeras para a detecção de movimentos e de intrusos, onde possivelmente um sensor de calor ou uma câmera infra-vermelho e a capacidade de comunicação seriam requisitos necessários. A partir destas informações, tem início a etapa de configuração dos sensores e atuadores que serão utilizados no sistema robótico. O ambiente de atuação e as tarefas a serem executadas tem forte impacto na escolha dos dispositivos eletromecânicos e na forma física do robô (WOLF *et al*. 2009).

Para melhorar os aspectos de evolução eletromecânica do robô, adicionar e usar novos sensores e atuadores foi testada uma solução proposta por Brugali (2007), o padrão AnyMorphology apresentado na Figura 5.1.

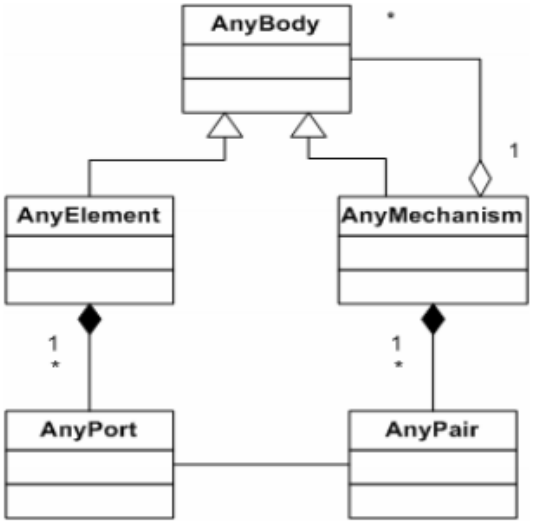

Figura 5.1: Diagrama UML do padrão AnyMorphology (BRUGALI, 2007).

O padrão *AnyMorphology* é uma solução proposta para abstrair os componentes de hardware que fazem parte da estrutura física do robô. Como pode ser visto na Figura 5.1, esse padrão é muito semelhante ao padrão *Composite* apresentado por Gamma *et al.* (1995) e foi adaptado por Brugali para o domínio da robótica.

O padrão *AnyMorphology* define cinco abstrações de software que mapeiam diretamente as entidades físicas que compõem mecanismos robóticos, *AnyElement, AnyPort, AnyPair, AnyMechanism* e *AnyBody*.

AnyBody é uma classe abstrata que representa o conjunto composto. *AnyElement* representa um componente como um elemento único que pode ser um sensor, atuador etc. *AnyMechanism* representa uma parte composta de mecanismos como um conjunto de sensores ou um conjunto de atuadores. *AnyPort* e *AnyPair* representam a junção de dois elementos da estrutura do robô. Com essas abstrações pode-se formar conjuntos compostos de vários elementos que são os dispositivos eletromecânicos, formando um determinado mecanismo, ou ainda formar um conjunto de mecanismos que compõem a estrutura completa do robô. A intenção desse padrão é que os dispositivos possam ser abstraídos de forma que adicionar ou remover elementos ou ainda acessar esses elementos seja uma funcionalidade básica da aplicação.

De acordo com Brugali (2007), o padrão *AnyMorphology* mantém a representação das interligações de elementos o mais simples possível. Isso torna o modelo resistente a mudanças nas especificações de requisitos da aplicação, uma vez que suporta a representação de novos mecanismos em diferentes topologias de forma integrada, como no caso de robôs reconfiguráveis. O modelo é altamente escalável graças à sua estrutura modular. Cada mecanismo é modelado como uma entidade separada, cujas propriedades são definidas dentro de um módulo individual. O modelo proposto é eficiente, pois apoia o acesso aos elementos que compõem estruturas mecânicas complexas, graças à organização hierárquica. Mesmo se o tamanho da composição for muito grande, algoritmos de busca e atualização podem explorar a hierarquia da composição para recuperar informações sobre cada elemento.

O padrão *AnyMorphology* foi testado e utilizado no contexto deste trabalho e se mostrou como uma boa solução para ser incorporada à arquitetura de referência aqui proposta. A Figura 5.2 ilustra o diagrama de classes para a arquitetura de referência da parte física do robô.

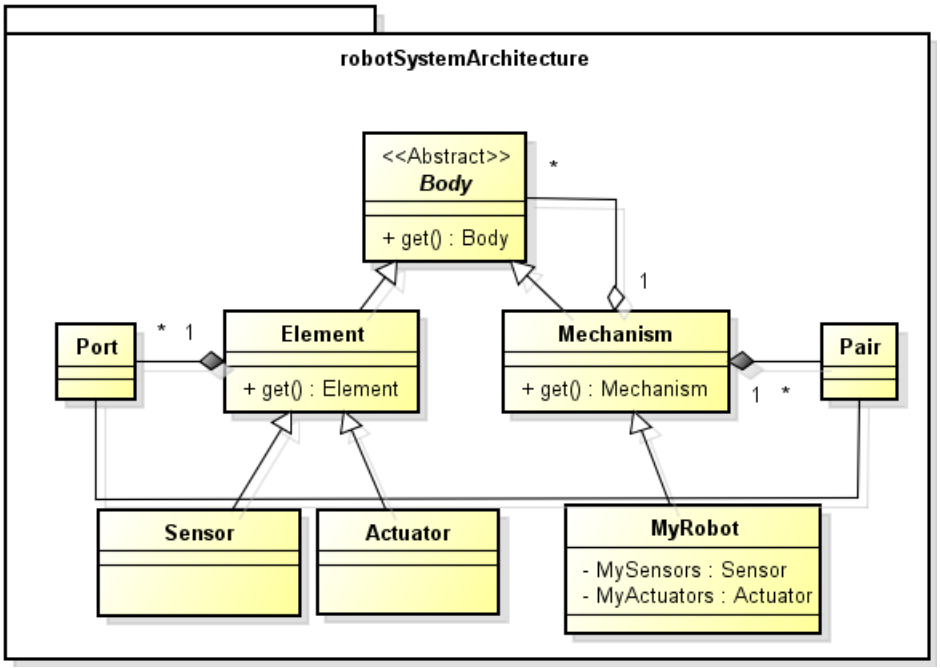

Figura 5.2: Diagrama de classes de projeto da parte física do robô.

Como pode ser visto no diagrama de classes da Figura 5.2, a solução proposta consiste em um modelo que possa representar cada componente da estrutura física do robô de forma individual (`Element`) ou, se necessário, de forma coletiva (`Mechanism`). Cada dispositivo, seja um sensor ou um atuador, pode ser modelado de forma individual, estendendo a classe `Element`, e posteriormente poderão ser agrupados em uma coleção de componentes de dois tipos: `MySensors;` que representa a coleção de todos os sensores, e `MyActuators;` que representa a coleção de atuadores. Essas duas classes estendem a classes `Mechanism`.

Essa solução proporciona à arquitetura atributos de flexibilidade e escalabilidade, atendendo aos requisitos do cenário de evolução eletromecânica do robô.

5.3 Evoluções da Representação do Ambiente

Localização e navegação são duas funcionalidades fundamentais para que um RMA possa realizar tarefas que envolvam mobilidade. O robô precisa "saber" qual a sua localização no ambiente e precisa de um plano que indique qual caminho deverá ser percorrido entre seu ponto de origem e um determinado destino (THRUN, 2003; BRÄUNL, 2008).

As alterações necessárias no sistema referentes a essa categoria de robôs ocorrem devido a natureza dinâmica da maioria dos ambientes físicos e a interação do robô com esse ambiente. Por exemplo, pode ser necessário incluir novos elementos no ambiente, como corredores, portas ou obstáculos. Por outro lado, pode ser necessária a exclusão de elementos que não estejam mais no ambiente, ou ainda representar um novo ambiente completamente diferente do ambiente anterior. Sempre que houver esse tipo alterações, o modelo espacial, com as distâncias métricas, deverá estar corretamente representado.

Com base nos estudos realizados, as alterações mais comuns que podem ocorrer em ambientes fechados são:

- inclusão ou remoção elementos: salas, corredores, portas, etc;
- representação de obstáculos de tamanhos variados;
- representação de novos ambientes;
- alterações no caminho a percorrer pelo ambiente.

Com base nesse tipo de evolução foram especificados alguns requisitos para a arquitetura de referência, que podem ser vistos na Tabela 5.3.

Tabela 5.3: Requisitos arquiteturais de evolução do ambiente.

Identificação	Descrição
REA-1	A AR deve permitir a inclusão/alteração de componentes de software que representam elementos do ambiente físico por meio da criação de objetos. A abordagem utilizada deve ser métrica em duas dimensões para largura e comprimento.
REA-2	Quando houver alterações nas dimensões de objetos físicos, a AR deve permitir que isso seja feito apenas alterando parâmetros nas referências métricas de comprimento e largura.
REA-3	A AR deve permitir a inclusão e a remoção de obstáculos de tamanhos variados.
REA-4	A AR deve facilitar a manutenção referente a alterações no ambiente de atuação do robô.
REA-5	A AR deve permitir a atualização do modelo do ambiente em tempo de execução.

A primeira coluna refere-se à identificação e classificação do requisito. REA significa Requisito de Evolução do Ambiente.

Com relação ao requisito REA-1, a arquitetura de referência deve fornecer uma estrutura de componentes para representar um ambiente fechado de forma métrica. A localização e o deslocamento do robô pelo ambiente deve utilizar coordenadas que proporcionam medidas de comprimento e largura. A arquitetura deverá prover componentes

específicos para uma clara separação dos interesses referentes ao mapeamento de ambientes fechados.

Com relação ao requisito REA-2, durante a fase de desenvolvimento foi observado que o sistema cartesiano com duas dimensões, coordenadas X e Y, facilita a definição de pontos navegáveis ou não no ambiente em que o robô está inserido. Os obstáculos fixos do ambiente podem ser facilmente representados utilizando retas que preenchem a grade de ocupação, uma vez demarcados os obstáculos, a área restante é o espaço de locomoção do robô. Uma outra situação observada é que a locomoção por meio de coordenadas métricas facilita as tarefas de planejamento do caminho.

Com relação ao requisito REA-3, durante a fase de desenvolvimento, utilizando a API LeJOS foi observado que apenas uma abstração que representa uma reta é utilizada para modelar qualquer tipo de elemento presente em um ambiente fechado. Portas, corredores, salas e todos os outros elementos são modelados utilizando uma composição de retas que unidas formam figuras geométricas como quadrados, triângulos e polígonos, ou seja qualquer tipo de elemento presente em um ambiente fechado. Assim, a arquitetura de referência deve simplificar a utilização mínima de componentes com bons níveis de abstração e capacidade de modelar diferentes tipos de elementos geométricos.

Com relação ao requisito REA-4, devido à grande diversidade de ambientes fechados existentes no domínio de segurança, é desejável que a manutenção necessária no código referente a alterações na configuração do ambiente seja uma tarefa muito simples, exigindo mínima alteração na codificação de forma que a manutenção seja rápida e fácil.

Com relação ao requisito REA-5, foi observado durante a fase de desenvolvimento a existência de obstáculos dinâmicos no ambiente e a necessidade de representação desses obstáculos em tempo de execução. Para que o robô seja mais eficiente no processo de locomoção, é necessário que exista um mecanismo de atualização do modelo do ambiente em tempo real. Os componentes da arquitetura de referência para o mapeamento devem estar estruturados e relacionados de forma que uma atualização dinâmica possa ser executada em tempo real aumentando o desempenho e a eficiência do robô nas tarefas de locomoção.

Para atender aos requisitos de evolução do ambiente especificados, uma pesquisa foi conduzida com o objetivo de conhecer quais as técnicas de mapeamento mais utilizadas na Engenharia de Software para resolver os problemas de mapeamento de ambientes fechados

encontrados no domínio da robótica móvel. As abordagens de mapeamento mais encontradas na literatura são o mapeamento métrico e o mapeamento topológico (SIEGWART, 2011). Essas duas abordagens fornecem um modelo espacial para representação do ambiente.

A abordagem topológica concentra-se em características do ambiente que são mais relevantes para localização do robô. Basicamente, uma representação topológica é um grafo onde os vértices são os locais do ambiente que o robô pode alcançar e as arestas são os caminhos que o robô pode seguir para chegar de um lugar a outro. Um exemplo de mapa topológico usando grafos é ilustrado na Figura 5.3.

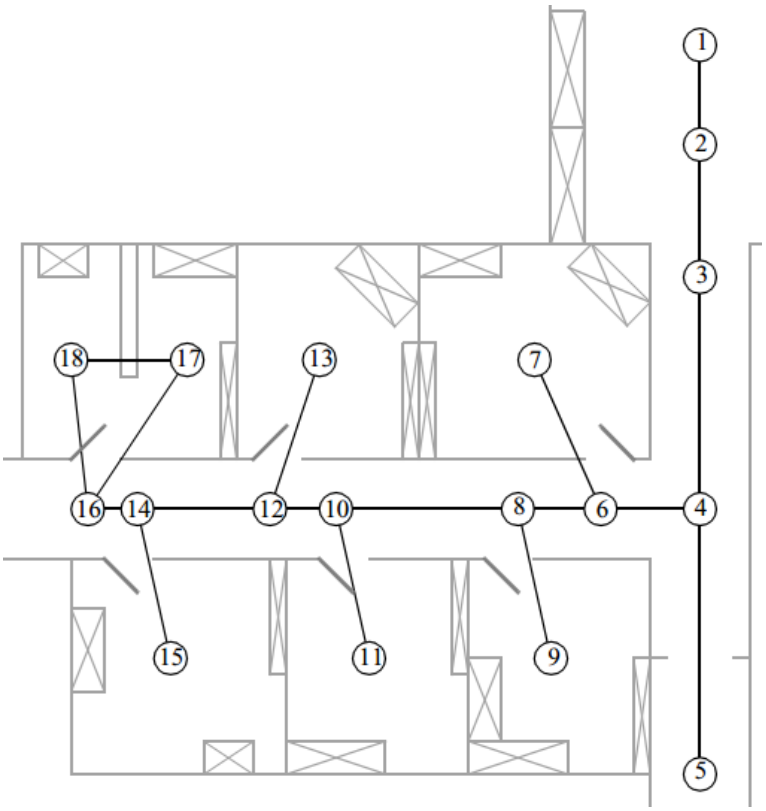

Figura 5.3: Grafo representando um mapa topológico, (SIEGWART *et al*, 2011).

Como é visto na Figura 5.3 o mapa topológico não leva em consideração a geometria de todas as partes do ambiente, pois não é um mapa métrico, e também não descreve com detalhes a presença de obstáculos no ambiente. As arestas são um sistema de referência com uma reta que faz a ligação entre os locais no ambiente. Um sistema de referência como tal é chamado de sistema em uma dimensão. O mapa métrico, no entanto, é um sistema de coordenadas composto de duas retas formando um ângulo reto uma com a outra. Esse abordagem permite localizar um ponto acima e abaixo, além de a direita ou a esquerda, formando um sistema de referência 2D (duas dimensões) (SIEGWART, 2011).

O sistema de coordenadas consiste em um plano cartesiano que estabelece um sistema de referência métrico no qual cada ponto sobre a reta tem uma distância até o próximo ponto, seja abaixo, acima, direita ou esquerda. O mapeamento do ambiente do robô por meio de mapas métricos é conhecido como grade de ocupação (SIEGWART, 2011).

O processo de mapeamento é realizado por meio de informações métricas coletadas do próprio ambiente onde o robô se encontra e a aplicação de algoritmos computacionais para a criação de uma grade de ocupação que representa a "visão" que o robô tem do ambiente. Com o processo de mapeamento concluído a grade de ocupação pode ser utilizada para a navegação, localização, planejamento de trajetória, desvio de obstáculos e outras aplicações (THRUN, 2005). Na Figura 5.4 é mostrado um exemplo de grades de ocupação.

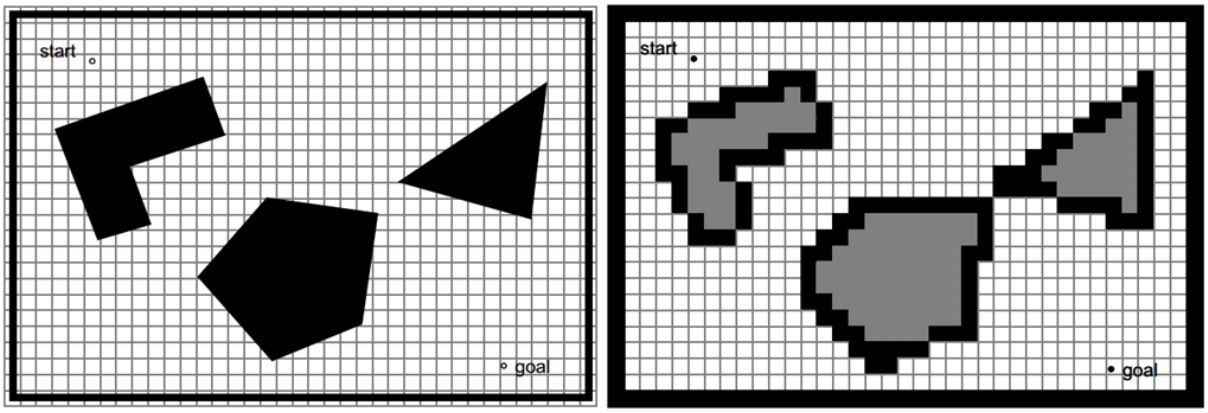

Figura 5.4: Exemplos de grade de ocupação (SIEGWART, 2011).

Geralmente, um RMA consegue estimar o seu deslocamento pela grade de ocupação que representa o ambiente utilizando um sistema de odometria. As informações odométricas são obtidas dos atuadores que estão acoplados às rodas do robô, e à partir do movimento das rodas é possível calcular o deslocamento (WOLF *et al*. 2009). O sistema de odometria é responsável pela correta "navegação" pelo ambiente usando como referência as métricas presentes na grada de ocupação.

Neste trabalho de pesquisa, no qual o foco principal é o ambiente fechado para os cenários de localização e locomoção de RMAs, a abordagem de grade de ocupação mostrou-se como uma boa forma de representar ambientes. Considerando que o ambiente de atuação do robô é um ambiente fechado, e esse tipo de ambiente, muitas vezes, é composto por um conjunto de elementos como salas, corredores e portas, ou seja, os obstáculos fixos são predominantes, é desejável que a aplicação tenha uma grade de ocupação como recurso cognitivo que mapeie o local de trabalho do robô.

Esses cenários representam situações onde a evolução do ambiente de atuação do robô tem impacto direto no software da aplicação robótica e mostram que é necessária uma boa solução arquitetural, abstrata o suficiente para acomodar essas mudanças. Nesse caso, para melhorar os aspectos de evolução, e após vários estudos e testes realizados, e inclusive observando os componentes da API LeJOS, foi adotada a solução para a representação do ambiente, ilustrada na Figura 5.5.

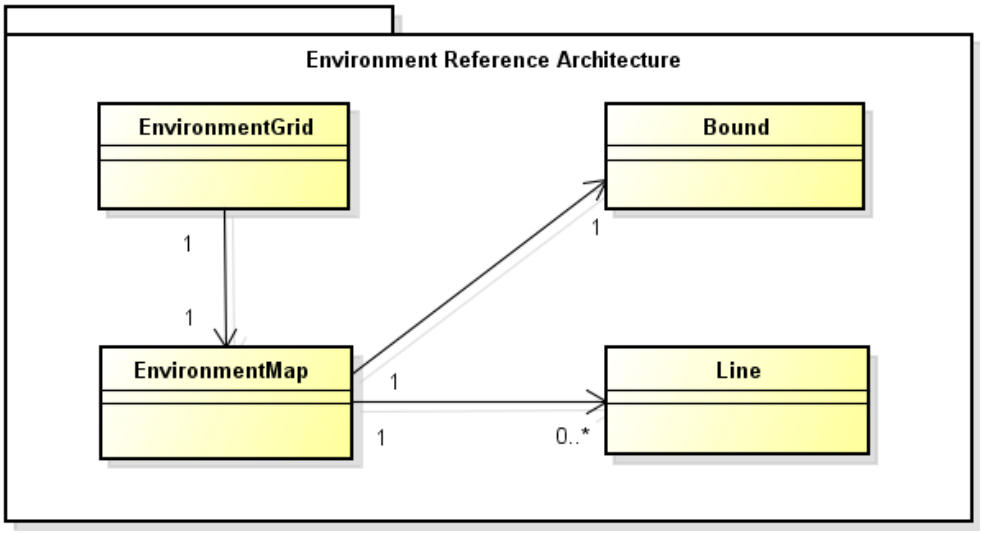

Figura 5.5: Diagrama de classes para o projeto de representação do ambiente.

A classe `EnvironmentGrid` é usada para modelar uma rede de nós que representa a grade de ocupação. A geometria ou espaçamento entre os nós da grade pode ser previamente especificada. Essa rede pode ser modelada no início da execução do programa, definindo-se a distância métrica padrão entre os nós, e pode ser utilizado durante toda a navegação subsequente.

A classe `EnvironmentMap` é composta de figuras geométricas que representam o mapa de um prédio ou uma sala ou outro tipo de ambiente fechado. As figuras geométricas são constituídas por um retângulo que demarca as fronteiras do ambiente fechado e vários segmentos de linha que são unidas para representar elementos do ambiente.

A classe `Bound` representa a forma geométrica de um retângulo que delimita as fronteiras do ambiente.

A classe `Line` representa uma linha num segmento entre dois pontos. Essa classe é usada para representar os elementos do ambiente.

Essas classes compõem o módulo que fornece as funcionalidades de representação do ambiente e "conhecimento" de obstáculos. Essas funcionalidades são fundamentais para a realização das tarefas mais básicas no domínio de robôs móveis em ambientes fechados. Dessa forma a arquitetura de referência deverá permitir a evolução da aplicação nesse sentido, e as alterações mais comuns identificadas nesse cenário devem causar mínimo impacto nos esforços de desenvolvimento e manutenção do software da aplicação robótica.

5.4 Evoluções do Controle do Robô

Para executar uma tarefa, geralmente um robô deve agregar diferentes objetivos. Andar em linha reta e em velocidade constante é uma tarefa simples, pois utiliza poucos objetivos agregados. Por exemplo, a tarefa de partir de um ponto para poder chegar a um local determinado no ambiente, utilizar um mapa para estabelecer o melhor caminho até o objetivo, desviar de obstáculos, são tarefas mais complexas, pois utilizam vários objetivos agregados. Essas preocupações estão ligadas ao estudo de técnicas utilizadas para resolver problemas de controle do robô.

Para resolver os problemas de controle de robôs móveis, a abordagem mais empregada costuma ser o paradigma de controle deliberativo, em que o robô possui uma representação do ambiente como recurso cognitivo. Outra abordagem, apresentada por Brooks (1986), é o controle reativo, baseado em comportamentos, em que o robô usa o próprio ambiente como modelo e reage diretamente aos estímulos dos sensores. Brooks também apresentou o conceito *Subsumption* que é o controle baseado em comportamentos. Arkin (1987) apresentou o conceito de arquitetura mista reativa/deliberativa.

Os comportamentos são módulos de controle com funções bem definidas que trabalham em conjunto para alcançar um objetivo desejado. Os comportamentos recebem dados dos sensores e enviam comandos aos atuadores. Quando agrupados e controlados possibilitam ao robô a execução das mais diversas e complexas tarefas.

Com base nos estudos realizados, as principais alteração que podem ocorrer nas abordagens de controle do robô são:

- alteração na abordagem do comportamento para deliberativo, reativo ou hibrido.

- inclusão e remoção de comportamentos
- alteração de comportamentos existentes

Com base nesse tipo de evolução foram especificados alguns requisitos para a arquitetura de referência, que podem ser vistos na Tabela 5.4.

Tabela 5.4: Requisitos arquiteturais de evolução do controle do robô.

Identificação	Descrição
REC-1	A AR deve fornecer um módulo com componentes para o controle baseado em comportamentos.
REC-2	A AR deve permitir a implementação de comportamentos nas formas reativa, deliberativa ou hibrida.
REC-3	A AR deve permitir a inclusão de novos comportamentos apenas por meio da criação de novas classes e facilitar a inclusão, alteração ou remoção de comportamentos.
REC-4	A AR deve fornecer componentes para a coordenação dos comportamentos.

A primeira coluna refere-se a identificação e classificação do requisito. REC significa Requisito de Evolução do Controle. Com relação ao requisito REC-1, ao invés de escrever grandes blocos de códigos com muitas funcionalidades implementadas, a arquitetura deve indicar a decomposição de funcionalidades em componentes mais coesos, dividindo-as em várias instancias de comportamentos. O ideal é que seja feita uma decomposição coerente distribuindo os objetivos de forma inteligente em vários comportamentos que, uma vez gerenciados, vão trabalhar de forma cooperativa.

Com relação ao requisito REC-2, foi observado que a implementação de um dado comportamento pode precisar ou não utilizar uma representação do ambiente. Isso vai depender da complexidade relacionada ao número de objetivos que o comportamento deve realizar. Pode ser que um determinado comportamento não precise utilizar mapas e nem calcular caminhos, nesse caso, o comportamento pode ser puramente reativo. Por outro lado, pode haver comportamentos que precisem das funcionalidades de localização e navegação, assim, o componente de implementação de comportamentos deve ser abstrato o suficiente para permitir as três formas de implementação.

Com relação ao requisito REC-3, é esperado o surgimento de novas funcionalidades ou a melhoria de funcionalidades existentes. Nesse caso, a arquitetura deve ser bem flexível e o esforço necessário para incluir, alterar ou excluir novos comportamentos deve ser mínimo,

ou seja a manutenção do código referente a esse requisito deve ser uma tarefa de fácil realização.

Com relação ao requisito REC-4, na existência de vários comportamentos funcionando de forma cooperativa, ficou claro que a arquitetura deve indicar a necessidade de um componente cuja função seja coordenar a concorrência entre os comportamentos.

Para atender aos requisitos de evolução do controle de robôs móveis, uma pesquisa foi conduzida com o objetivo de conhecer quais as técnicas utilizadas na engenharia de software para essa categoria de problema. Duas abordagens clássicas que definem a forma de atuação de robôs móveis são os paradigmas baseados em comportamentos da arquitetura reativa e da arquitetura deliberativa. A união dos dois paradigmas formam a arquitetura de comportamentos mista, ou seja, reativa e deliberativa. O modelo de arquitetura baseado em comportamentos chamada de *subsumption architectute* foi apresentado por Brooks (1986). Essa abordagem foi considerada como um sistema de controle para RMAs que precisam realizar tarefas em tempo real. Um comportamento pode ser um conjunto de ações realizadas pelo robô. A partir da percepção do ambiente, por meio das informações dos sensores, uma ação gera um estimulo em forma de comandos sobre os atuadores do robô.

Antes da proposta de Brooks, tradicionalmente, o desenvolvimento de sistemas de controle para robôs móveis usa a decomposição de comportamentos em módulos com funcionalidades para tarefas simples como ilustrado na Figura 5.6. Cada uma dessas unidades está fortemente ligada e dependente da unidades vizinha, de tal forma que o sistema de comportamentos deve ser projetado de forma completa, pois uma unidade isolada não é capaz de executar nenhuma ação.

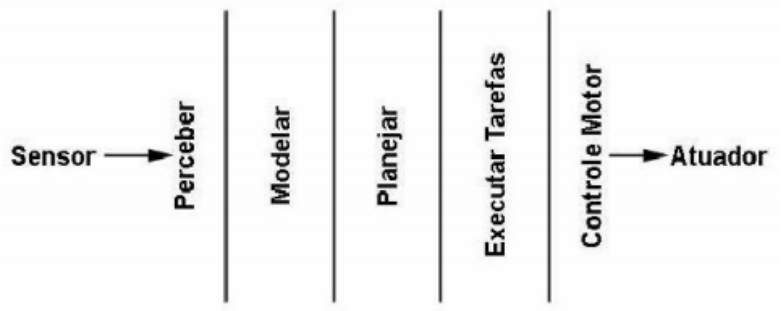

Figura 5.6: Divisão das funcionalidades em módulos (BROOKS, 1986).

O sistema de controle baseado nessa arquitetura executam suas tarefas nas seguintes etapas:

- A entrada de dados vinda dos sensores é usada como informações para perceber situações do ambiente;
- Os dados coletados são usados para atualizar a representação interna do ambiente;
- Em seguida, baseado nessas informações, um plano de ação é elaborado;
- Uma série de comandos é enviada aos atuadores o que resulta no ciclo de ações que o robô realiza.

Obter as informações relevantes do ambiente e construir um modelo o mais completo possível é parte fundamental dessa arquitetura.

Com a proposta de Brooks (1986), ao invés da divisão das funcionalidades em módulos dependentes uns dos outros, o sistema de controle deve dividir as tarefas em comportamentos organizados em camadas. A organização do controle é realizado em um conjunto de camadas que representam uma tarefa ou comportamento complexo. Um exemplo desses comportamentos organizados em camadas é ilustrado na Figura 5.7.

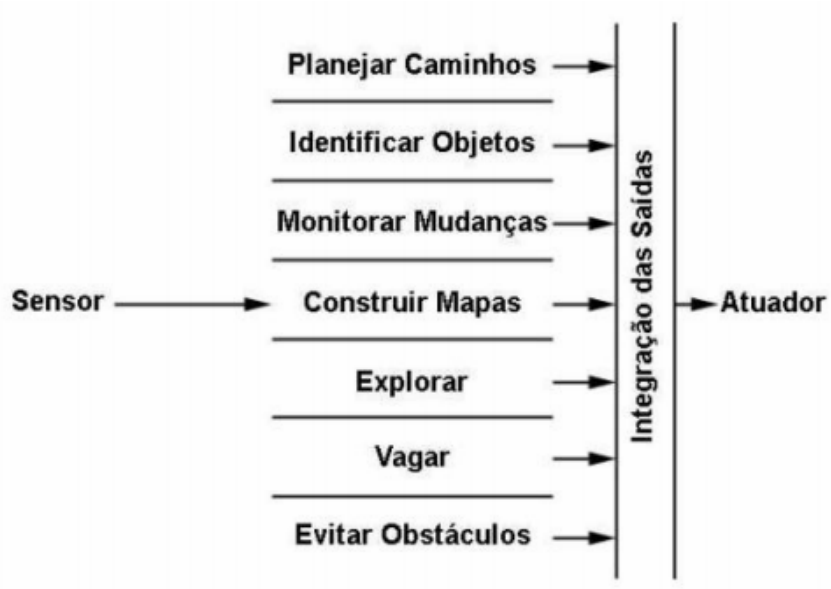

Figura 5.7: Arquitetura baseada em comportamentos, (BROOKS, 1986).

A arquitetura estruturada em camadas permite a adição ou remoção de comportamentos como ilustrado na Figura 5.8.

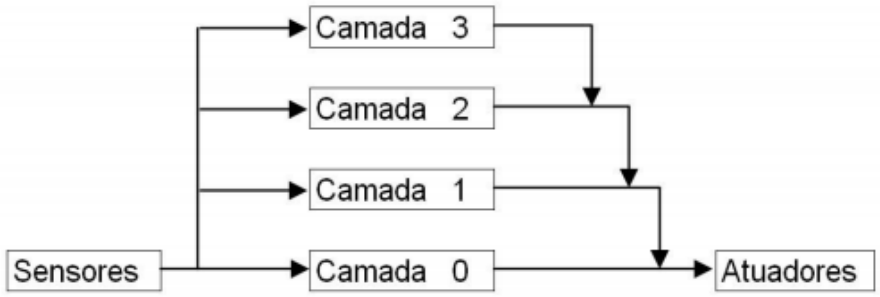

Figura 5.8: Arquitetura em camadas (BROOKS, 1986).

Em vez de projetar grandes blocos de código, cada funcionalidade do robô é encapsulada em um módulo pequeno chamado de comportamento. Todos os comportamentos são executados em paralelo, sem necessidade de sincronização. Um dos objetivos dessa arquitetura é simplificar a capacidade de estender novas funcionalidades na aplicação, por exemplo, poder adicionar um novo sensor ou uma nova característica comportamental no sistema robótico. Todos os comportamentos podem acessar todos os sensores e atuadores produzindo um ponto único para aquisição de informações e realização de comandos para os atuadores do RMA (BRÄUNL, 2006).

A API LeJOS possui um pacote chamado *Subsumption* com um par de componentes, a interface *Behavior* e a classes *Arbitrator*. A finalidade desses componentes de fornecer suporte ao desenvolvimento do controle baseado em comportamentos. Como essa API é muito utilizada no meio acadêmico para o desenvolvimento de software para robôs móveis, foi possível encontrar vários exemplos de código utilizando essa abordagem. Após uma série de estudos realizados, foi adotada a seguinte proposta de arquitetura de referência para os componentes de controle do robô, ilustrada na Figura 5.9.

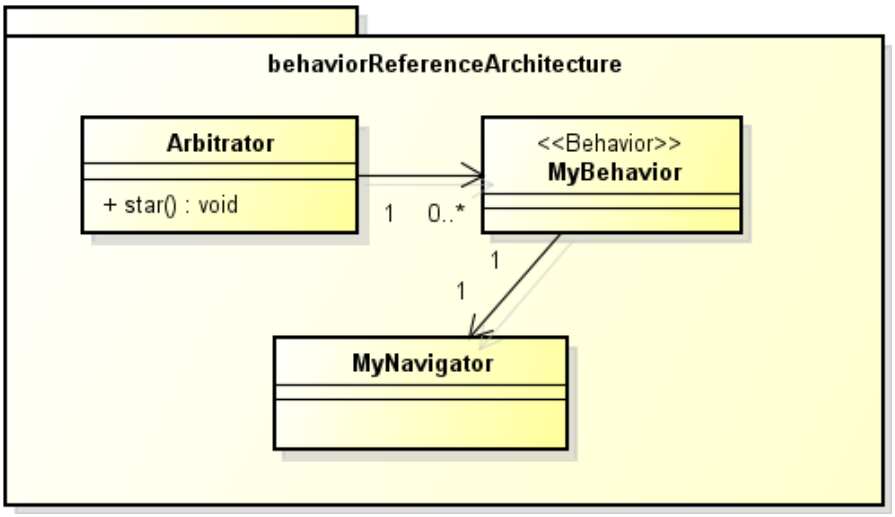

Figura 5.9: Diagrama de classes para o projeto de controle do robô.

A classe `MyBehavior` implementa a interface `Behavior` da API LeJOS. Essa interface representa um objeto que contém um determinado comportamento pertencente ao robô. Para concretizar a interface é necessário implementar três métodos:

- **`takeControl()`**: As circunstâncias que fazem o comportamento tomar o controle do robô;
- **`action()`**: A ação a ser executada quando este comportamento assume o controle;
- **`supress()`**:Uma maneira rápida de encerrar a ação que está sendo executada quando o Arbtrator escolhe um comportamento de maior prioridade para assumir o controle.

A classe `Arbtrator` recebe uma coleção de comportamentos e sua função é coordenar a concorrência. Um objeto da classe `Arbtrator` controla qual objeto da classe `MyBehavior` tornará-se ativa no processo de controle de comportamentos. Para realizar essa funçã,o a classe tem três grandes responsabilidades:

- Determinar o comportamento de maior prioridade;
- Suprimir o comportamento ativo se a sua prioridade é inferior a prioridade mais alta;
- Executar uma chamada ao método `action()` do comportamento de maior prioridade.

Criar um sistema de controle por meio desses componentes significa implementar um ou mais objetos da classe `MyBehavior` que contém comportamentos, definir a prioridade de cada comportamento, adicioná-los em uma coleção que pertence a um objeto da classe `Arbtrator` e executar uma chamada ao método start(). A partir disso o objeto da classe `Arbtrator` inicia a coordenação gerenciando o sistema de controle.

A classe `MyNavigator` recebe os comandos de navegação provenientes das classes de comportamentos que dão mobilidade ao robô. Essa classe faz a ligação entre o módulo de controle de comportamentos e o módulo de representação do ambiente.

5.5 Evoluções de Confiabilidade e Desempenho

Robôs autônomos fazem parte de uma categoria especial de robôs projetados para realizar tarefas sem a intervenção de seres humanos. Alguns robôs são projetados para realizar tarefas em ambientes completamente inóspitos à vida humana como o subsolo terrestre, as profundezas de oceanos ou a exploração espacial. Nessas situações podem ocorrer problemas inesperados como falhas ou a indisponibilidade do sistema. Isso pode prejudicar a realização das operações, pode causar riscos para seres humanos ou mesmo ao sistema robótico utilizado.

Nesse contexto, confiabilidade e desempenho são dois atributos de qualidade muito desejados para esse tipo de robô. Vários estudos (MACCAL, 1977; PRESSMAN, 1997; RAKITIN, 1997; EELES, 2005) têm apontado atributos, propriedades e características de qualidade. Confiabilidade é um atributo de qualidade que diz respeito a três propriedades: disponibilidade, precisão e recuperação. Disponibilidade é a capacidade de um sistema estar sempre disponível para utilização. Precisão é a capacidade de um sistema de ser exato, confiável e rigoroso. Recuperação é a capacidade de um sistema de se recuperar de falhas. Desempenho é um atributo de qualidade que diz respeito a características como: rendimento, tempo de resposta, tempo de recuperação, tempo de inicialização, produtividade e tempo de desligamento.

Esse cenário aponta para a necessidade de sistemas com funcionalidades para monitorar e prover estratégias de autoadaptação a situações imprevistas no ambiente. Autoadaptação é a capacidade do sistema de ajustar o seu comportamento em função do ambiente decidindo de forma autônoma (BRUN *et al*. 2009). Sistemas autoadaptativos têm sido estudados nas diferentes áreas de pesquisa da Engenharia de Software, incluindo a Engenharia de Requisitos (BROWN, 2006), Arquitetura de Software (GARLAN, 2003; RICHTER, 2006), *midware* (LIU, 2006), e desenvolvimento baseado em componentes (PEPER, 2008). De acordo com Brun *et al*. (2009), loops de controle fornecem um mecanismo genérico para prover autoadaptação de um sistema.

Com base nos estudos realizados, os cenários que podem ocorrer na evolução de desempenho são:

- inclusão de um sistema para recuperação de falhas;

- inclusão de um sistema para gerenciamento de consumo de energia.

Com base nesse tipo de evolução, foram especificados cinco requisitos de autoadaptação para a arquitetura de referência, que podem ser vistos na Tabela 5.5. A primeira coluna refere-se à identificação e classificação do requisito. RED significa Requisito de Evolução de Desempenho.

Tabela 5.5: Requisitos arquiteturais de evolução do desempenho.

Identificação	Descrição
RED-1	A AR deve fornecer um módulo para autoadaptação em tempo de execução.
RED-2	A AR deve indicar a necessidade de implementação políticas de adaptação.
RED-3	A AR deve indicar a separação dos interesses de autoadaptação.
RED-4	A AR deve prover melhor desempenho, utilizando o processamento no módulo de adaptação somente quando for necessário.
RED-5	A AR deve permitir o acoplamento de loops de controle de mais alto nível sobre outros loops.

Com relação ao requisito RED-1, a funcionalidade de autoadaptação deve ser executada em tempo de execução.

Com relação ao requisito RED-2, as funcionalidades de autoadaptação devem seguir regras predefinidas que façam parte de políticas que coordenem a adaptação. A arquitetura de referência deve indicar a necessidade de se estabelecer essas políticas.

Com relação ao requisito RED-3, uma aplicação robótica tem funcionalidades implementadas para diferentes tipos de comportamentos. É importante que a arquitetura de referência indique uma clara separação dos interesses, mantendo as funcionalidades de autoadaptação separadas dos comportamentos de controle do robô, de forma que facilite a manutenção e evolução referentes ao módulo de autoadaptação. Garlan *et al*., (2003) e Cheng *et al*. (2005) defendem essa abordagem de separar as preocupações de funcionalidades do sistema das preocupações da autoadaptação.

O requisito RED-4 é uma medida exigida como atributo de qualidade que visa melhor desempenho. É importante ter um sistema de adaptação que somente seja utilizado quando necessário. Assim, o sistema autonômico só utiliza os recursos de processamento da máquina quando for requerida a sua atuação.

Com relação ao requisito RED-5, módulo de loop de controle deve ser estabelecido de forma que possibilite controlar a aplicação do robô ou mesmo outro loop de controle, ou seja, possibilitar que um loop possa ser acoplado ao sistema de controle do robô ou possa ser acoplado a outro loop de controle.

Para atender aos requisitos de evolução de desempenho para robôs móveis, uma pesquisa foi conduzida com o objetivo de conhecer quais as técnicas mais utilizadas na engenharia de software para essa categoria de problema. Garlan *et al*. (2003) desenvolveram a arquitetura *Rainbow* que utiliza *feedback control loops* como um sistema automático para corrigir falhas. Uma pesquisa sobre autoadaptação realizada por Dobson *et al*., (2006) aponta que a sistemática para autoadaptação é separada em quatro atividades: coletar, analisar, decidir e agir. Essas atividades foram abordadas na seção anterior de Teoria do Controle e Sistemas Autoadaptativos.

Uma outra solução arquitetural estudada foi a apresentada por Kephart and Chess (2003) e popularizada pela IBM *Corporation* com o nome de "Modelo para Computação Autonômica", e que, de acordo com Brun et al. (2009), é a primeira arquitetura para sistemas autoadaptativos que expõe explicitamente os componentes de um loop de controle (f*eedback control loop*). Na Figura 5.10, é mostrado o diagrama de blocos da arquitetura detalhando os principais componentes:

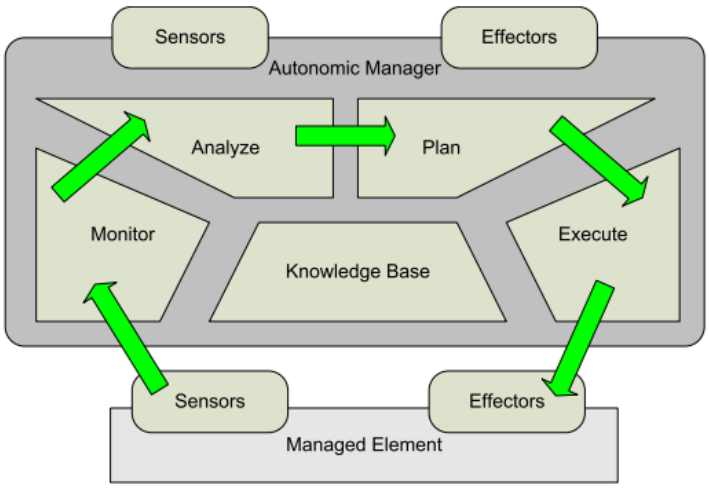

Figura 5.10: Gerenciador autonômico (KEPHART and CHESS, 2003).

Autonomic Manager é o sistema de *feedback control loop*, os elementos que compõem o núcleo dessa arquitetura são: *Monitor, Analyze, Plan, Execute* e *Knowledge Base*. Os seus nomes juntos formam o acrônimo MAPE-K. *Managed Element* é o sistema que está sendo

monitorado e gerenciado, *Sensors* e *Effectors* são as interfaces de interligação entre os dois sistemas.

- **Monitor**: O *monitor* detecta o processo gerenciado e seu contexto, filtra os dados acumulados dos sensores e armazena eventos relevantes na base de conhecimento (*Knowledge Base*) para referência futura (BRUN *et al*., 2009).

- ***Analyzer***: O *Analyzer* compara dados de eventos contra dados padrões na base de conhecimento para diagnosticar sintomas e armazena os sintomas para na base de conhecimanto (*Knowledge Base*) para referência futura (BRUN *et al*., 2009).

- ***Planner and Executor***: O *planner* interpreta os sintomas e elabora um plano para de ação para mudar a situação no processo gerenciado. O executor aplica o plano de mudança por meio dos atuadores (BRUN *et al*., 2009).

Considerando as soluções apresentadas e alguns estudos realizados com loops de controle, foi adotada a estrutura de classes ilustradas na Figura 5.11.

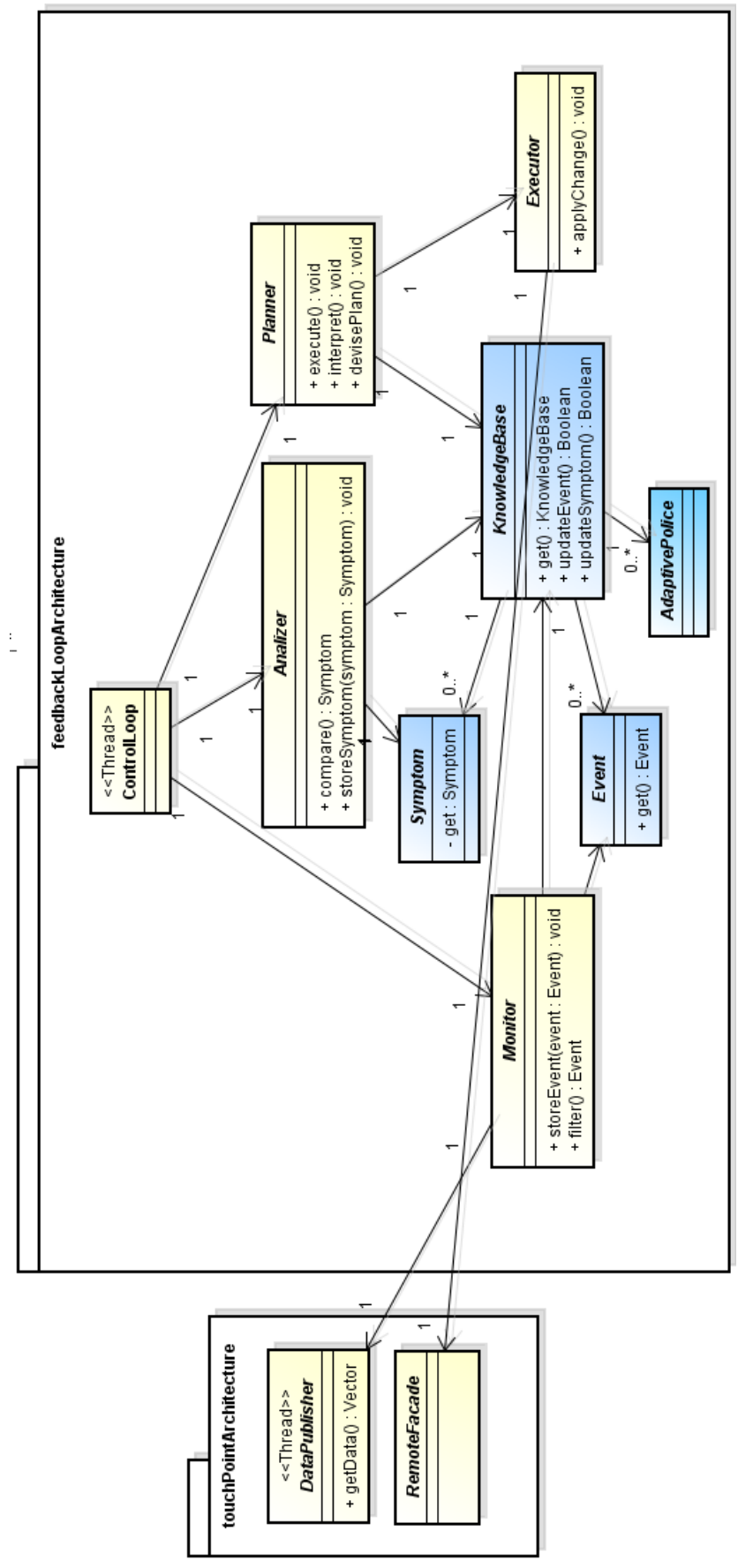

Figura 5.11: Estrutura de classes para um sistema de feedback control loop.

É importante observar que a solução arquitetural para esse cenário é composta de dois módulos: `Touch Point` e `Feedback Control Loop`.

O módulo Touch Point é constituído dos componentes `DataPublisher` e `RemoteFacade` e foi projetado para fazer a integração entre um sistema autonômico e um sistema gerenciado. A funcionalidade de integração entre sistemas será melhor apresentada no próximo capítulo.

O módulo de `Feedback Control Loop` constitui o sistema de loop de controle e possui as quatro funcionalidades básicas: coleta, análise, planejamento e ação. Os componentes que compõem esse módulo são: `Monitor`, `Analizer`, `Planner`, `Executor`, `KnowledgeBase`, `Event`, `Symptom`, `AdaptivePolice` e `ControlLoop`. Os relacionamentos entre esses componentes, bem como o fluxo de dados entre eles serão detalhados no próximo capítulo que apresenta a arquitetura de referência como um todo.

Capítulo 6

ARQUITETURA DE REFERÊNCIA SARA^{MR}

6.1 Considerações Iniciais

Neste capítulo, é apresentada SARA^{MR}, uma arquitetura de referência para robôs móveis autoadaptativos cujo objetivo é facilitar manutenções evolutivas no software que controla o robô. SARA^{MR} é um acrônimo para as palavras *Self-Adaptation*, *Reference Architecture* e *Mobile Robots*. No capítulo anterior, as principais soluções que compõem a arquitetura foram apresentadas, porém, em partes separadas de acordo com o cenário que foi investigado. Neste capítulo, as partes são integradas, mostrando a arquitetura de forma completa.

No decorrer deste capítulo, inicialmente a arquitetura é mostrada em uma visão mais abstrata ilustrada por um diagrama de blocos. Nessa perspectiva, pretende-se facilitar o entendimento da arquitetura de referência destacando os principais módulos bem como os relacionamentos e o fluxo de dados entre eles. Em outro ponto de vista, a arquitetura será mostrada por meio de um diagrama de classes ilustrando o relacionamento entre as classes que compõem módulos. A partir dessa perspectiva, maiores detalhes são revelados ilustrando as funcionalidades básicas nas classes abstratas e concretas. Depois que a arquitetura for apresentada, as próximas subseções mostram como utilizá-la, ou seja, como especializar as classes abstratas e realizar as interfaces para produzir software para robôs móveis autoadaptativos. Adicionalmente, como forma de avaliação da arquitetura, as próximas seções também mostram como foram atendidos os requisitos identificados nos cenários de evolução.

6.2 Detalhamento da Arquitetura de Referência

Na Figura 6.1 é mostrado o diagrama de blocos com os três principais módulos da arquitetura de referência SARA^{MR}, *Mobile Robot*, *FeedBack Control Loop* e *Touch Point*.

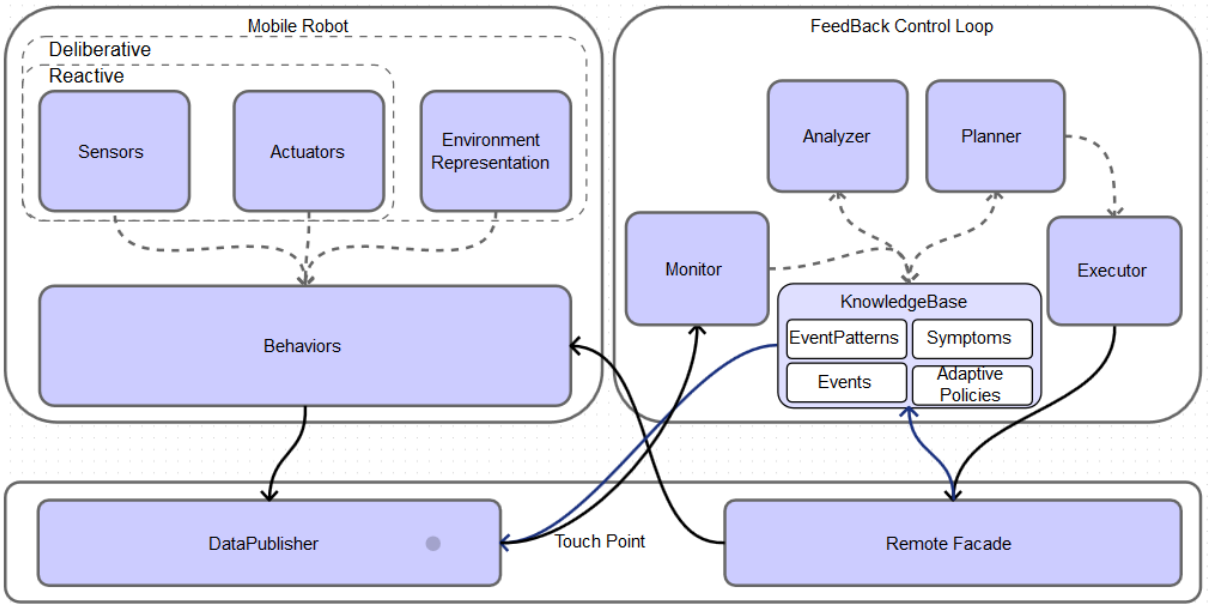

Figura 6.1: Diagrama de blocos da arquitetura SARA^{MR}

A visão de diagrama de blocos é muito utilizada nas representações de arquiteturas de software, pois representa uma visão bem mais abstrata que sintetiza a perspectiva e direciona o foco do observador num paradigma de alto nível. Com isso, os detalhes são abstraídos para permitir a apresentação dos módulos e seus elementos, bem como o relacionamento e o fluxo de dados entre eles. A estrutura interna dos módulos é formada por elementos que podem ser constituídos de uma ou mais classes. As linhas interligando os módulos indicam o relacionamento e as setas indicam o sentido do fluxo de dados. As linhas pontilhadas indicam o relacionamento interno entre os elementos que compõem o módulo. As setas nas linhas pontilhadas indicam a direção do fluxo de dados. A seguir, encontra-se uma explicação mais detalhada de cada módulo.

O **módulo *Mobile Robot*** representa os elementos que fazem parte da estrutura física de um robô móvel. Como pode ser visto na Figura 6.1, no módulo *Mobile Robot*, no quadro interno com linhas pontilhadas, os comportamentos do robô, representados pelo elemento *Behaviors*, podem ser implementados utilizando o controle reativo, que utiliza somente os dados dos sensores e atuadores, ou controle reativo/deliberativo que além dos dados de

sensores e atuadores, utiliza também os dados de representação do ambiente que estão no elemento *Environment Representation*. A estrutura interna do módulo Mobile Robot é detalhada a seguir:

- ***Sensors***: Representa o conjunto de sensores que compõem a estrutura física do robô e são utilizados na coleta de dados do ambiente. Todos os sensores da estrutura eletromecânica do robô devem ser implementados nessa parte da arquitetura.
- ***Actuators***: Representa o conjunto de atuadores que compõem a estrutura física do robô e são utilizados para que o robô possa interagir com o ambiente. Todos os atuadores devem ser implementados nessa parte da arquitetura..
- ***Environment Representation***: Representa o ambiente de atuação do robô. São utilizados para implementar uma grade de ocupação com as métricas que representam ambientes fechados.
- ***Behaviors***: Representa as funcionalidades de controle do robô de acordo com as arquiteturas reativa, deliberativa ou mista, apresentadas por Brooks (1986), e também por Arkin (1987).

O **módulo *FeedBack Control Loop*** representa os elementos que fazem parte da estrutura de um sistema autoadaptativo de acordo com o modelo apresentado por Kephart e Chess (2001). Sua estrutura interna é composta pelos seguintes elementos:

- ***Monitor***: Detecta o processo gerenciado e seu contexto, filtra os dados coletados dos sensores e, em seguida, armazena eventos relevantes na base de conhecimento para referência futura.
- ***Analyzer***: Compara dados de eventos contra dados padrões na base de conhecimento para diagnosticar sintomas e, em seguida, armazena os sintomas na base de conhecimento para referência futura.
- ***Planner and Executor***: Interpreta os sintomas e elabora um plano para de ação mudar no processo gerenciado. O executor aplica o plano de mudança por meio dos atuadores.
- ***Knowledge Base***: A base de conhecimento armazena os dados utilizados pelos elementos do loop de controle. Esses dados são estruturados em classes que representam eventos, sintomas e políticas de adaptação (*Event*, *Event Pattern*, *Symptom*, *Adaptive Policies*).
  - ***Event:*** São dados coletados a partir dos sensores e atuadores do robô.

- ***Event Pattern***: São dados estabelecidos com valores dentro de um padrão desejado. Esses dados serão comparados com os dados coletados dos sensores.
- ***Symptom***: É a interpretação dos dados confrontado os valores de um evento contra os valores de um evento padrão.
- ***Adaptive Policies***: São comandos estabelecidos com ordem de precedência que podem ser executados pelos atuadores, alterando o comportamento do robô. Pode ser também um conjunto de valores em parâmetros que podem ser atribuídos a aplicação.

O **módulo *Touch Point*** é um módulo de integração com os outros módulos da arquitetura de referência. A função desse módulo é criar uma camada para facilitar a integração e que forneça a propriedade de baixo acoplamento entre os módulos que estão sendo integrados. Além de facilitar a integração, um outro objetivo do módulo *Touch Point* é minimizar os efeitos de alterações efetuadas em um módulo causando pouco ou nenhum impacto sobre o outro módulo integrado. A estrutura interna do módulo *Touch Point* é detalhada a seguir:

- ***Data Publisher***: Esse elemento tem acesso a todos os dados utilizados no comportamento do robô. Os dados coletados dos sensores e atuadores são armazenados em sua estrutura de dados interna.
- ***Remote Facade***: Esse elemento possui acesso aos comandos disponíveis nos comportamentos do robô e também à base de conhecimento de um loop de controle. O remote Facade deve ser implementado como o padrão Facade (Gamma *et al.*, 1995), simplificando a utilização dos comandos disponíveis nos comportamentos do robô.

6.2.1 Relacionamento e Fluxo de dados entre os módulos na arquitetura

Como já foi mencionado, o módulo *Touch Point* é utilizado para realizar a integração entre os outros módulos. No módulo *Mobile Robot*, o elemento *Behaviors* disponibiliza os dados dos sensores, atuadores e vários outros dados que são utilizados nos comportamentos do robô. No módulo *Touch Point*, o elemento *DataPublisher* tem acesso a todos os dados disponíveis no módulo *Behaviors*. Assim, em uma taxa de tempo predeterminada, o *DataPublisher* atualiza e armazena as informações em sua estrutura interna de dados.

É importante destacar que um *DataPublisher* foi projetado para poder acessar os dados de elementos do tipo *Behavior* ou de elementos do tipo *KnowledgeBase*. Ambos fazem parte dos módulos *Mobile Robot* e *FeedBack Control Loop*, respectivamente, e são as principais fontes concentradoras de dados na arquitetura. Esse tipo de relacionamento produzido pela arquitetura de referência vai permitir que o projetista de software, utilizando essa arquitetura, possa projetar loops de controle diretamente integrados ao sistema do robô ou também loops de controle integrados a outros loops de controle.

O *DataPublisher* é monitorado constantemente, no módulo *FeedBack Control Loop* pelo elemento *Monitor*, que tem acesso às informações atualizadas na estrutura de dados do *DataPublisher*. O *Monitor* filtra os dados que são de seu interesse e guarda-os em um componente *Event*. A partir dessa ação inicia-se o ciclo no *loop* de controle, o *Analyzer* compara o evento filtrado contra um evento padrão (*Event Pattern*) e insere sintomas (*Symptoms*) na base de conhecimento (*KnowledgeBase*). O elemento *Planner* interpreta os sintomas e elabora um plano corretivo. Esse plano corretivo deve seguir uma política de adaptação (*Adaptive Policies*), que estabelece uma sequência de comandos e envia o plano ao elemento *Executor*. O elemento *Excutor* envia esses comandos para o módulo *Touch Point* por meio do elemento *RemoteFacade*, que por sua vez aplica as modificações ao módulo que está sendo gerenciado. Se o módulo gerenciado for um *Mobile Robot*, as alterações são efetuadas diretamente nos comportamentos do robô por meio do elemento *Behavior*. Caso contrário, se for um outro *FeedBack Control Loop* as alterações são efetuadas diretamente na base de conhecimento por meio do elemento *KnowledgeBase*, alterando as políticas de adaptação.

Uma descrição detalhada do relacionamento entre os elementos do módulo *FeedBack Control Loop* foi necessária para especificar como os dados são utilizados no ciclo que fecha a integração entre os módulos representados no diagrama de blocos da Figura 6.1.

A seguir, são apresentados dois diagramas de classes com uma visão mais detalhada da arquitetura de referência. No diagrama da Figura 6.2 são mostradas as classes abstratas, as classes da API LeJOS e outras classes que compõem a base da arquitetura. No diagrama da Figura 6.3 são mostradas todas as classes, após a instanciação de uma aplicação robótica com um *loop* de controle.

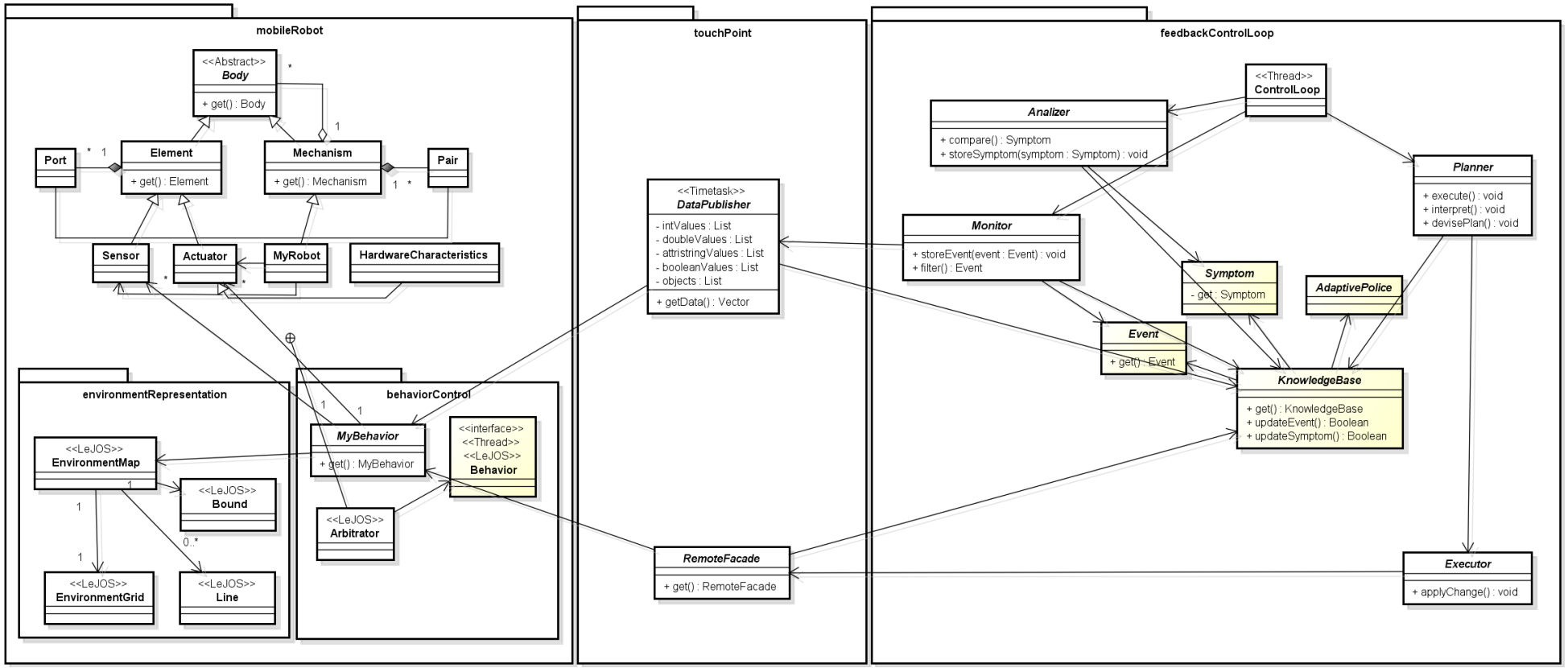

Figura 6.2: Diagrama de classes da arquitetura SARA^{MR}.

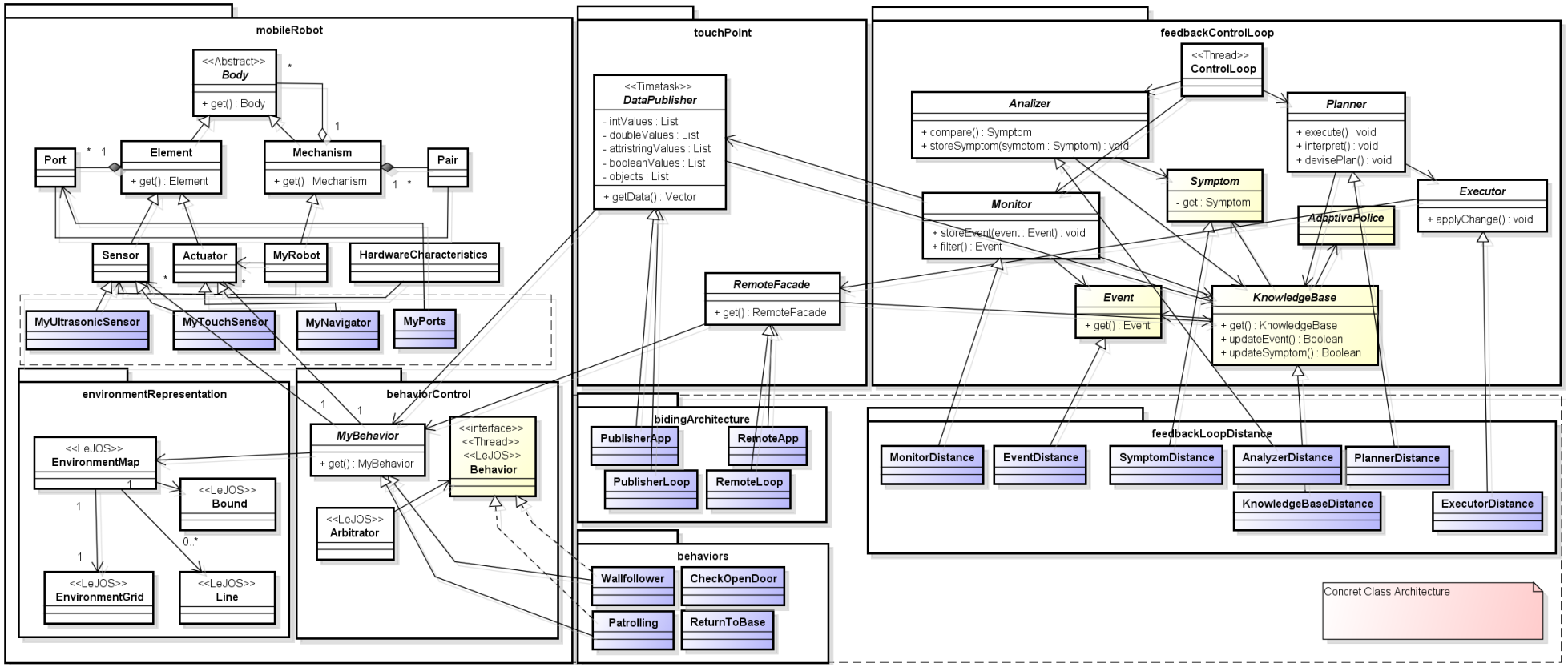

Figura 6.3 Diagrama de classes da arquitetura SARA^{MR}.

Como pode ser visto na Figura 6.2, a arquitetura foi organizada em três módulos principais; (i) o módulo **`mobileRobot`**, que representa o sistema de robô móvel, com os submódulos **`environmentRepresentation`** e **`behaviorControl`**; (ii) o módulo **`touchPoint`** de integração de sistemas; (iii) o módulo **`feedbackControlLoop`**, que representa o sistema de loop de controle. Os módulos são organizados como pacotes no projeto Java/LeJOS NXJ. Enquanto a Figura 6.1 está em um nível mais alto de abstração, e propositalmente "escondendo" detalhes da arquitetura, no diagrama da Figura 6.2 é possível observar todos os detalhes, com as classes e os relacionamentos de cada módulo. Na parte superior do módulo **`mobileRobot`** estão todas as classes que são utilizadas para a representação física do robô móvel. Na parte inferior aparecem as classes da API LeJOS NXJ nos submódulos **`environmentRepresentation`** e **`behaviorControl`.** Esses dois submódulos são utilizados para a representação do ambiente e para a implementação de comportamentos do robô. No centro é mostrado o módulo **`touchPoint`** com as duas classes que fazem a integração entre o loop de controle e a aplicação. Por fim, no lado direito, são mostrados os detalhes de todas as classes necessárias para a implementação de um loop de controle.

Na Figura 6.3 é mostrado, em cor mais escura, as classes concretas pertencentes a uma aplicação criada à partir da arquitetura de referência. Os detalhes de cada módulo são apresentados nas próximas seções.

6.3 Detalhamento do Uso da Arquitetura de Referência

Essa seção concentra-se em mostrar como a arquitetura de referência proposta pode ser usada. Ressalta-se que a arquitetura proposta foi implementada na forma de um framework, contendo um conjunto de classes abstratas e concretas. Assim, para que um sistema possa ser criado a partir dela, deve-se estender suas classes abstratas, concretizando-as. As classes utilizadas para a representação do ambiente pertencem à API LeJOS NXT, nesse caso, sua utilização é efetuada apenas instanciando as classes e não estendendo.

As próximas subseções concentram-se em mostrar como utilizar a arquitetura, sem levar em conta a criação de uma aplicação específica. Embora a ordem das subseções a seguir estejam mostrando inicialmente o detalhamento da parte eletromecânica, depois do ambiente,

depois do controle e por último da parte de integração, essa ordem não é obrigatória na implementação de uma aplicação.

6.3.1 Detalhamento da Parte Eletromecânica do Robô

Para criar uma representação física de um robô deve-se criar classes concretas que estendem as classes `Sensor` e `Actuator`. Como pode ser observado no diagrama de classes mostrado na Figura 6.3, `MyUltrasonicSensor` e `MyTouchSensor` são as classes concretas para os sensores ultrassônicos e sensor de toque do robô Lego. O método `get`() herdado da classe `Element` deve ser sobrescrito para retornar uma referência para a própria classe do sensor. O método `get`() será utilizado posteriormente para obter referência para o sensor a partir de listas genéricas e acessar os métodos da classe específica. Para acessar e ler os dados de sensores de forma individual basta referenciar pelo índice da lista. Para ler os dados de todos os sensores de uma só vez, ou seja, de forma coletiva, basta uma iteração pelos itens da lista. A mesma coisa pode ser feita com os atuadores. O projeto da arquitetura de referência segundo essa modelagem atende os requisitos REE-1 e REE-4.

A seguir, é mostrado um trecho de código da classe `MyUltrassonicSensor` que mostra como instanciar a classe `UltrasonicSensor` da API LeJOS NXT, e como atribuir ao sensor a porta de conexão.

```
public class MyUltrasonicSensor extends Sensor{

	private UltrasonicSensor us;

	public MyUltrasonicSensor() {
		us = new UltrasonicSensor(SensorPort.S1);
	}

	@Override
	public MyUltrasonicSensor get() {
		return this;
	}

gets e sets...
```

Código 6.1: Classe MyUltrassonicSensor.

A classe `MyRobot` representa o robô como um todo. Essa classe possui listas de Sensores e Atuadores, assim os dispositivos implementados devem ser adicionados a essas

listas. A seguir, é mostrado um trecho de código da classe `MyRobot` com a coleção de sensores e atuadores.

```
public class MyRobot extends Mechanism {

	private List<Sensor> sensors = new ArrayList<>();
	private List<Actuators> actuators = new ArrayList<>();
	private Arbitrator arbitrator;

	public MyRobot(MySensors sensors, MyActuators actuators) {
		this.sensors = sensors;
		this.actuators = actuators;
	}

gets e sets...
```

Código 6.2: Classe MyRobot.

As informações de configuração do robô que especificam características como o tamanho das rodas, largura entre os eixos são fornecidas no momento de instanciação da classe `HardwareCharacteristics`. Outras informações como tamanho e peso do robô são informadas no momento de instanciação da classe `MyRobot`. Essa modelagem atende os requisitos REE-2, REE-3 e REE-4. A seguir, é mostrado um trecho de código da classe `HardwareCharacteristics` que mostra como configurar as características do robô.

```
public class HardwareCharacteristics {

	private double wheelDiameter = 5.6f;
	private double trackWidth = 11.7f;
	private RegulatedMotor leftMotor = Motor.A;
	private RegulatedMotor rightMotor = Motor.C;
	private DifferentialPilot pilot;

	public HardwareCharacteristics(){

	}

gets e sets...
```

Código 6.3: Classe HardwareCharacteristics.

A classe `MyNavigator` estende `Actuator` e usa a classe `DiferencialPilot` que pertence à API LeJOS NXT. Essa classe da API fornece suporte para o controle simultâneo de dois motores. Caso seja necessário implementar os motores individualmente, basta proceder da mesma forma estendendo a classes `Actuator` para cada motor do robô.

6.3.2 Detalhamento do Módulo de Representação do Ambiente

Para criar a representação de um ambiente não é necessário criar classes novas, basta utilizar as classes já disponibilizadas pela API LeJOS NXT. A implementação das classes para a representação da grade de ocupação e dos elementos existentes no ambiente é realizada por meio de informações métricas em um plano cartesiano. A classe `Bound` é uma representação geométrica de um retângulo que delimita as fronteiras do ambiente fechado. A classe `Line` é utilizada para "desenhar" figuras geométricas que preenchem a grade de ocupação representando os elementos no ambiente, o espaço que eles ocupam, por onde o robô não poderá transitar.

A classes `Bound` e todos os objetos instanciados da classe `Line` devem ser adicionados à classe `EnvironmentMap` que representa o ambiente como um todo. Essa modelagem atende os requisitos REA-1, REA-2 e REA-3.

O código a seguir mostra como um determinado ambiente pode ser instanciado.

```
public class EnvironmentMap {

	private FourWayGridMesh grid;
	private LineMap environment;

	public EnvironmentMap(Rectangle bounds, Line[] lines) {

		environment = new LineMap(lines, bounds);
		grid = new FourWayGridMesh(environment, 10, 10);
	}

gets e sets...
```

Código 6.4: Classe EnvironmentMap.

Após a implementação inicial da representação do ambiente, caso seja necessário realizar alterações na disposição ou localização dos elementos no ambiente, abrangendo também a inclusão ou remoção de elementos, basta acessar os métodos da classe `EnvironmentMap` e alterar linhas ou fronteiras. Isso se aplica também na representação de novos ambientes. Essa modelagem atende os requisitos REA-4 e REA-5.

6.3.3 Detalhamento do Módulo de Controle do Robô

Essa parte da arquitetura compreende a forma de controle do robô que foi projetada para atender as funcionalidades de controle baseado em comportamentos que podem ser implementados nas formas de controle reativo, deliberativo ou misto.

Para criar os comportamentos do robô deve-se criar classes concretas que estendem a classe `MyBehavior` e implementem a interface `Behavior` da API LeJOS NXT. Os comportamentos devem ser implementados individualmente em blocos de código mais coesos com funcionalidades mais específicas. Assim o controle do robô pode dispor de vários comportamentos, ordenados para trabalhar em conjunto, como processos controlados, na realização de uma determinada tarefa.

Se a opção do arquiteto de software for projetar uma aplicação que funcione de forma reativa, o módulo de representação do ambiente não é utilizado e o comportamento não recebe uma instância de objetos desse módulo. Caso a opção seja por uma aplicação na forma deliberativa ou mista, o módulo de representação do ambiente deve ser implementados e os comportamentos concretizados podem ser utilizados nas duas formas, podendo receber ou não uma instância para os objetos do módulo de representação do ambiente.

`Behavior` é uma `Thread` e essa classe faz parte da API LeJOS. Possui os métodos: `takeControl()`, `action()` e `suppress()`. Esses três métodos são utilizados pela classe `Arbitrator` para controle de concorrência na gerência de comportamentos. Essa modelagem atende aos requisitos REC-1, REC-2, REC-3 e REC-4.

Para ser um comportamento concreto, a classe deve estender `MyBehavior` e implementar a interface `Behavior`. Depois de especializada, classes filhas de `MyBehavior`, possuem referências para todos os dados que são utilizados no comportamento, inclusive para as listas de sensores e atuadores instanciados na aplicação. Isso significa que essas classes concentram toda a informação que circula pela aplicação do robô. A seguir é mostrado um trecho de código da classe abstrata `MyBehavior`.

```
public abstract class MyBehavior {

	private MySensors sensors;
	private MyActuators actuators;
	private boolean onOff;

	public MyBehavior(MySensors sensors, MyActuators actuators) {
```

```
            this.sensors = sensors;
            this.actuators = actuators;
      }

gets e sets...
```

Código 6.5: Classe MyBehavior.

Trecho de código da classe `WallFollower` que é a implementação de um comportamento do robô.

```
public class WallFollower extends MyBehavior implements Behavior{

      private boolean suppressed;

      public WallFollower(MySensors sensors, MyActuators actuators) {
            super(sensors, actuators);
      }

      @Override
      public void action() {
            while(super.isOnOff()){
                  super.getActuators().getMyNavigator().getPilot().steer(0);
            }
      }
```

Código 6.6: Classe wallFollower.

6.3.4 Detalhamento do Módulo de Integração

Para criar um módulo de integração *Touch Point* basta estender as duas classes que compõem o módulo, `DataPublisher` e `RemoteFacade`. A classe abstrata `DataPublisher` implementa a interface `TimeTask` da API LeJOS NXT, isso significa que o método run() deve ser concretizado, e também deve ser passada uma taxa de tempo para acionar o método. Essa classe também possui uma estrutura de dados com listas de valores para vários tipos de dados, a saber: inteiros, booleanos, *strings*, *object*, etc.

A classe `DataPublisher` recebe uma referência para `MyBehavior` e dessa forma possui acesso aos dados disponíveis nos comportamentos incluído sensores e atuadores. O método `run()` deve ser implementado de forma que os dados de interesse sejam coletados e armazenados em sua estrutura de dados. A frequência de atualização desses dados ocorre na taxa de tempo configurada no timer.

A classe `RemoteFacade` também recebe uma referência para `MyBehavior` e dessa forma possui acesso aos dados disponíveis nos comportamentos incluído sensores e atuadores.

Essa classe deve ser implementada como um controle remoto com todos os comandos disponíveis nas classes do tipo `MyBehavior` quando o sistema gerenciado é o software do robô ou com todos os comandos disponíveis nas políticas de adaptação de uma classe `KnowledgeBase` quando o sistema gerenciado é outro loop de controle. Essa modelagem permite que o módulo *Touch Point* atue como uma ponte de integração entre um sistema de loop de controle e um sistema gerenciado. Com isso é atendido o requisito RED-5. A seguir é mostrado um trecho de código da classe abstrata `DataPublisher`.

```
public abstract class DataPublisher extends TimerTask {

	private Timer timer;
	private long ini;
	private long rate;
	private List<Integer> intValues;
	private List<Boolean> booleanValues;
	private List<Double> doubleValues;
	private List<String> stringValues;
	private List<Object> objects;
	private List<MyBehavior> behaviors;
	private KnowledgeBase knowledgeBase;

	public DataPublisher(List<MyBehavior> behaviors, KnowledgeBase
knowledgeBase, long ini, long rate) {
		intValues = new ArrayList<Integer>();
		booleanValues = new ArrayList<Boolean>();
		doubleValues = new ArrayList<Double>();
		stringValues = new ArrayList<String>();
		objects = new ArrayList<Object>();
		this.behaviors = behaviors;
		this.knowledgeBase = knowledgeBase;
		timer = new Timer();
		this.ini = ini;
		this.rate = rate;
	}
gets e sets...
```

Código 6.7: Classe DataPublisher

A seguir é mostrado um trecho de código da classe abstrata `RemoteFacade`.

```
public abstract class RemoteFacade {

	private List<MyBehavior> behaviors;
	private KnowledgeBase knowledgeBase;

	public RemoteFacade(List<MyBehavior> behaviors, KnowledgeBase
knowledgeBase) {
		this.behaviors = behaviors;
		this.knowledgeBase = knowledgeBase;
	}
```

```
gets e sets...
```

Código 6.8: Classe RemoteFacade.

6.3.5 Detalhamento do Módulo de Loop de Controle

Para criar um loop de controle é necessário concretizar todas as classes que compõem o módulo, a saber: `Monitor`, `Analizer`, `Planner`, `Executor`, `KnowledgeBase`, `Event`, `Symptom`, `AdaptivePolice` e `ControlLoop`. O módulo de loop de controle é o foco principal desse trabalho de mestrado. Esse módulo foi projetado, com base na combinação de duas propostas. Kephart e Chess (2001) apresentam uma proposta para a arquitetura de um loop de controle, e Brun *et al*. (2009) mostram o papel de atuação e interação entre cada elemento para prover funcionalidades de autoadaptação em tempo de execução. Essa modelagem atende os requisitos RED-1 e RED-3.

A primeira classe a ser concretizada é a classe `Event`, e deve ser implementada de acordo com os dados a serem monitorados a partir da aplicação gerenciada. Essa classe será utilizada para instanciar dois objetos um com os dados que vem da aplicação e outro com os dados estabelecidos de acordo com um padrão desejado.

De acordo com Albus (2002), a observação das correlações entre sensoriamento e a representação interna geram expectativas que são utilizadas para detectar e classificar eventos e situações. Uma determinada situação ou evento é a transformação de informações dos sensores, incluindo relações semânticas, em representações simbólicas, modelada em termos de objetos. Um sintoma é uma previsão gerada a partir da observação da diferença entre os dados coletados e os dados da representação interna. Tanto eventos como sintomas são guardados na base de conhecimento.

Baseado nessas informações, as classes abstratas `Event` e `Symptom` são classes bem básicas, e devem ser utilizadas para modelar eventos e sintomas em termos de objetos. Na concretização e especialização de ambas as classes o método `get()` deve ser implementado para retornar uma referência de si mesma, ou seja da classe especializada. Esse recurso permitirá acesso aos métodos implementados na especialização. Um evento padrão, com informações que representam uma situação desejada pelo loop de controle, deve ser instanciado e atribuído a uma `KnowledgeBase`.

A seguir é mostrado um trecho de código da implementação de uma classe para armazenar um número intero e é utilizado como um evento que representa a distância entre o robô e uma parede:

```
public class EventDistance extends Event {

	private int distance;
	private int position;

	public EventDistance(int id, String description) {
		super(id, description);
	} gets e sets...
```

Código 6.9: Classe EventDistance.

A classe `KnowledgeBase` foi projetada para armazenar eventos, sintomas e políticas de adaptação. Além disso, essa classe possui dois atributos booleanos, `updateEvent()` e `updateSymptom()`. Esses atributos indicam quando eventos e sintomas são atualizados pelas classes do tipo `Monitor` e `Planner`.

A classe `Monitor` tem a função de coletar dados de um `DataPublisher` e armazená-los na `KnowledgeBase`. Para isso, essa classe possui os métodos `storeEvent`() e `filter`(). O método `filter`() deve ser implementado para uma filtragem na estrutura de dados de um objeto da classe `DataPublisher`, selecionar os dados e atribuí-los a um `Event`. Esse método deve obrigatoriamente retornar um objeto `Event`. O método `storeEvent()` recebe o retorno de `filter()`, ou seja um evento, e insere o evento na `KnowledgeBase`. O método `storeEvent()` é invocado pelo gerenciador do loop de controle. A seguir é mostrado um trecho de código da classe `Monitor` e do método `filter()`.

```
public abstract class Monitor {

	private KnowledgeBase knowledgeBase;
	private DataPublisher publisher;

	public Monitor(KnowledgeBase knowledgeBase, DataPublisher publisher) {
		this.knowledgeBase = knowledgeBase;
		this.publisher = publisher;
	}

	public abstract Event filter();

	public void storeEvent(){
```

```
            this.knowledgeBase.setEvent(filter());
            this.knowledgeBase.setUpdateEvent(true);
      }

      gets e sets...
```

Código 6.10: Classe Abstrata Monitor

A classe `Analizer` tem a função de comparar eventos a fim de diagnosticar sintomas e armazená-los na `KnowledgeBase`. Para isso, essa classe possui os métodos `compare()` e `storeSymptom()`. O método `compare()` deve ser implementado para acessar a `KnowledgeBase` e realizar um procedimento de diagnóstico de sintoma, ou seja, uma comparação entre um evento coletado e um evento padrão. O resultado da comparação é atribuida a um `Symptom`. Esse método deve obrigatoriamente retornar um objeto `Symptom`. O método `storeSymptom()` recebe o retorno de `compare()`, ou seja, recebe um `Symptom`, e insere o sintoma na `KnowledgeBase`. O método `storeSymptom()` é invocado pelo gerenciador do loop de controle. A seguir é mostrado um trecho de código da classe `Analizer` e do método `compare`():

```
public abstract class Analyzer {

      private KnowledgeBase knowledgeBase;

      public Analyzer(KnowledgeBase knowledgeBase) {
            this.knowledgeBase = knowledgeBase;
      }

      public abstract Symptom compare();

      public void storeSymptom(){
            this.knowledgeBase.setSymptom(compare());
            this.knowledgeBase.setUpdateEvent(false);
            this.knowledgeBase.setUpdateSymptom(true);
      }

      public KnowledgeBase getKnowledgeBase() {
            return knowledgeBase;
      } gets e sets...
```

Código 6.11: Classe Abstrata Analyzer.

A classe `Planner` tem a função acessar `KnowledgeBase` e interpretar o sintoma e elaborar um plano de adaptação que restabeleça o sistema gerenciado à situação desejada. O plano deve ser elaborado de acordo com políticas de adaptação.

De acordo com Georgas e Taylor (2008) políticas de adaptação podem ser definidas como o encapsulamento de comportamentos adaptativos, atribuídos ao sistema que os utiliza

para reagir a uma situação. Um comportamento determina quais ações devem ser tomadas em resposta a eventos que indiquem a necessidade dessas ações. Diante disso, na arquitetura SARA^{MR}, a implementação de políticas de adaptação pode ser realizada com blocos básicos de comandos organizados em ordem de prioridade. Um objeto da classe `AdaptivePolicie` pode ser implementado com um conjunto de políticas de adaptação. Uma ou mais de uma delas poderá ser aplicada ao sistema dependendo da situação. Nesta arquitetura de referência, a classe responsável por determinar essa escolha é a `Planner`. Essa modelagem atende o requisito RED-2 .

Para realizar a sua função, a classe `Planner` possui os métodos `interpret()`, `devisePlan()` e `execute()`. O método `interpret()` deve ser implementado na classe concreta, de acordo com a regra que identifique a melhor relação entre sintoma e política de adaptação, com isso um plano de correção é estabelecido. O método `devisePlan()` deve ser implementado com a função de passar o plano de correção para o `Executor`. O método `execute()` é invocado pelo gerenciador do loop de controle. A seguir é mostrado um trecho de código da classe abstrata `Planner`:

```
public abstract class Planner {

	private KnowledgeBase knowledgeBase;
	private Executor executor;

	public Planner(KnowledgeBase knowledgeBase, Executor executor) {
		this.knowledgeBase = knowledgeBase;
		this.executor = executor;
	}
	public abstract void interpret();
	public abstract void devisePlan();

	public void execute(){
	interpret();
	devisePlan();
	this.knowledgeBase.setUpdateSymptom(false);
	executor.applyChange();
} gets e sets...
```

Código 6.12: Classe Abstrata Planner.

A classe `Executor` possui o método `aplyChange()` e também possui uma referência para a classe `RemoteFacade` do módulo *Touch Point*. A seguir é apresentado o código fonte da classes `Executor` e o método `aplyChange()`.

```
public abstract class Executor {
```

```
	private RemoteFacade remoteFacade;

	public Executor(RemoteFacade remoteFacade) {
		this.remoteFacade = remoteFacade;
	}

	public abstract void applyChange();

gets e sets...
```

Código 6.13: Classe Abstrata Executor.

O método `aplyChange()` deve ser implementado para aplicar o plano de adaptação por meio da fachada de comandos disponíveis no `RemoteFacade`. Como já foi mencionado, o `RemoteFacade` pode ser usado como um controle remoto enviando comandos diretamente para os comportamentos do sistema gerenciado.

O gerenciamento interno do módulo de *FeedBack Control Loop* é uma função da classe `ControlLoop`. Essa classe é uma `Thread` e possui o método `run()` que dispara o ciclo de funcionamento do loop de Controle. Um ciclo somente se completa através do loop de controle, quando o Monitor insere um novo sintoma na base de conhecimento e assinala essa ação por meio do método `setUpdateEvent()`. Esse conceito foi implementado na arquitetura como atributo de qualidade visando melhor desempenho, tanto do sistema autonômico quando do sistema gerenciado. Com isso é atendido o requisito RED-4 “.

Essa medida é importante para a arquitetura, uma vez que as classes `ControlLoop` e `Behavior` são `Threads` que estão em constante concorrência pelo processamento do sistema. É importante ressaltar que comportamentos e *loops* de controle podem ser implementados tantos quanto for necessário e todos eles vão concorrer pelo processamento do sistema.

A seguir é mostrado um trecho de código da classe abstrata `ControlLoop` e o método `run`().

```
public class ControlLoop extends Thread {

	private boolean on;
	private Monitor monitor;
	private Analyzer analyzer;
	private Planner planner;

	public ControlLoop(Monitor monitor, Analyzer analyzer, Planner planner) {
		this.monitor = monitor;
		this.analyzer = analyzer;
```

```
            this.planner = planner;
      }

@Override
public void run() {

      while (isOn()) {
            monitor.storeEvent();
            if (analyzer.getKnowledgeBase().isUpdateEvent()) {
                  analyzer.storeSymptom();
            }
            if (planner.getKnowledgeBase().isUpdateSymptom()) {
                  planner.execute();
            }
      }
}
```

Código 6.14: Classe ControlLoop.

6.4 Considerações Finais

Neste capítulo foi mostrada uma visão completa e explícita de todos os módulos que compõem a arquitetura de referência SARA^{MR}. Também foi apresentado, de forma detalhada, como utilizar a arquitetura como referência para a implementação de software no domínio de aplicações para RMAs. A partir do detalhamento de cada módulo da arquitetura, conforme as funcionalidades foram sendo explicitadas, foi mostrado uma relação entre funcionalidade e requisito, como arquitetura SARA^{MR} foi projetada para atender aos requisitos de software especificados nos cenários de evolução.

No próximo capítulo, serão abordados dois exemplos de aplicação que serão apresentados com o objetivo de completar a tarefa de avaliação e entendimento da arquitetura SARA^{MR}.

Capítulo 7

EXEMPLO DE APLICAÇÕES ROBÓTICAS DESENVOLVIDAS COM A ARQUITETURA SARA^{MR}

7.1 Considerações Iniciais

Este capítulo tem a finalidade de apresentar dois exemplos de aplicações desenvolvidas com o uso da arquitetura SARA^{MR}. As duas aplicações foram desenvolvidas na linguagem Java com a API LeJOS NXJ.

A primeira aplicação é para um robô seguidor de paredes e sua particularidade é a presença de dois *loops* de controle, um sobre o outro. Isso evidencia a capacidade da arquitetura SARA^{MR} em apoiar o desenvolvimento de sistemas autoadaptativos. A segunda aplicação consiste em um robô de monitoramento interno. Esse segundo exemplo não possui o objetivo de exemplificar detalhes de adaptação, mas mostrar que sistemas robóticos que precisam de representações internas de ambiente também podem se beneficiar da arquitetura.

7.2 Aplicação 1: Robô Seguidor de Parede

Essa aplicação consiste em um robô que segue um caminho a frente, de forma totalmente autônoma e deve manter a distância regular de 20 centímetros da parede. Esse tipo de experimento já é muito conhecido e é um clássico na área de pesquisa de RMAs, porem, é muito importante para o contexto deste trabalho, pois propicia a oportunidade de utilizar dois

loops de controle ativos simultaneamente adaptando o comportamento do robô em tempo real. É importante destacar, que para essa aplicação em particular, a forma de controle do robô é realizada de forma exclusivamente reativa, ou seja, o robô não possui nenhuma representação interna do ambiente e a locomoção é feita totalmente baseada na leitura de informações oriundas de sensores e atuadores. O controle do robô é feito com apenas um único comportamento, dessa forma não existe ainda a necessidade de se implementar concorrência entre comportamentos.

A aplicação utiliza como base a arquitetura de referência SARA^{MR} com todos os módulos e as classes abstratas conforme foi descrito no capítulo 6. Será utilizado também algumas classes da API LeJOS NXJ que tem a função de acessar os sensores e os atuadores para leitura de dados e envio de comandos. O objetivo principal da aplicação é mover o robô para frente, em linha reta, mantendo uma distância fixa em relação à parede do lado direito do robô.

7.2.1 Implementação da parte Eletromecânica

Conforme é mostrado na Figura 7.1, a configuração física do robô consiste de um microcontrolador montado com peças no formato de um veículo, com duas rodas dianteiras e uma pequena roda na parte de trás funcionando como contra apoio. Cada uma das duas rodas dianteiras possui um atuador acoplado para realizar a força de locomoção. Um sensor ultrassônico é colocado na parte da frente do robô, apontando para o lado direito, e é utilizado para medir constantemente a distância entre o robô e a parede. O robô possui também um sensor de toque colocado na parte dianteira.

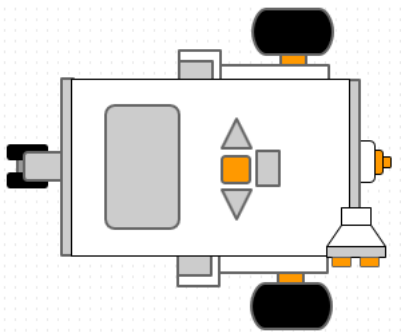

Figura 7.1: Configuração física do robô Lego NXT.

Na Figura 7.2 são mostradas as classes implementadas para a representação física do robô.

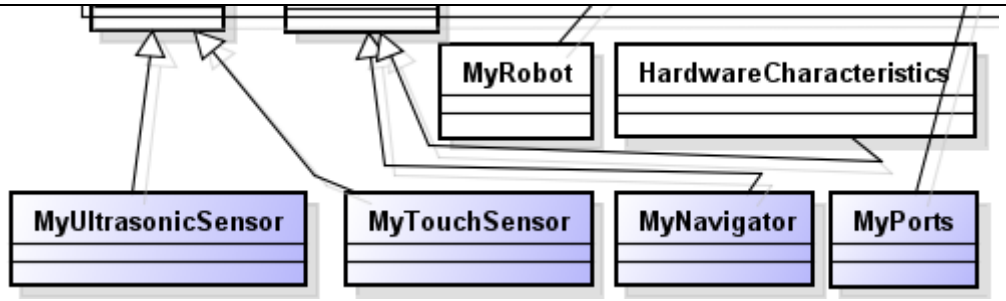

Figura 7.2: Classes para a configuração física do robô.

A implementação do sensor ultrassônico e do sensor de toque segue as instruções da arquitetura de referência, ou seja, os sensores da API LeJOS NXT são instanciados internamente nas classes `MyUltrasonicSensor` e `MyTouchSensor`. A classe `MyRobot` e `MyPorts` são implementadas para representar o robô Lego. Os dois sensores instanciados são adicionados à coleção de sensores da robô. Os atuadores que representam os motores das duas rodas do robô são instanciados na classe `DiferencialPilot` API LeJOS NXT, que é passada por referência à classe `MyNavigator`. As configurações de tamanho de roda e distância entre eixos são definidas na classe `HardwareCharacteristics`.

7.2.2 Implementação do Controle do Robô

Nesse exemplo de aplicação o robô conta basicamente com apenas um comportamento. Uma classe `WallFollower` é implementada estendendo a classe `MyBehavior`.

Na figura 7.3 é mostrada uma pequena parte do diagrama de classes com a classe `Wallfollower`.

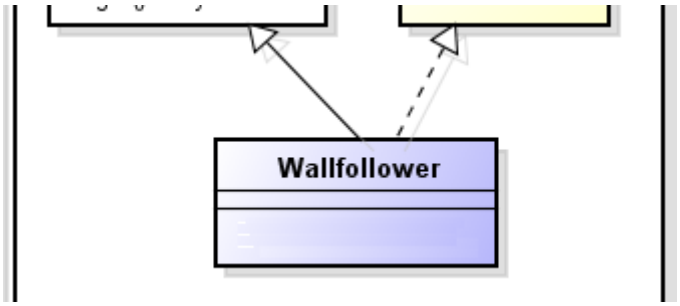

Figura 7.3: Classe Wallfollower para implementação do comportamento.

A função desse comportamento é fazer com que o robô siga adiante enquanto não colidir com um obstáculo à sua frente. Se isso ocorrer, o sensor de toque é acionado e o comportamento básico do robô envia um comando de *stop* para os dois motores, fazendo o

robô parar simultaneamente. O comportamento funciona de forma reativa, ou seja, não foi implementado o módulo de representação do ambiente, assim o robô conta apenas com os dois sensores, o ultrassônico e o sensor de toque, para perceber a sua situação no ambiente e se locomover.

7.2.3 Implementação do Loop de Controle

Para cumprir a tarefa de pilotar corretamente em linha reta, sem bater na parede, a aplicação implementada conta com uma funcionalidade de piloto automático. Para isso foi implementado um módulo de loop de controle de acordo com o modelo MIAC *Model Identification Adaptive Control* (SODERSTROM e STOICA, 1988).

A implementação do módulo de piloto automático é realizada a partir da concretização das classes que compõem o módulo de loop de controle. As classes abstratas que constituem o módulo são o `Monitor`, `Analizer`, `Planner`, `Executor`, `KnowledgeBase`, `Event`, `Symptom`, `AdaptivePolice` e `ControlLoop`.

Na Figura 7.4 é mostrado uma parte do diagrama de classes com as classes concretas que foram implementadas para um dos loops de controle.

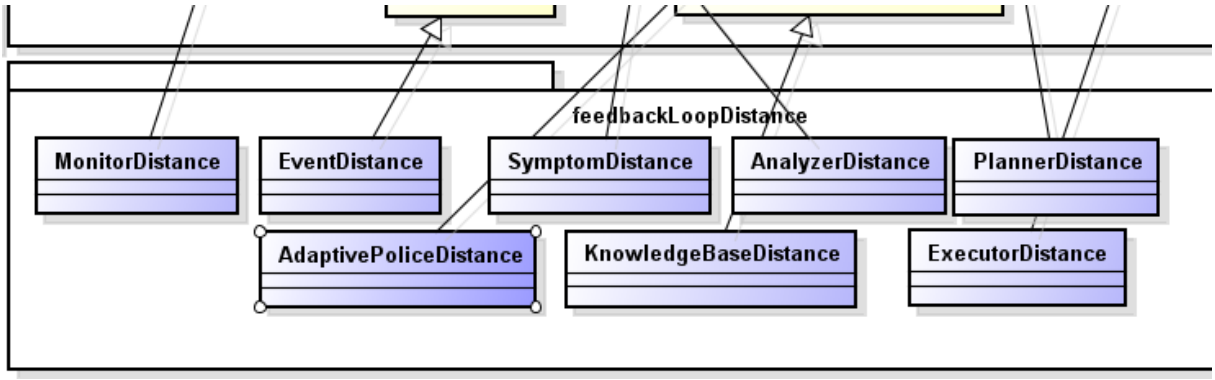

Figura 7.4: Classes do concretas do módulo de loop de controle.

Como pode ser visto na figura, são oito classes que compõem o ciclo de autoadaptação de um loop de controle, a função desse loop é monitorar e ajusta a distância em relação à parede enquanto o robô segue em frente. A Figura 7.5 ilustra como é a integração do loop de controle com a aplicação base do robô, nela é possível ver os principais elementos, assim como o fluxo de dados pela arquitetura.

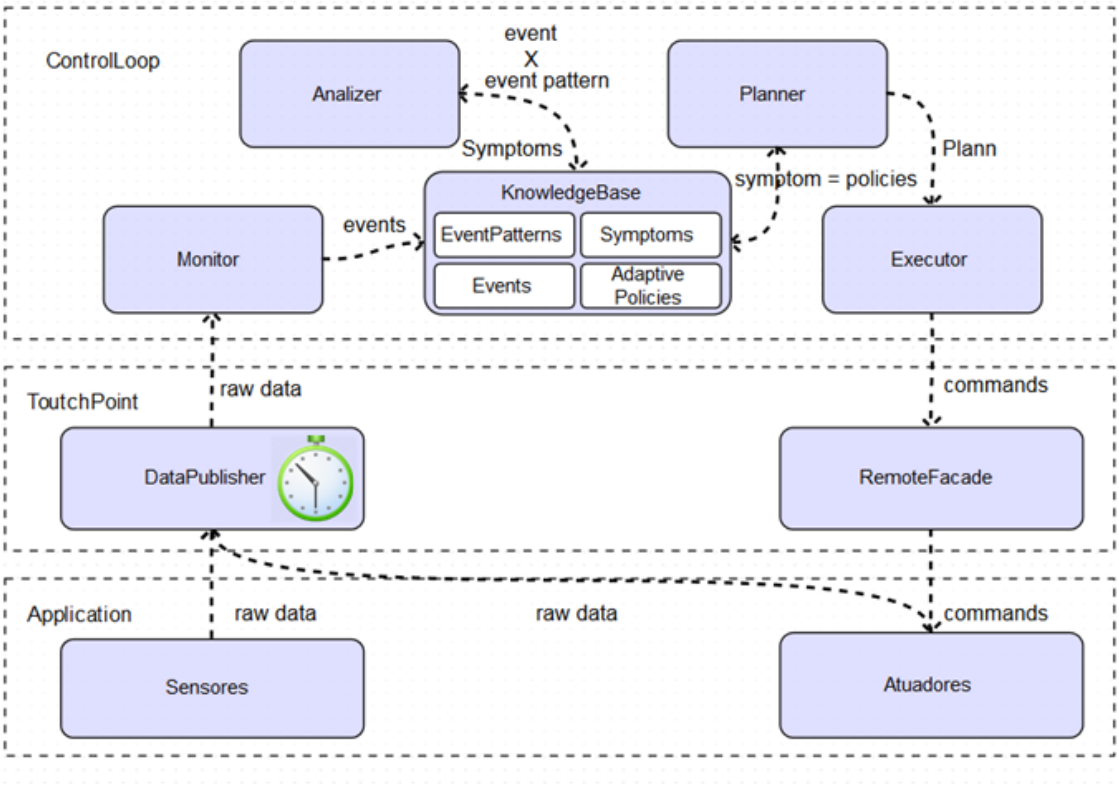

Figura 7.5: Principais elementos do piloto automático e seu fluxo de dados.

Como pode ser visto na Figura 7.5, a classe `DataPublisher` recebe dados da aplicação. A distância atual vem do sensor ultrassônico, a velocidade atual vem dos atuadores. Esses dados são atualizados a cada 500 milissegundos. O ciclo do fluxo de dados no módulo de loop de controle se inicia na classe `Monitor` que está em constante monitoração de eventos sobre os dados que vem do `DataPublisher`. A classe `DataPublisher` recebe uma um objeto `List` com todos os comportamentos do robô. A partir do índice dessa lista se obtêm uma referência para a coleção de sensores e atuadores da aplicação e dessa forma tem acesso a todos os dados de métricas que são utilizados no comportamento do robô. Um `DataPublisher` é uma classes que estende um `Timer` das classes básicas da API LeJOS, assim essa classe tem uma taxa temporal configurável e a cada 500 milissegundos os dados vindos dos sensores são atualizados no `DataPublisher`.

O objetivo principal do loop de controle, é que o robô siga em frente, evitando uma aproximação que resulte na colisão com a parede, mantendo a distância constante de 20cm, evitando também que o robô se distancie muito em relação à parede, como é mostrado na Figura 7.6.

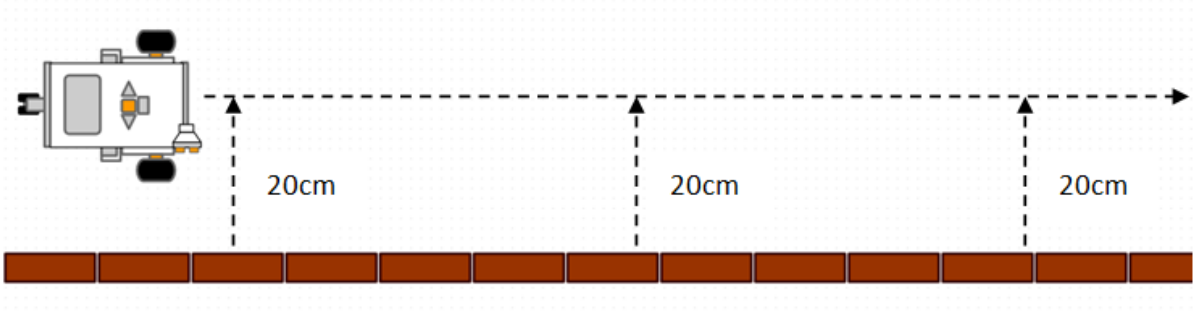

Figura 7.6: Robô seguindo a parede.

A integração do módulo de loop de controle com aplicação é realizada implementado um módulo de `TouchPoint` da mesma forma, estendendo as classes abstratas `DataPublisher` e `RemoteFacade` da arquitetura de referência.

Como nesta aplicação foram criadas novas classes à partir das abstratas, doravante serão usados os nomes das classes concretas para melhorar o entendimento do papel de cada uma das classes: `DataPublisherApp`, `RemoteFacadeApp`, `MonitorDistance`, `AnalyzerDistance`, `PlannerDistance`, `ExecutorDistance`, `KnowledgeBaseDistance`, `EventDistance`, `EventPatternDistance`, `SymptomDistance`, `AdaptivePoliceDistance`.

`MonitorDistance` é a primeira classe implementada para o loop de controle. Essa classe recebe referência para um `DataPublisherApp` e para um `KnowledgeBaseDistance`, sua função é monitorar e gravar eventos na base de conhecimento. A classe `MonitorDistance` tem o método `filter()` sobrescrito para filtrar e os eventos de interesse a esse loop de controle. O método `storeEvent()`, mostrado na classe abstrata, vai invocar o método `filter()` para armazenar os eventos em uma `KnowledgeBaseDistance`. O método de `filter()` deve, obrigatoriamente, retornar um objeto `Event`, o código de implementação desses dois métodos é mostrado a seguir.

```
public class MonitorDistance extends Monitor{
...
@Override
public Event filter() {
	EventDistance event;
	event = new EventDistance(1, "distance");
	event.setDistance(super.getPublisher().getIntValue(0));
	return event;
}
```

Código 7.1: Classe MonitorDistance.

O método `setUpdateEvent()` é uma implementação padrão da arquitetura de referência como medida de desempenho. Sempre que um novo evento é armazenado essa ação é sinalizada como verdadeira na base de conhecimento, isso permite ao componente `ControlLoop` gerenciar a execução de um ciclo completo entre os componentes do loop de controle evitando processamento desnecessário. O método `run`() que executa esse procedimento no `ControlLoop` é mostrado a seguir.

```
public class ControlLoop{
...
@Override
public void run() {
	while (isOn()) {
		monitor.storeEvent();
		if (analyzer.getKnowledgeBase().isUpdateEvent()) {
			analyzer.storeSymptom();
		}
		if (planner.getKnowledgeBase().isUpdateSymptom()) {
			planner.execute();
		}
	}
}
```

Código 7.2: Classe ControlLoop.

É importante destacar o que é um evento de interesse para o loop de controle e como implementar esse componente em termos de modelagem de objetos. O loop de controle dispõe de dois eventos que foram modelados estendendo a classe abstrata `Event` e serão utilizados nas funcionalidades de monitoramento e análise. O primeiro evento é chamado de `EventPatternDistance` (evento padrão) e no caso específico desse loop de controle, guarda dois valores inteiros, um referente a distância padrão a ser mantida, no caso 20 cm, e o outro valor é a velocidade padrão. O segundo evento, chamado de `EventDistance`, estendendo a mesma classe, também possui dois valores inteiros, a diferença é que os dados desse objeto imputados pelo componente `DataPublisherApp` de acordo com a leitura dos dados de distância vindos do sensor ultrassônico e com os dados da velocidade vindos dos atuadores.

O `AnalyzerDistance` é o próximo componente no ciclo do loop de controle, sua função e analisar eventos e diagnosticar sintomas, para isso serão comparados eventos da aplicação com eventos padrão. Dependendo do resultado da análise esse componente determina se será necessário incluir um sintoma na base de conhecimento. A analise somente é realizada se a sinalização de que existe um novo evento na base de conhecimentos for

verdadeira. Esse procedimento é realizado pelos métodos `compare`() e `storeSymptom`() sobrescritos a partir da classe abstrata da arquitetura de referência. Utilizando métodos de acesso implementados nas classes de eventos os valores de `EventDistance` e `EventPatternDistance` são confrontados, se os valores são diferentes um sintoma é gravado e essa ação é sinalizada na base de conhecimento para o próximo componente. Após realizar a análise a sinalização de novo evento é alterada para falsa e a de novo sintoma é alterada para verdadeira. No caso específico desse loop de controle se a distância medida no sensor for diferente de 20cm um objeto da classe `SymptomDistance` é gravado. Nesse caso um sintoma é uma classe implementada com um atributo inteiro que representa a diferença entre o valor lido do sensor e o valor padrão. Se o valor for positivo significa que o robô esta com uma distância maior no que o desejado, se for um valor negativo significa que o robô está mais próximo da parede do que é esperado, em ambos os casos o indicativo é de um sintoma de desvio. A implementação do método `storySymptom()` na classe abstrata e `compare`() na classe concreta pode ser vista no código a seguir.

```
public class AnalyzerDistance extends Analyzer{

...
@Override
public Symptom compare() {
	SymptomDistance symptomDistance;

	int patternDistance = ((EventDistance) ((KnowledgeBaseDistance)
	super.getKnowledgeBase().get()).getEventPattern().get()).getDistance();
	int currentDistance = ((EventDistance) ((KnowledgeBaseDistance)
	super.getKnowledgeBase().get()).getEvent().get()).getDistance();

	if(currentDistance != patternDistance){
		symptomDistance = new SymptomDistance(1, "deviation");
		symptomDistance.setDeviation(currentDistance - patternDistance);
		return symptomDistance;
	} else {
		return null;
	}
}
```

Código 7.3: Classe AnalyzerDistance.

O próximo componente na sequência é o `PlannerDistance`, sua função é interpretar sintomas e elaborar um plano de ações corretivas baseadas em uma política de adaptação. Essa classe está interessada em sintomas e só entra em atividade quando a sinalização de que existe um novo sintoma é verdadeira na base de conhecimentos.

Nesse ponto, é importante destacar o que é uma política de adaptação para o loop de controle, e como implementar esse componente em termos de modelagem de objetos. Geralmente a classe que representa as políticas de adaptação pode ser modelada com o padrão de projetos *Command* (GAMMA *et al.*,1995), para organizar uma coleção de comandos, obedecendo uma ordem de precedência específica para uma determinada ação corretiva. Outra abordagem para políticas de adaptação pode ser o uso de algoritmos específicos. Essas abordagens ou a combinação delas são utilizadas para compor o plano de adaptação do loop de controle.

Para este loop de controle a classe `AdaptivePoliceDistance` foi implementada com um algoritmo de controle de PID (Proporcional, Integral e Derivativo). Esse algoritmo calcula um fator de correção em função da diferença entre um valor de erro medido em um processo e um valor de ajuste desejado (NORMEY-RICO *et al*, 2002; ZHUANG e ATHERTON, 1993; COMINOS e MUNRO, 2002). Resumindo, o `PlannerDistance` usa o `AdaptivePoliceDistance` invocando o PID que recebe o valor de desvio do componente `SymptomDistance` e retorna um valor de correção que deve ser passado ao `ExecutorDistance` e aplicado nos atuadores por meio do `RemoteFacadeApp` para corrigir a trajetória do robô. Esse procedimento é realizado pelos métodos `interpret()` e `devisePlan()` da classe `PlannerDistance`.

```
public class PlannerDistance extends Planner{
...
@Override
public void interpret() {
	deviation = ((SymptomDistance) ((KnowledgeBaseDistance)
super.getKnowledgeBase().get()).getSymptom()).getDeviation();
}

@Override
public void devisePlan() {
	steer = ((AdaptivePoliceDistance)
super.getKnowledgeBase().getAdaptivePolice().get()).getPID().doPID(deviation);
	((ExecutorDistance) super.getExecutor().get()).setSteer(steer);
}
```

Código 7.4: Classer PlannerDistance.

A classe `ExecutorDistance` tem uma referência para a classe `RemoteFacadeApp`, assim o comando de adaptação passado para ser executado na aplicação por meio do método `applyChange()`.

```
public class ExecutorDistance extends Executor{
@Override
public void applyChange() {
	((RemoteApp) super.getRemoteFacade().get()).steer(steer);
}
```

Código 7.5: Classe ExecutorDistance.

A classe `RemoteFacadeApp` funciona como um controle remoto para a aplicação, essa classe recebe um objeto `List` com todos os comportamentos do robô. A partir disso, para acessar um comportamento específico, o índice da lista é utilizado e assim se obtém a referência para atuadores, dados ou métodos que pertencem ao contexto do comportamento acessado. Para facilitar as tarefas de adaptação um conjunto com métodos mais utilizados pelo loop de controle pode ser implementado seguindo o padrão de projetos *Facade* (GAMMA *et al.*,1995). Esse conjunto de métodos mais utilizados pode incluir tanto os métodos para controle dos atuadores como métodos para controlar o comportamento em questão. No caso desse loop de controle um método `steer()` implementado em `RemoteFacadeApp` acessa o comportamento por meio do índice zero da lista e aplica o valor de correção diretamente nos atuadores.

```
public class RemoteFacadeApp extends RemoteFacade{
...
public void steer(int amount){
	super.getBehaviors().get(0).getActuators().getMyNavigator().getPilot().steer
(amount);
}
```

Código 7.6: Classe RemoteFacadeApp.

Todo esse ciclo de funcionamento do sistema de loop de controle é gerenciado pelo componente `ControlLoop`. Esse componente é uma classe abstrata da arquitetura de referência e também é uma `Thread,e` o procedimento de execução do ciclo é implementado no método `run()`. É importante observar que a partir do `Monitor` o ciclo tem continuidade somente se houver eventos e sintomas para serem processados.

```
public class ControlLoop {
...
@Override
public void run() {

	while (isOn()) {
		monitor.storeEvent();
		if (analyzer.getKnowledgeBase().isUpdateEvent()) {
```

```
                    analyzer.storeSymptom();
              }
              if (planner.getKnowledgeBase().isUpdateSymptom()) {
                    planner.execute();
              }
        }
}
```

Código 7.7: Classe ControlLoop.

A funcionalidade de piloto automático implementada pelo módulo de loop de controle cumpre em parte a tarefa, isto é, o robô não colide com a parede e nem se afasta muito dela. Mas o percurso efetuado pelo robô não é efetuado totalmente em linha reta mantendo os 20cm. Conforme é mostrado na Figura 7.7, o percurso é feito com pequenas variações de distância, isso mostra a atuação do loop de controle corrigindo a trajetória.

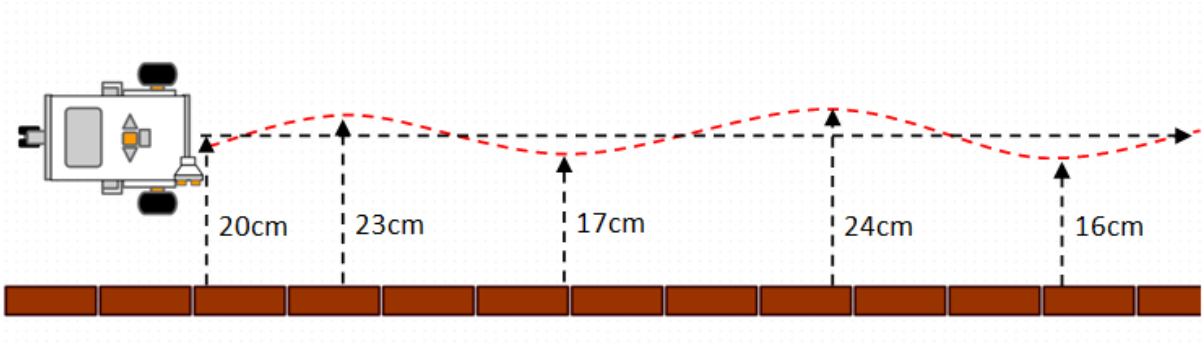

Figura 7.7: Trajetória do robô seguindo a parede com o loop de controle.

Esse problema ocorre porque embora o loop de controle atue corrigindo a trajetória do robô, o algoritmo de controle PID não está bem ajustado conforme as condições reais do ambiente e a velocidade atual do robô. O algoritmo PID possui três variáveis que devem ser ajustadas para se obter um melhor funcionamento, são elas: *Kp*, *Ki* e *Kd*, o ideal é que o ajuste seja feito em tempo de execução, analisando o histórico da trajetória por uma pequena fração de tempo e aplicando o ajuste em conformidade. Assim, um segundo loop de controle foi implementado, funcionando de acordo com o modelo MRAC *Model Reference Adaptive Control* (ASTROM e WITTENMARK, 1995), isso é, o sistema de loop de controle guarda um modelo de referência com uma sequência de valores colhidos da aplicação gerenciada numa pequena fração de tempo. Com isso, o segundo loop de controle atua sobre as variáveis de ajuste do algoritmo de controle de PID do primeiro loop controle realizando a adaptação em tempo de execução. A integração entre os dois loops é mostrada na Figura 7.8.

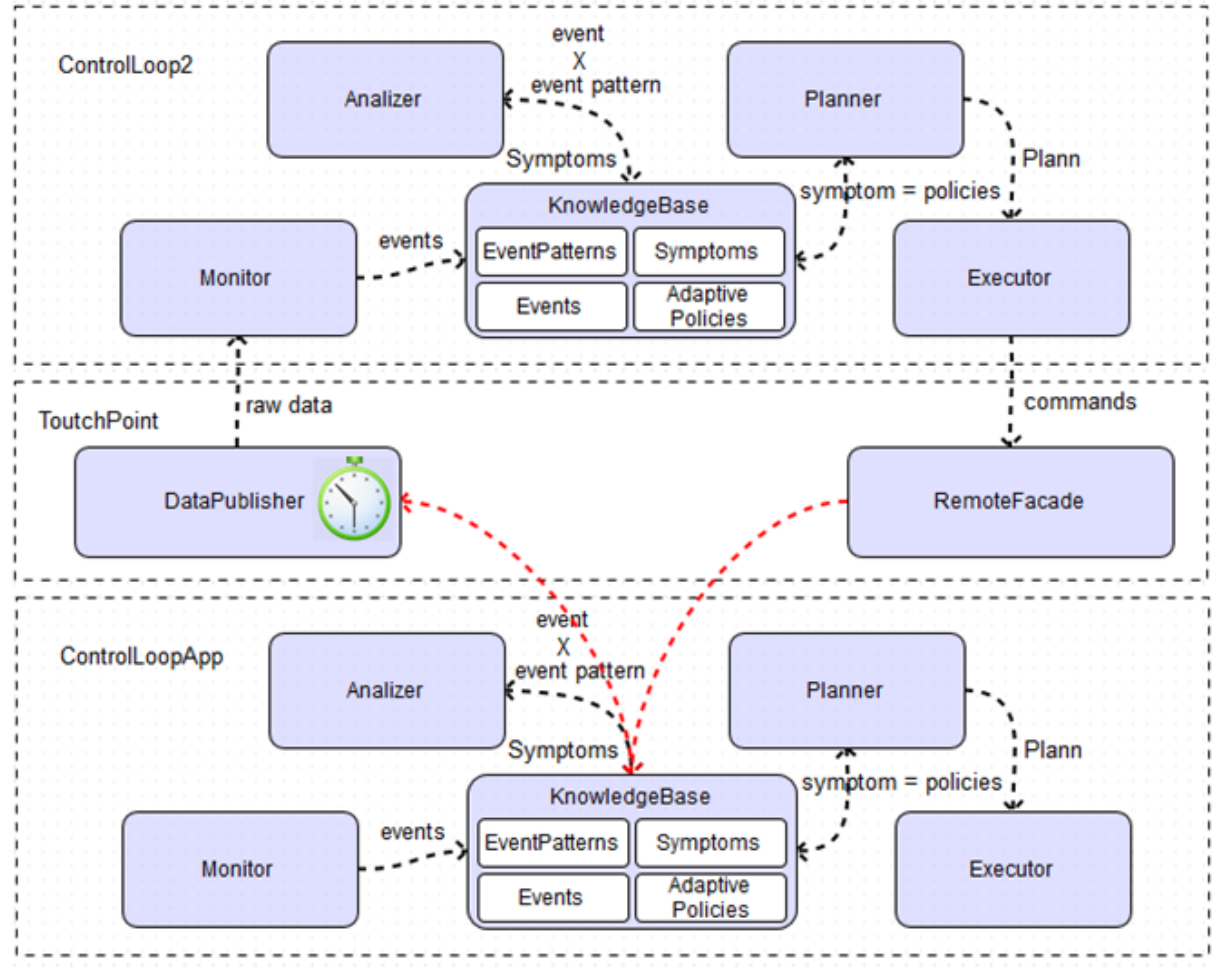

Figura 7.8: Dois loops de controle integrados.

É importante observar que a integração do `TouchPoint` acessando diretamente a base de conhecimento do primeiro loop propicia ao segundo loop, que está mais acima, o controle sobre os eventos, sintomas e políticas de adaptação do primeiro loop. Dessa forma é possível ter acesso total aos dados que estão sendo monitorados, e também possibilita alterações nas políticas de adaptação no primeiro loop. Essa abordagem permite que o comportamento do primeiro loop de controle seja alterado e, consequentemente seja ajustada a sua forma de atuação sobre a aplicação que controla o robô.

A diferença de funcionamento do segundo loop está na implementação do evento a ser monitorado e na função de análise. Como o segundo loop funciona de acordo com o modelo MRAC, ou seja, primeiro os dados são coletados durante uma fração de tempo para obter uma representação do que acontece no ambiente, nesse caso, é coletada uma série de fatores de desvio (diferença entre distância real e distância padrão) que representa a trajetória do robô num segmento durante uma determinada fração de tempo. Isso faz com que o tempo de resposta do segundo loop seja um pouco maior em relação ao primeiro loop.

A classe que representa o evento de interesse para o segundo loop de controle é implementada com um atributo do tipo `List,` de valores inteiros, que armazena uma série de valores de desvios no segmento. Após uma análise desse segmento um plano de adaptação e elaborado e uma ação corretiva é efetuada sobre os valores dos parâmetros *Kp*, *Ki* e *Kd* do algoritmo de PID.

Tabela 7.1: Dados da trajetória do robô em três segmentos

Segundos	**0s**	**3s**	**6s**	**9s**	**12s**	**15s**	**18s**	**21s**	**25s**
Segmento 1 Kp = 2 Ki = 1 Kd = 0	20	22	20	18	20	23	20	17	21
Desvio =	0	2	0	-2	0	3	0	-3	1
Segmento 2 Kp = 3 Ki = 1 Kd = 0,5	20	25	21	16	20	26	22	15	21
Desvio =	0	5	1	-4	0	6	2	-5	1
Segmento 3 Kp = 1 Ki = 0,5 Kd = 0	20	22	20	19	20	21	20	19	20
Desvio =	0	2	0	-1	0	1	0	-1	0

A Tabela 7.1 representa uma amostra dos dados em três segmentos distintos durante 25 segundos cada segmento. Os dados foram coletados observando os valores de distância diretamente no display do robô Lego durante o trajeto no segmento. Na primeira coluna são mostrados os valores utilizados nos parâmetros *Kp*, *Ki* e *Kd* do algoritmo de PID durante cada segmento. Na Figura 7.9 é mostrado um gráfico do desempenho do piloto automático do robô em cada um dos segmentos da trajetória do robô e os efeitos dos ajustes de autoadaptação.

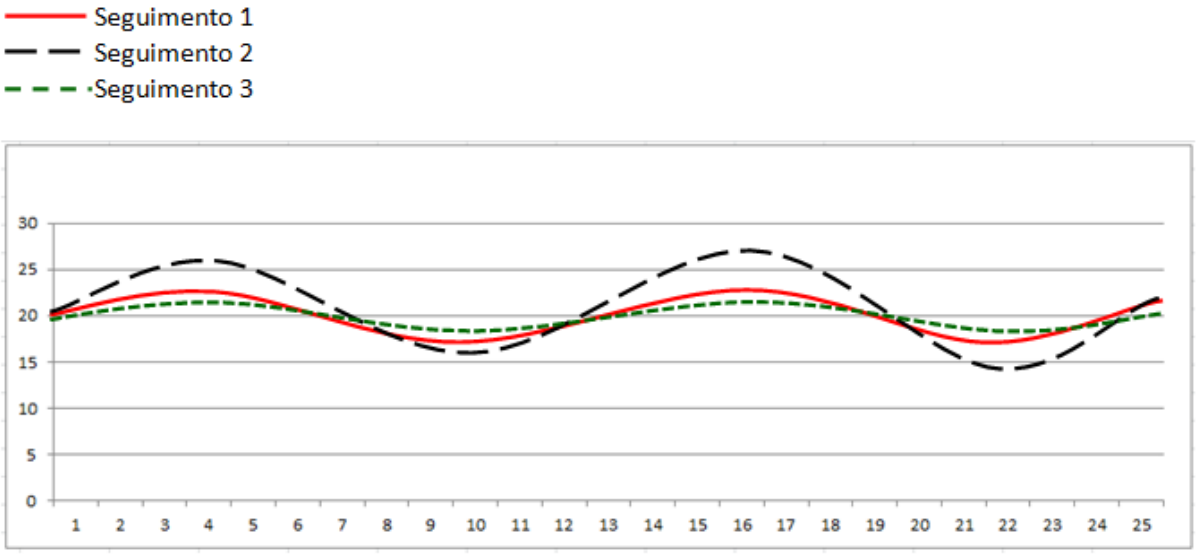

Figura 7.9: Desempenho do piloto automático.

A estratégia de adaptação para a finalidade desse experimento foi realizada em três fases conforme descrito a seguir:

- **Segmento 1**: Os parâmetros do PID estão com os valores *Kp* = 2, *Ki* = 1 e *Kd* = 0 que foram configurados na implementação, não houve ainda uma alteração em função da adaptação do segundo loop de controle. Essa série de dados de desvio é usada como entrada para "perceber" a trajetória do robô na primeira análise e do loop de controle.

- **Segmento 2**: Após 25 segundo e uma análise sobre a série de dados do desvio, o loop de controle realiza a primeira adaptação ajustando os valores do algoritmo de PID no primeiro loop para *Kp* = 3, *Ki* = 1 e *Kd* = 0,5. Depois de mais 25 segundos, uma nova análise é realizada sobre a segunda série de dados. Os dados coletados mostram um desempenho pior do que a primeira série, isso indica que a alteração piorou a trajetória do robô.

- **Segmento 3**: Uma nova adaptação é efetuada atribuindo os valores *Kp* = 1, *Ki* = 0,5 e *Kd* = 0 aos parâmetros do PID no primeiro loop. Depois de mais 25 segundos a trajetória do robô melhorou bastante, apesar de ainda não manter a distância sempre em 20 centímetros, mas a maior variação é de um centímetro, mostrando um desempenho bem melhor.

Este exemplo ilustrou uma experiência de desenvolvimento de uma aplicação de software para o domínio de robôs móveis autoadaptativos, utilizando como base para esse desenvolvimento, toda a estrutura de componentes da arquitetura de referência SARA^{MR} em conjunto com a API LeJOS NXJ e os recursos da linguagem Java. As alterações no código referentes à necessidade de implementação do segundo loop de controle será descrita no próximo capítulo. Não foi utilizado o módulo de representação do ambiente, esse módulo será utilizado no próximo exemplo.

7.3 Aplicação 2: Patrulhamento de Segurança

Este exemplo consiste em uma aplicação em que o robô realiza o patrulhamento de uma área em um ambiente fechado. O robô deve realizar uma sequência de tours nesse ambiente, completando um percurso padrão de forma regular, visitando sempre pontos conhecidos do ambiente. Cada ponto a ser visitado representa uma porta, e ao passar por esse ponto, o robô deve fazer uma leitura para verificar se a porta está aberta.

7.3.1 Implementação da parte Eletromecânica

A implementação dessa aplicação foi efetuada com o reúso do código referente ao robô implementado no exemplo anterior. As classes `MyRobot`, `MyPorts`, `MyUltrasonicSensor` e `MyTouchsensor` foram reaproveitadas. Nesse ponto, é importante destacar que um dos requisitos dessa aplicação foi a necessidade de gerenciar o consumo de bateria. Assim, foi necessário implementar uma classe chamada `MyBatery` que informa a quantidade de energia restante nas baterias do robô. Apesar de não ser um sensor físico, essa classe é uma oportunidade para demonstrar que para adicionar mais dispositivos basta implementar a classe e adicioná-la à coleção de sensores do robô. Assim, depois que ela estiver instanciada, estará disponível junto com os outros sensores do robô.

7.3.2 Implementação do Controle do Robô

O controle do robô é composto por um conjunto de classes que representam comportamentos com prioridade definida. Todas as classes estendem a classe `MyBehavior` e implementa a classe `Behavior`. A classe `Patroller` é implementada para o comportamento

é responsável pela locomoção do robô pelo caminho planejado no tour, parando em frente aos pontos que representam cada porta no percurso. A classe `CheckOpenDoor` é implementada para o comportamento que deverá verificar se a porta está aberta a cada parada durante o tour. O robô inicia em um ponto denominado como "base", realiza o tour completo até retornar a base. A classe `ReturnToBase` é implementada um comportamento para conduzir o robô de volta até a base caso os níveis de energia estejam baixos, impossibilitando a continuação do *tour*. Esse comportamento recebe um `Path`, que é o caminho mais curto entre a localização do robô e a base.

7.3.3 Implementação da Representação do Ambiente

Para essa aplicação em particular, a forma de controle do robô é feita de forma mista, deliberativa e reativa, ou seja, o robô possui uma representação interna do ambiente que é usada como um mapa pelo robô para conhecer o ambiente. O mapa possui uma grade, como em um plano cartesiano, com coordenadas X e Y, e os elementos do ambiente são demarcados na grade de ocupação. A locomoção é feita com base em pontos de referência que são coordenadas no plano cartesiano e, também na leitura de informações oriundas de sensores e atuadores.

O controle do robô consegue estimar o deslocamento pela grade de ocupação utilizando um sistema de odometria. As informações odométricas são obtidas dos atuadores que estão acoplados às rodas do robô, e à partir do movimento das rodas, é possível calcular o deslocamento. A classe `MyNavigator` utiliza a classe `Navigator` da API LeJOS NXT que possui a implementação d sistema de odometria e é responsável pela correta "navegação" pelo ambiente.

Na Figura 7.10 é mostrado o mapa do ambiente implementado para este estudo de caso.

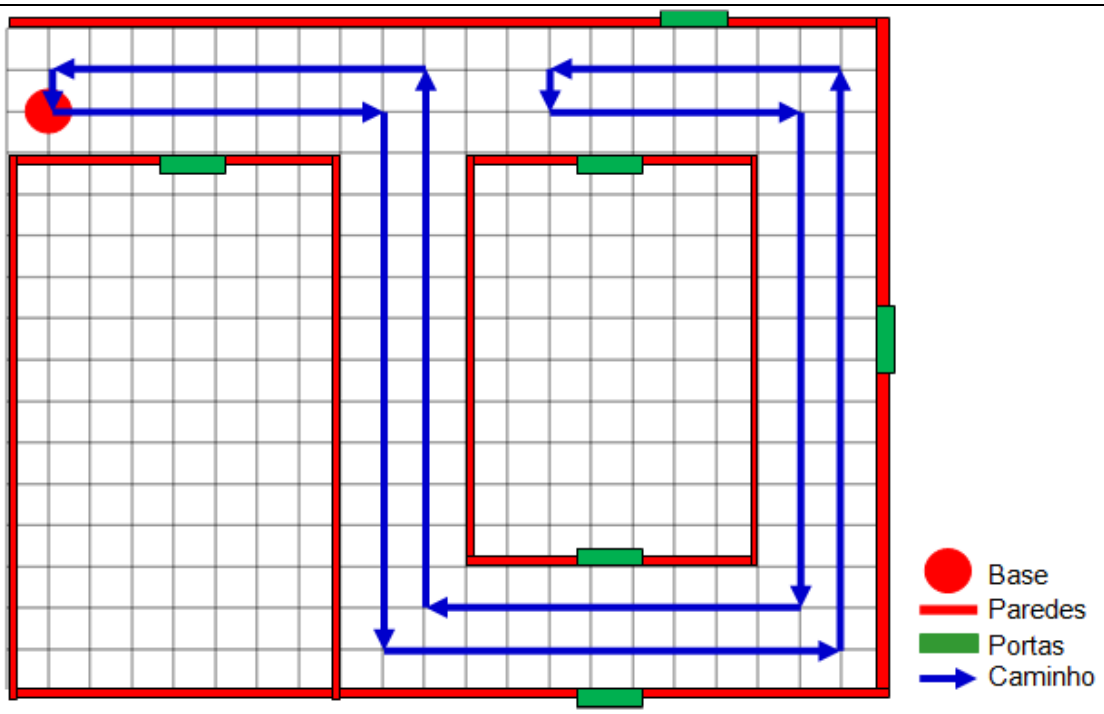

Figura 7.10: Mapa de ocupação do ambiente.

O bloco de código do método `configureEnvironment()` é um exemplo de como foi implementada a representação do ambiente utilizando a abordagem de grade de ocupação.

```
public EnvironmentMap configureEnvironment() {

	// Map Frontier( x, y, width, height)
	Rectangle bounds = new Rectangle(0, 0, 160, 210);

	// Map all walls in the closed environment
	// to delimit the corridors
	// Line (x1, y1, x2, y2)
	Line[] lines = new Line[6];
	lines[0] = new Line(30, 0, 30, 80);
	lines[1] = new Line(30, 80, 160, 80);
	lines[2] = new Line(30, 110, 30, 180);
	lines[3] = new Line(30, 110, 130, 110);
	lines[4] = new Line(30, 180, 130, 180);
	lines[5] = new Line(130, 110, 130, 180);

	// Map by grid when start and finish all doors in all corridors
	// numbering doors by follow base path.
	List<Door> doors = new ArrayList<Door>();
	doors.add(new Door(new Waypoint(30,40), new Waypoint(30,50)));
	doors.add(new Door(new Waypoint(160,140), new Waypoint(160,150)));
	doors.add(new Door(new Waypoint(80,210), new Waypoint(70,210)));
	doors.add(new Door(new Waypoint(0,170), new Waypoint(0,160)));
	doors.add(new Door(new Waypoint(30,140), new Waypoint(30,150)));
	doors.add(new Door(new Waypoint(130,150), new Waypoint(130,140)));

	return new EnvironmentMap(bounds, lines, doors);
}
```

Código 7.8: Classe EnvironmentMap.

O bloco de código do método `configureStopWaypoints()` é um exemplo de como foi implementada a localização das portas que devem ser monitoradas no ambiente.

```
public static List<Door> configureStopWayPoints(){

	List<Door> stopWayPoints = new ArrayList<Door>();

	//Door 1
	stopWayPoints.get(0).setDoorIni((new Waypoint(20,40)));
	stopWayPoints.get(0).setDoorEnd((new Waypoint(20,50)));
	stopWayPoints.get(0).setDoorNumber(1);

	//Door 2
	stopWayPoints.get(1).setDoorIni((new Waypoint(150,140)));
	stopWayPoints.get(1).setDoorEnd((new Waypoint(150,150)));
	stopWayPoints.get(0).setDoorNumber(2);

	//Door 3
	stopWayPoints.get(2).setDoorIni((new Waypoint(80,200)));
	stopWayPoints.get(2).setDoorEnd((new Waypoint(70,200)));
	stopWayPoints.get(0).setDoorNumber(3);

	//Door 4
	stopWayPoints.get(3).setDoorIni((new Waypoint(10,170)));
	stopWayPoints.get(3).setDoorEnd((new Waypoint(10,160)));
	stopWayPoints.get(0).setDoorNumber(4);

	//Door 5
	stopWayPoints.get(4).setDoorIni((new Waypoint(20,140)));
	stopWayPoints.get(4).setDoorEnd((new Waypoint(20,150)));
	stopWayPoints.get(0).setDoorNumber(5);

	//Door 6
	stopWayPoints.get(5).setDoorIni((new Waypoint(140,150)));
	stopWayPoints.get(5).setDoorEnd((new Waypoint(140,140)));
	stopWayPoints.get(0).setDoorNumber(6);

	return stopWayPoints;
}
```

Código 7.9: Classe EnvironmentMap.

7.3.4 Implementação do Loop de Controle

Nessa aplicação, houve a necessidade de monitorar a a quantidade de energia restante nas baterias do robô durante as atividades de patrulhamento de segurança. Uma opção seria a implementação de um *loop* de controle com essa finalidade, porém, a abordagem escolhida foi a implementação de um novo comportamento do robô quando o nível de carga assume um valor crítico. A capacidade de carga máxima do kit Lego NXT é 9,0 volts. O nível crítico de carga é 7,5 volts, por isso o valor do evento padrão foi ajustado para 8,0 volts.

Quando a carga da bateria atinge esse valor padrão, um algoritmo Dijkstra é utilizado para calcular o menor caminho entre a localização do robô e a base. O algoritmo produz um `Path` que é uma sequência de coordenadas que representa esse caminho. O caminho de retorno é atribuído ao comportamento que, ao assumir o controle, conduz o robô de volta a base.

7.4 Considerações Finais

Neste capítulo, foram apresentados dois exemplos de aplicações para robôs móveis autoadaptativos. Nos dois exemplos, foram experimentadas a implementação de diferentes funcionalidades do domínio desse tipo de aplicação. A arquitetura de referência SARA^{MR} foi utilizada e cumpriu com sucesso ao propósito pelo qual foi projetada. Os quatro cenários de evolução identificados neste trabalho de mestrado ocorrem com frequência e a arquitetura de referência aqui proposta mostrou-se como uma forma eficiente para produzir software para o domínio de RMAs.

Capítulo 8

ANÁLISE PRELIMINAR DO ESFORÇO DE MANUTENÇÃO

8.1 Considerações Iniciais

A arquitetura de referência SARA^{MR} foi estabelecida com o objetivo de fornecer uma estrutura que faça com que os sistemas desenvolvidos com o seu apoio sejam mais manuteníveis. O emprego de SARA^{MR} faz com que a arquitetura geral de todo o sistema seja mais inteligível em razão da modularidade de sua estrutura.

No capítulo anterior foram desenvolvidas duas aplicações para robôs móveis tendo como apoio a arquitetura de referência SARA^{MR}. O objetivo deste capítulo é fornecer evidências de que os sistemas para RMA desenvolvidos com o seu apoio são mais fáceis de serem evoluídos/mantidos do que sistemas que não seguem essa arquitetura. Para isso, são usadas, ao longo deste capítulo, as duas aplicações desenvolvidas e mostradas no capítulo anterior. A seguir, são apresentadas as características iniciais das duas aplicações, implementadas na primeira versão, ou seja, antes das atividades de manutenção.

A aplicação para o robô seguidor de parede teve as seguintes características:

- Na parte eletromecânica: dois sensores, ultrassônico e toque;
- Na parte de controle: um comportamento para a locomoção do robô;
- Na parte de representação do ambiente não foi necessário a utilização;
- Na parte de autoadaptação: um loop de controle com a função de piloto automático.

A aplicação de patrulhamento de segurança teve as seguintes características:

- Na parte eletromecânica: dois sensores, ultrassônico e toque;
- Na parte de controle: dois comportamentos, patrulhamento, verificação de portas abertas;
- Na parte de representação do ambiente foi necessário configurar os corredores, portas e a fronteira do ambiente fechado;
- Na parte de autoadaptação, não foi implementado loop de controle.

8.2 Evoluções Realizadas

A abordagem escolhida para essa avaliação baseia-se na demonstração de atividades de manutenção no código necessárias para evolução das duas aplicações robóticas. Para a aplicação do robô seguidor de paredes, realizou-se a evolução mostrada no Quadro 1 e para a aplicação de patrulhamento realizou-se as evoluções do Quadro 2.

Quadro 1: Evolução da Aplicação Seguidora de Paredes

Evolução	Descrição da Necessidade de Evolução
Inclusão de um novo loop de controle	Em consequência do mau desempenho do robô, optou-se por incluir um novo loop de controle para obter melhores níveis de desempenho.

Quadro 2: Evoluções da Aplicação de Patrulhamento

Evolução	Descrição da Necessidade de Evolução
Inclusão de um Componente de Gerenciamento de Carga de Bateria	Como havia o risco do robô ficar sem bateria no meio de um percurso, houve a necessidade de incluir um novo componente para monitorar a carga da bateria e uma alteração de comportamento para fazê-lo retornar para a base.
Inclusão de um novo sensor de cores	Necessitou-se diferenciar a prioridades das portas. Optou-se por fazer isso instalando etiquetas de duas cores nos batentes. Etiqueta na cor verde para portas de média prioridade e de cor vermelha para alta prioridade.

8.3 Manutenção no Robô Seguidor de Paredes

A aplicação do robô seguidor de paredes apresentou um mau desempenho na funcionalidade de piloto automático. Para resolver esse problema um segundo loop de controle foi implementado atuando sobre os valores dos parâmetros de ajuste do algoritmo de controle de PID do primeiro loop controle. Com isso foi estabelecida a funcionalidade de autoadaptação em tempo de execução.

A integração entre os dois loops é realizada por meio de um segundo módulo `TouchPoint`, composto de um `DataPublisher` e um `RemoteFacede`.

Nos trechos de código a seguir são mostradas as implementações das duas classes, `DataPublisherAdjustLoop` e `RemoteAdjustLoop`, do módulos de integração do segundo loop.

```
public class DataPublisherAdjustLoop extends DataPublisher {

	public DataPublisherAdjustLoop(List<MyBehavior> behaviors, KnowledgeBase
knowledgeBase, long ini, long rate) {
		super(behaviors, knowledgeBase, ini, rate);
	}

	@Override
	public void run() {

		if (getDeviation() > 0){
			if (super.getIntValues().size() > 10) {
				super.getIntValues().clear();
			} else {
				super.addIntValue(getDeviation());
			}
		}
	}

	public int getDeviation(){
		if (((KnowledgeBaseDistance)
super.getKnowledgeBase().get()).isUpdateSymptom()){
			return ((SymptomDistance) ((KnowledgeBaseDistance)
super.getKnowledgeBase().get()).getSymptom()).getDeviation();
		} else {
		  return 0;
		}
	}
...
```

Código 8.1: Classe DataPublisherAdjustLoop.

```
public class RemoteAdjustLoop extends RemoteFacade {

	public RemoteAdjustLoop(List<MyBehavior> behaviors, KnowledgeBase
```

```
knowledgeBase) {
		super(behaviors, knowledgeBase);
	}

	public void setKp(float kp){
		((AdaptivePoliceDistance)
super.getKnowledgeBase().getAdaptivePolice()).getPID().setKp(kp);
	}

	public void setKi(float ki){
		((AdaptivePoliceDistance)
super.getKnowledgeBase().getAdaptivePolice()).getPID().setKi(ki);
	}

	public void setKd(float kd){
		((AdaptivePoliceDistance)
super.getKnowledgeBase().getAdaptivePolice()).getPID().setKd(kd);
	}
...
```

Código 8.2: Classe RemoteAdjustLoop.

Como pode ser visto nos códigos 8.1 e 8.2, os construtores das duas classes recebem, como parâmetro, objetos das classes `MyBehavior` e `KnowledgeBase`, assim o módulo de integração tem acesso aos valores dos parâmetros do algoritmos de PID do primeiro Loop. Essa atividade de manutenção foi realizada com a implementação das classes do módulo de integração e do módulo de loop de controle. O esforço necessário para se realizar as alterações foi aplicado nas atividades de manutenção no código e não em atividades de rastreamento de código. O módulo `TouchPoint` fornece uma camada de integração que elimina a dependência direta entre os dois loops de controle.

A implementação do segundo loop de controle foi realizada a partir da concretização das classes abstratas que constituem o módulo de loop de controle. A seguir são apresentados os códigos das classes do segundo loop.

```
Public class MonitorAdjustLoop extends Monitor {

	public MonitorAdjustLoop(KnowledgeBase knowledgeBase, DataPublisher
publisher) {
		super(knowledgeBase, publisher);
	}

	@Override
	public Event filter() {
		EventAdjustLoop event = new EventAdjustLoop(2, "deviation");
		if(super.getPublisher().getIntValues().size() == 10){
		event.setListDeviation(super.getPublisher().getIntValues());
		return event;
	}
}
```

Código 8.3: Classe MonitorAdjustLoop.

No código 8.3 é mostrada a classe `MonitorAdjustLoop`, observa-se que essa classe possui código de implementação apenas no método `filter`(). Este método faz a coleta de dados publicados por uma classe `Publisher` pertencente ao módulo de integração.

```
public class AnalyzerAdjustLoop extends Analyzer{

	public AnalyzerAdjustLoop(KnowledgeBase knowledgeBase) {
		super(knowledgeBase);
	}

	@Override
	public Symptom compare() {
		SymptomAdjustLoop symptomAdjustLoop = new SymptomAdjustLoop(3,
"Average Desviation");
		int averageDeviation = 0;
		int sumValues = 0;
		List<Integer> listDeviation = ((EventAdjustLoop)
((KnowledgeBaseAdjustLoop)
super.getKnowledgeBase().get()).getEvent().get()).getListDeviation();

			if (listDeviation.size() > 10){
				for(int x = 1; x < listDeviation.size(); x++){
					sumValues += listDeviation.get(x);
				}
			}
		averageDeviation = (sumValues / 10);
		symptomAdjustLoop.setAverageDesviation(averageDeviation);
		return symptomAdjustLoop;
	}
}
```

Código 8.4: Classe AnalyzerAdjustLoop.

No código 8.4 é mostrada a classe `AnalyzerAdjustLoop`, esta classe possui código de implementação apenas no método `compare`(). Este método realiza a atividade de comparação.

```
public class PlannerAdjustLoop extends Planner{

	private int averageDeviation;
	private List<Float> kValues;

	public PlannerAdjustLoop(KnowledgeBase knowledgeBase, Executor executor) {
		super(knowledgeBase, executor);
	}

	@Override
	public void interpret() {
		averageDeviation = ((SymptomAdjustLoop)
((KnowledgeBaseAdjustLoop)
super.getKnowledgeBase().get()).getSymptom()).getAverageDesviation();
		if(averageDeviation > 5){
			kValues.add(3.0f);
```

```
                    kValues.add(0.5f);
                    kValues.add(0.5f);
              }

              if(averageDeviation < 5){
                    kValues.add(3.0f);
                    kValues.add(0.0f);
                    kValues.add(0.0f);
              }
        }

        @Override
        public void devisePlan() {
              ((ExecutorAdjustLoop)
super.getExecutor().get()).setkValues(kValues);
        }
}
```

Código 8.5: Classe PlannerAdjustLoop.

No código 8.5 é mostrada a classe `PlannerAdjustLoop`, esta classe possui código de implementação apenas nos métodos `devisePlan`() e `interpret`(). Estes dois métodos realizam a atividade de decisão e preparação do plano corretivo.

```
public class ExecutorAdjustLoop extends Executor{

        private int steer;

        public ExecutorAdjustLoop(RemoteFacade remoteFacade) {
              super(remoteFacade);
              kValues = new ArrayList<Float>();

        }

        @Override
        public void applyChange() {
              ((RemoteAdjustLoop) super.getRemoteFacade().get()).
setKp(kValues.get(0));
              ((RemoteAdjustLoop) super.getRemoteFacade().get()).
setKi(kValues.get(1));
              ((RemoteAdjustLoop) super.getRemoteFacade().get()).
setKd(kValues.get(2));
        }
...gets e sets
```

Código 8.6: Classe ExecutorAdjustLoop.

No código 8.6 é mostrada a classe `ExecutorAdjustLoop`, esta classe possui código de implementação apenas no método `applyChange`(). Este método aplica o plano corretivo na aplicação base do robô por intermédio da classe `RemoteFacade` pertencente ao módulo de integração.

```
public class KnowledgeBaseAdjustLoop extends KnowledgeBase{

        @Override
```

```
	public KnowledgeBaseAdjustLoop get() {
		return this;
	}
}
```

Código 8.7: Classe KnowledgeBaseAdjustLoop.

No código 8.7 é mostrada a classe `KnowLedgeBaseAdjustLoop`, para essa classe não foi necessário implementar métodos. Esta classes armazena eventos, sintomas e políticas de adaptação.

```
public class EventAdjustLoop extends Event{

	private List<Integer> listDeviation;
	private int distance;
	private int position;

	public EventAdjustLoop(int id, String description) {
		super(id, description);
		listDeviation = new ArrayList<Integer>();
	}

	@Override
	public EventAdjustLoop get() {
		return this;
	}
...gets e sets
```

Código 8.8: Classe EventAdjustLoop.

No código 8.8 é mostrada a classe `EventAdjustLoop`, para esta classe foram criados três atributos e os métodos `get()`. Esta classe armazena valores que representam eventos monitorados à partir da aplicação base.

```
public class SymptomAdjustLoop extends Symptom{

	private int averageDesviation;

	public SymptomAdjustLoop(int id, String description) {
		super(id, description);
	}

	@Override
	public SymptomAdjustLoop get() {
		return this;
	}
...gets e sets
```

Código 8.9: Classe SymptomAdjustLoop.

No código 8.9 é mostrada a classe `SymptomAdjustLoop`, para essa classe foi criado um atributo e o método `get()`. Esta classe armazena sintomas previamente definidos de acordo com regras de adaptação.

```
public class AdaptivePoliceAdjustLoop extends AdaptivePolice{

	@Override
	public AdaptivePoliceAdjustLoop get() {
		return this;
	}
}
```

Código 8.10: Classe AdaptivePoliceAdjustLoop.

No código 8.10 é mostrada a classe `AdaptivePoliceAdjustLoop`, para essa classe não foi necessário implementar métodos. Esta classe armazena políticas de adaptação previamente definidas de acordo com regras de adaptação.

Cada classe que compõe o ciclo de adaptação do loop de controle possui uma funcionalidade bem específica, todas com apenas um método a ser implementado, com exceção da classe `Planner` que possui dois métodos. O conjunto de classes estabelece um protocolo de comunicação instituindo o fluxo de dados no ciclo do loop de controle. Cada classe tem seu papel e responsabilidades bem definidos para a funcionalidade de loop de controle. Assim, os pontos onde existe a ocorrência de impactos de modificação são apontados sistematicamente pela arquitetura de referência. As atividades de manutenção são facilitadas, pois é mais fácil de resolver um problema complexo quando ele é dividido em partes menores e administráveis.

Nas aplicações desenvolvidas de forma tradicional, geralmente as funcionalidades dos loops de controle estão espalhadas internamente junto ao código da aplicação base do robô (BRUN *et al*. 2009, CHENG *et al.* 2009). Nesse caso, o impacto de alterações nos loops de controle pode gerar modificações em outros módulos do sistema dificultando as atividades de manutenção evolutiva.

Na arquitetura de referência SARA^{RM} os loops de controle são projetados como módulos externos à aplicação base, e para cada módulo de loop implementado é necessária a implementação de um módulo de integração. Essa solução estrutural, de tornar os loops de controle explícitos, permite ao arquiteto de software separar as funcionalidades básicas do sistema das preocupações com a autoadaptação.

Uma particularidade interessante de se possuir loops de controle mais bem modularizados e implementados de forma separada do código base, é que torna-se possível ativá-los ou desativá-los conforme necessário. No caso de uma aplicação em que os loops de controle estão implementados de forma espalhada junto ao código da aplicação base, desabilitar loops de controle pode ser uma atividade complexa.

Quadro 3: Resumo das atividades de manutenção no robô seguidor de paredes.

Tipo de atividade	Descrição	Quantidade
Criação de classes	Foram criadas duas classes para o módulo de integração e oito classes para o loop de controle.	10
Implementação de métodos	Foram implementados cinco métodos nas classes do módulo de integração e cinco métodos para as classes do loop de controle.	10
Declaração de atributos	Foram declarados sete atributos para as classes do loop de controle.	7

No Quadro 3 é apresentado um resumo com os tipos e a quantidade de atividades de manutenção realizadas. Os pontos importantes a serem observados e que apresentam vantagens no aspecto de manutenção evolutiva nos loops de controle são:

- O mantenedor do código pode identificar os componentes do loop de controle na aplicação de forma mais direta do que quando isso é feito de forma tradicional, já que esses componentes estão representados como entidades de primeira classe, isto é, classes;
- As alterações/substituições nesses componentes podem ser feitas sem causar impacto nos outros componentes do loop de controle. Por exemplo, substituir um analisador por outro sem causar impacto em outros componentes do loop;
- A arquitetura emprega a estratégia de políticas de adaptação. Quando se trata apenas de alterações na lógica das adaptações existentes, basta alterar os comandos que compõem essas políticas de adaptação;
- A inclusão, remoção ou alteração de loops de controle não causam impacto de manutenção no restante da aplicação base do robô. Isso é possível porque o módulo `TouchPoint` provê uma camada de integração que elimina as dependências entre os dois módulos. Essas atividades são facilitadas pela arquitetura, independentemente se o loop de controle é acoplado à aplicação base do robô ou acoplado a um outro loop de controle.

Dessa forma, nota-se que o esforço necessário para se realizar as alterações é direcionado apenas aos elementos que realmente estão envolvidos naquela alteração e não com atividades de rastreamento e correção de código em locais conceitualmente desvinculados desses elementos. Com base em nossas observações, estudos e análise de trabalhos relacionados, aplicações que não usam a arquitetura SARA^{MR}, possivelmente requererem um esforço de manutenção maior. Com isso, a criação ou a manutenção de loops de controle podem ser atividades difíceis de serem realizadas, dependendo da complexidade do problema de autoadaptação.

8.4 Manutenção no Robô de Patrulhamento

A aplicação do robô de patrulhamento apresentou novos requisitos referentes ao monitoramento do nível de carga das baterias e a identificação do grau de prioridade de segurança para as portas do ambiente. Essa necessidade disparou novas atividades de evolução mediante a instalação um componente para gerenciamento de carga das baterias e a instalação de um sensor de cores RGB.

Em um sistema desenvolvido para RMAs, normalmente engenheiros de software instanciam as classes que representam sensores e atuadores em vários locais do código. Na arquitetura de referência SARA^{MR}, os sensores e atuadores são instanciados uma única vez, em apenas um local, e adicionados a um conjunto objetos desses dispositivos que estão disponíveis a todos os comportamentos de controle do robô. Essa solução estrutural da arquitetura procura evitar que classes que representam sensores e atuadores sejam instanciadas várias vezes em diversos componentes, e espalhados por toda a aplicação. Essa abordagem evita gastos desnecessários com memória, evita rastreamento de código em busca desses locais de instanciação e fornece uma maneira padronizada de desenvolvimento.

Outra decisão de projeto tomada foi criar "classes espelho" de classes da API LeJOS. Essa estratégia faz com que o código da aplicação desenvolvida fique bem menos dependente da nomenclatura de classes da API. A independência ocorre porque todos os locais do código que precisarem instanciar/referenciar uma determinada classe irão ficar dependentes apenas da nova classe espelho criada, e não mais de uma classe de API. O resultado concreto disso é que, caso a classe de API seja substituída, apenas um local do código da aplicação será

impactado. Diferentemente de situações em que a própria classe da API é instanciada diretamente em vários locais.

A necessidade de gerenciamento de energia e identificação do grau de prioridade de segurança disparou a criação de dois novos sensores na aplicação. A seguir são apresentadas as atividades a serem efetuadas neste cenário de evolução/manutenção.

O engenheiro de aplicação deve criar uma classe espelho que representa o sensor, fazê-la estender a classe Sensor pertencente à arquitetura e instanciar o tipo de correto de sensor no construtor da classe. Nesse caso, foi criada uma classe para representar o sensor de cor RGB (Red, Green, Blue). No trecho de código a seguir, é mostrada a estratégia adotada para representar a existência de um sensor. Note-se que a classe de API que representa o sensor de cor RGB é a `ColorSensor`, que está sendo instanciada dentro do construtor da `MyColorSensor`. Caso o nome da classe da API seja modificado, o único local que será necessário mudar é dentro desse construtor.

```
import lejos.nxt.SensorPort;
import lejos.nxt.ColorSensor;
...
public class MyColorSensor extends Sensor{

	private ColorSensor cs;

	public MyColorSensor () {
		cs = new ColorSensor(SensorPort.S4);
	}

	public Color getColor() {
		return cs.getColor();
	}

	@Override
	public MyColorSensor get() {
		return this;
	}

gets e sets...
```

Código 8.11: Classe MyColorSensor.

Como pode ser visto no trecho de código 8.11, a inclusão de um novo sensor é efetuada em um único local, ou seja, na classe `MyColorSensor`. Sempre que for necessário incluir um sensor, basta criar uma classe que estende a classe `Sensor` e sobrescrever o método `get()`.

No trecho de código 8.12, é mostrada a nova classe `MyBatery` implementada da mesma forma que o sensor de cor RGB mostrado anteriormente. Após a implementação, a classe é instanciada em um único local do código e o objeto é incluído na coleção de sensores do robô.

```
import lejos.nxt.Battery;

public class MyBattery extends Sensor{

	private Battery battery;

	construtor...

	public float getVoltage() {
		return battery.getVoltage();
	}

	public MyBattery get(){
		return this;
	}
```

Código 8.12: Classe MyBatery.

As classes que estendem `MyBehavior`, para a implementação dos comportamentos do robô, e também as classes que estendem `RemoteFacade` do módulo de integração, se beneficiam com a estratégia de criação de "classe espelho" quando um sensor precisa ser substituído por outro, pois essa estratégia elimina as dependências com a nomenclatura das classes da API LeJOS. No caso de inclusão ou exclusão de sensores, a modularidade da arquitetura de referência facilita a localização do código que deve ser alterado.

Nas aplicações que são desenvolvidas sem a utilização da arquitetura SARA^{MR} e que não levam em conta uma estratégia única de acesso à leitura dos dados dos sensores, é necessário identificar todas as classes que necessitam ler os valores dos sensores e espalhar referências por todos esses pontos de acesso. Em uma eventual necessidade de manutenção, um rastreamento de código deverá ser realizado e todos esses pontos deverão ser novamente identificados para se proceder com as modificações necessárias.

A seguir, é apresentado um resumo dos impactos de evolução com a inclusão dos sensores.

- A inclusão de novos sensores e atuadores é efetuada em um único local do código fonte do sistema. O acesso ao dispositivo eletromecânico incluído estará disponível a todos os módulos por meio da coleção de sensores.

- A estratégia de utilização de classes espelho evita que as alterações nos sensores e atuadores causem impacto no código fonte das classes já existentes e que realizam leitura de sensores e enviam comandos para os atuadores.

- A exclusão de sensores e atuadores causam impactos de manutenção no código fonte existente da aplicação, porém os pontos de ocorrência são mais facilmente identificados pois em geral, encontram-se dentro dos comportamentos.

A evolução da aplicação do robô de patrulhamento referente ao gerenciamento de energia disparou a necessidade de inclusão de um novo comportamento com a função de retornar o robô a base quando o nível de carga na bateria atingir um ponto crítico.

Geralmente, os sistemas desenvolvidos para RMAs possuem extensos blocos de código que agrupam um grande número de funções que implementam os comportamentos do robô. Essa abordagem aumenta a dependência entre os componentes de controle e os componentes da parte eletromecânica do robô causando grande impacto de modificações e dificultando a manutenção evolutiva no cenário de inclusão e remoção e alteração de comportamentos.

Na arquitetura de referência SARA^{MR}, o controle do robô é realizado por meio de comportamentos em classes distintas, com funções bem definidas que devem ser coordenadas para alcançar um objetivo desejado. A solução estrutural da arquitetura minimiza o impacto de alterações nos comportamentos do robô. A inclusão de comportamentos é realizada com a criação de uma nova classe que estende a classe abstrata `MyBehavior` e implementa a interface `Behavior` da API LeJOS NXJ. No código a seguir é mostrada a implementação do comportamento `ReturnToBase`.

```
public class ReturnToBase extends MyBehavior implements Behavior{

	private boolean suppressed;
	private Route route;

	public ReturnToBase(MySensors sensors, MyActuators actuators, Route route){
		super(sensors, actuators);
	}

	@Override
	public boolean takeControl() {
		return (super.getSensors().getElement("BT").getVoltage() <= 8.5);
	}
```

```
      @Override
      public void action() {
                        super.getActuators().getMyNavigator().getNavigator().
setPath(route.getPath());
                        super.getActuators().getMyNavigator().getNavigator().
followPath();
                }
      }
...gets e sets
```

Código 8.13: Classe ReturnToBase.

Como pode ser visto no trecho de código 8.13 os comportamentos recebem uma referência para os sensores e atuadores instanciados. O método `takeControl()` define quando esse comportamento assume o controle, no caso, quando a carga da bateria for menor ou igual a 8.5 volts. No método `action`(). o caminho de retorno é atribuído e o navegador segue até o local determinado como base.

No Quadro 4 é apresentado um resumo com os tipos e a quantidade de atividades de manutenção realizadas.

Quadro 4: Resumo das atividades de manutenção no robô de patrulhamento.

Tipo de atividade	Descrição	Quantidade
Criação de classes	Foram criadas duas classes para inclusão de novos sensores e uma classe para inclusão de novo comportamento.	3
Implementação de métodos	Foram implementados dois métodos nas classes dos sensores e dois métodos para a classe do novo comportamento.	4
Declaração de atributos	Foram declarados dois atributos para as classes dos sensores e dois para o novo comportamento.	4

Os pontos nos quais existem a ocorrência desses impactos de modificação são apontados pela arquitetura de referência. O esforço necessário para se realizar as alterações é gasto com as atividades de manutenção no código e não com atividades de rastreamento de código. Numa eventual necessidade de manutenção, as aplicações que são desenvolvidas sem a utilização da arquitetura SARA^{MR} e que não levam em conta uma estratégia baseada em blocos de comportamentos, é necessário identificar todas as classes que possuem código implementando comportamentos para as modificações necessárias.

A seguir, é apresentado um resumo dos impactos de evolução com a inclusão de comportamentos.

- A inclusão de novos comportamentos não causa impacto no código fonte da aplicação base do robô, entretanto, pode causar impacto de manutenção nas classes estendem `MyBehavior`, mas apenas no método `takeControl()` que é utilizado no gerenciamento de comportamentos da aplicação base. Tipicamente, a inclusão de comportamento não causa impacto de manutenção nos módulos de integração e loops de controle existentes, exceto se houver a necessidade, por parte desses loops, de gerenciar a ativação ou desativação do comportamento incluído, porém essa atividade é facilitada pela arquitetura de referência. No caso de aplicações que não utilizam a arquitetura SARA MR, e não utiliza a estratégia baseada em comportamentos, as atividades de inclusão de comportamentos pode ser complexa.

- A exclusão não causa impacto no código fonte da aplicação base do robô, porém pode haver impacto nas classes `DataPublisher` e `RemoteFacade` do módulo de integração, mas somente se houver loops monitorando esses comportamentos.

- A alteração nos comportamentos não causa impacto no código fonte da aplicação base do robô, e geralmente não causa impacto nos módulos de integração ou loops de controle existentes.

Com a realização das atividades de manutenção ficou evidente que o mantenedor do código pode identificar facilmente os componentes na aplicação e os pontos de alteração. O esforço necessário é gasto com as atividades de manutenção no código e não com atividades de rastreamento de código. A estratégia de utilização de classes espelho e políticas de adaptação facilita muito no cenário de exclusão de sensores.

Com o objetivo de avaliar o esforço gasto com as atividades de manutenção evolutiva, foi realizada uma avaliação do impacto que as alterações de códigos efetuadas em um determinado componente podem causar aos outros componentes da arquitetura. O termo impacto aqui se refere ao fato de que uma determinada alteração em um componente pode disparar necessidades de alterações em outros componentes. Assim, foi elaborada uma matriz de relacionamentos, conforme é mostrado na Tabela 9, contrastando as atividades de manutenção com o impacto de alterações que eventualmente possam se propagar pela arquitetura. Na primeira linha estão os módulos e as atividades de manutenção (Inclusão, Alteração, Exclusão). Na primeira coluna são apresentados os módulos que recebem algum

impacto. Nas intersecções são mostradas as letras N (Não) para assinalar que não há impacto e a letra S (Sim) para assinalar que há impacto entre atividade e módulo.

Conforme é mostrado na Tabela 9, as atividades de inclusão e alteração de sensores não causam impacto de manutenção nos outros módulos da aplicação. Já a atividade de exclusão de sensores pode causar impacto nos módulos de Comportamentos e Loops de Controle. As atividades de inclusão, alteração ou exclusão de Loops de Controle não causam impactos nos outros módulos da aplicação.

Tabela 2. Matriz de relacionamentos (Manutenção/Evolução x Impacto).

		MANUTENÇÃO (Inclusão, Alteração, Exclusão)															Total de Impactos	
		Sensores			Atuadores			Comportamentos			Ambiente			Loops de Controle				
		I	A	E	I	A	E	I	A	E	I	A	E	I	A	E	Sim	Não
IMPACTO	Sensores				N	N	N	N	N	N	N	N	N	N	N	N	0	12
	Atuadores	N	N	N				N	N	N	N	N	N	N	N	N	0	12
	Comportamentos	N	N	S	N	N	S				N	S	S	N	N	N	4	8
	Ambiente	N	N	N	N	N	N	N	N	N				N	N	N	0	12
	Loops de Controle	N	N	S	N	N	S	N	S	S	N	S	S				6	6
																Totais =	10	50
																	17%	83%

Assim como nos outros módulos, os pontos nos quais existem a ocorrência desses impactos de modificação são apontados pela arquitetura de referência. Nas aplicações que são desenvolvidas sem a utilização da arquitetura SARA^{MR} e que não levam em conta uma estratégias como: classes espelho, gerenciamento de comportamentos e loops de controle externalizados, as atividade de manutenções evolutivas podem ser iguais ou até mais complexas de serem realizadas.

8.5 Considerações Finais

Neste capítulo a arquitetura de referência SARA^{MR} foi avaliada de acordo com as suas propriedades de facilitar a manutenção evolutiva nos sistemas que a utilizam como apoio em seu desenvolvimento. Foi demonstrado, com os exemplos de código que é possível rastrear e determinar exatamente quais os locais que deverão receber alterações provenientes das atividades de manutenção evolutiva.

Capítulo 9

CONCLUSÃO

9.1 Considerações sobre a Pesquisa

Este trabalho de pesquisa apresentou a arquitetura de referência SARA^{RM} como uma proposta para apoiar a construção de sistemas de software para robôs móveis autoadaptativos. A arquitetura SARA^{RM} apresenta uma estrutura com os módulos: i) módulo para a implementação da parte física que compõe o robô; ii) módulo para implementação do ambiente e navegação; iii) módulo para a implementação de comportamentos, e iv) módulo de autoadaptação baseada em *loops* de controle. A forma como os módulos foram estruturados e organizados na arquitetura visam facilitar as atividades de manutenção e a evolução dos sistemas construídos com o seu apoio. O módulo de loops de controle tem o objetivo de separar as preocupações de funcionalidades do sistema das preocupações da autoadaptação.

Como parte da pesquisa foram apresentados dois exemplos de aplicação para demonstrar as formas de implementação dos módulos e a utilização da arquitetura. Além disso, foi apresentada uma avaliação da arquitetura, com exemplos de código, demonstrando a facilidade de manutenção evolutiva dos módulos da aplicação de acordo com os quatro cenários de evolução identificados no domínio de aplicações para robôs móveis autoadaptativos.

9.2 Contribuições

Este trabalho é um dos primeiros que focam especificamente em uma arquitetura de referência para robôs móveis autônomos autoadaptativos baseada em *loops* de controle. Uma de suas principais contribuições é fornecer uma estrutura com diretrizes, técnicas e padrões

para melhorar as atividades ligadas às tarefas de manutenção evolutiva no software desse tipo de robô.

Outra contribuição importante foi a abordagem utilizada para a extração e especificação dos requisitos para a criação da arquitetura. A metodologia baseada em cenários que foi usada pode servir de base para que outros arquitetos de software criem outras arquiteturas de referência para outros domínios.

Outra contribuição deste trabalho foi a padronização de módulos para as principais funcionalidades inerentes ao domínio de robôs móveis autônomos, como módulos para implementação da parte física que compõe o robô, para implementação do ambiente e navegação, e para a implementação de comportamentos. Em adicional, foi alcançado o objetivo de separar as preocupações de funcionalidades do sistema das preocupações das funcionalidades de autoadaptação com o estabelecimento de um módulo de autoadaptação baseada em *loops* de controle.

Uma outra contribuição deste trabalho, que estabelece uma modelagem para boas práticas de projeto e desenvolvimento, que facilitam a evolução de aplicações para robôs autônomos autoadaptativos, é o módulo de integração entre a aplicação base do robô e os *loops* de controle. A utilização desse módulo também possibilitou a integração de um *loop* de controle a outro *loop* de controle.

9.3 Dificuldades e Limitações

Uma limitação do trabalho realizado é quanto a falta de arquiteturas de referência para robôs móveis autoadaptativos que apresentem outras soluções que poderiam servir de base em um estudo comparativo. Por esse motivo, não foi realizada uma avaliação quantitativa.

Uma dificuldade, que é natural ao desenvolvimento de aplicações robóticas, é que o software tem de ser executado e testado diretamente no sistema do robô, ou seja, o software é implementado no ambiente da plataforma de desenvolvimento no computador, mas precisa ser instalada no sistema do robô para poder ser executada e testada com os dispositivos eletromecânicos como sensores e atuadores. Isso causa uma grande dificuldade, e gasta-se tempo considerável com as tarefas de rastreamentos de erros nas classe.

Outra limitação é que a solução desenvolvida neste trabalho foi construída com base no paradigma de orientação a objetos e grande parte dos sistemas robóticos na indústria de sistemas embarcados ainda utiliza como plataforma a linguagem C que fornece uma plataforma para o desenvolvimento no paradigma procedimental.

O estudo foi realizado apenas com a plataforma Lego NXT que é uma plataforma para estudos acadêmicos.

Não foi realizado um estudo dos impactos de balanceamento entre projeto e desempenho, assim não foi possível medir o quanto a modularidade da arquitetura poderia interferir no desempenho da aplicação robótica.

9.4 Trabalhos Futuros

Há vários trabalhos que podem ser realizados no futuro como continuação da pesquisa iniciada neste dissertação de mestrado. A seguir, é apresentada uma lista de sugestões:

- Com o surgimento de novas arquiteturas de referência para robôs móveis autoadaptativos, um estudo comparativo, envolvendo essas arquiteturas, poderia ser realizado.

- Outra implementação da arquitetura de referência SARA^{RM} poderia ser realizada em outra plataforma de desenvolvimento diferente da Lego NXT com LeJOS NXJ.

- Um estudo poderia ser realizado sobre o impacto referente às alterações com a inclusão de um novo módulo à arquitetura. Neste trabalho, em consequência das limitações da plataforma Lego NXT, não foi desenvolvido um módulo de comunicação de dados.

- Uma investigação sobre o uso de outros padrões de projeto na arquitetura de referência, além dos que foram utilizados.

REFERÊNCIAS

ABEYWICKRAMA, Dhaminda B.; ZAMBONELLI, Franco; HOCH, Nicklas. Towards simulating architectural patterns for self-aware and self-adaptive systems. In: Self-Adaptive and Self-Organizing Systems Workshops (SASOW), IEEE Sixth International Conference on. IEEE, 2012. p. 133-138, 2012.

ALBUS , J. S., "4D/RCS - a reference model architecture for intelligent unmanned ground vehicles," Unmanned Ground Vehicle Technology, vol. 4715, pp. 303–310, 2002.

ARKIN, Ronald C. Motor schema-based mobile robot navigation: An Approach to Programming by Behavior. IEEE The International journal of robotics research, Raleigh, NC, p. 264-271, 1987.

AFFONSO, Frank José; NAKAGAWA, Elisa Yumi. A Reference Architecture Based on Reflection for Self-Adaptive Software. In: Software Components, Architectures and Reuse (SBCARS), 2013 VII Brazilian Symposium on. IEEE, 2013. p. 129-138.

ANGELOV, S., GREFEN, P.W.P.J., GREEFHORST, D.: A classification of software reference architectures: Analyzing their success and effectiveness. In: WICSA 2009, Cambridge, UK, Sep 2009, pp. 141–150

ANGELOV, S., TRIENEKENS, J., and GREFEN, P., "Towards a Method for the Evaluation of Reference Architectures: Experiences from a Case," in Software Architecture, 2nd European Conf., ECSA 2008, Springer, 2008, pp. 225-240.

ASTROM, K., WITTENMARK, B.: Adaptive Control, 2nd ed. Addison-Wesley, Reading, 1995

ARKIN, R.C., Behavior-based Robotics, 2 ed. Cambridge: MIT Press,1999.

BAKER, Christopher R.; DOLAN, John M.; WANG, Shige; LITKOUHI, Bakhtiar B.. "Toward adaptation and reuse of advanced robotic software." In Robotics and Automation (ICRA), 2011 IEEE International Conference on, pp. 6071-6077. IEEE, 2011.

BASS, L.; CLEMENTS, P.; KAZMAN, R. Software Architecture in pratice. 2nd ed. Boston, MA: Addison Wesley, 2003. 560 p.

HAYES-ROTH, Barbara et al. A domain-specific software architecture for adaptive intelligent systems. IEEE Transactions on Software Engineering, vol. 21, no. 4, pp. 288–301, 1995.

BARR, Michael.; MASSA A. J., "Introduction". Programming embedded systems: with C and GNU development tools. O'Reilly. pp. 1–2, 2006

BARR, Michael. "Embedded Systems Glossary". Barr Group, Neutrino Technical Library. disponível em http://www.barrgroup.com/Embedded-Systems/Glossary-A, acesso em: 19 de junho de 2014

BARRAQUAND, Jerome; LATOMBE, Jean-Claude. Robot motion planning: A distributed representation approach. The International Journal of Robotics Research, v. 10, n. 6, p. 628-649, 1991.

BARCHANSKI, J.A., "Safety of mobile robot control architectures," Mechatronics and Automation, 2005 IEEE International Conference , vol.2, no., pp.917,922 Vol. 2, 29 July-1 Aug. 2005

BERGER, Matthias Oliver; KUBITZ, Olaf; DUMOULIN, René; POSIELEK, Robert. "A modular, layered client-server control architecture for autonomous mobile robots." In Industrial Electronics, ISIE'97. Proceedings of the IEEE International Symposium on, vol. 2, pp. 697-701. IEEE, 1997.

BRAGANÇA, A. and MACHADO, R. J.. Adopting computational independent models for derivation of architectural requirements of software product lines. In Proc. of the 4th Int. Workshop on Model-Based Methodologies for Pervasive and Embedded Software - MOMPES'07, pp. 91–101, Braga, Portugal, Mar. 2007.

BRÄUNL, Thomas. Embedded Robotics. Mobile Robot Design and Applications with Embedded Systems. 3rd ed. Australia: Springer, 2008

BROOKS, Rodney A. A robust layered control system for a mobile robot. Robotics and Automation, IEEE Journal of 2, no. 1, p. 14-23, 1986.

BROOKS, Alex; KAUPP, Tobias; MAKARENKO, Alexei; WILLIAMS, Stefan; OREBACK, Anders. "Towards component-based robotics." In Intelligent Robots and Systems. IEEE/RSJ International Conference on, pp. 163-168, 2005.

BROWN, G., CHENG, B.H., GOLDSBY, H., ZHANG, J.: Goal-oriented specification of adaptation requirements engineering in adaptive systems. In: ACM International Workshop on Self-Adaptation and Self-Managing Systems, Shanghai, China, pp. 23–29, 2006.

BROTEN G.; MONCKTON, S.; GIESBRECHT, J. & COLLIER, J., Software Systems for Robotics: An Applied Research Perspective. International Journal of Advanced Robotic Systems, Vol. 3, No. 1, pp. 11-16, 2006

BRUGALI, D., and SALVANESCHI, Paolo. "Stable aspects in robot software development." International Journal of Advanced Robotic Systems 3, no. 1, 2006.

BRUGALI, D. Software abstractions for modeling robot mechanisms. In: Advanced intelligent mechatronics, IEEE/ASME international conference on. IEEE, p. 1-6, 2007.

BRUN, Yuriy et al. Engineering self-adaptive systems through feedback loops. In: Software engineering for self-adaptive systems. Springer Berlin Heidelberg, p. 48-70, 2009.

CECCARELLI, N., DI MARCO, M., GARULLI, A., GIANNITRAPANI, A., & VICINO, A. Path planning with uncertainty: A set membership approach. International Journal of Adaptive Control and Signal Processing, v. 25, n. 3, p. 273-287, 2011.

CHENG, S.W., GARLAN, D., SCHMERL, B.: Making self-adaptation an engineering reality. In: Babaoglu, O., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M. (eds.) SELF-STAR. LNCS, vol. 3460, pp. 158–173. Springer, Heidelberg, 2005.

CHENG, Betty HC et al. Software engineering for self-adaptive systems: A research roadmap. In: Software engineering for self-adaptive systems. Springer Berlin Heidelberg, p. 1-26, 2009.

CLARK, M. N., "JAUS compliant systems offers interoperability across multiple and diverse robot platforms," in AUVSI'2005, Baltimore, USA, pp. 249–255, 2005.

CLEMENTS, P., GARLAN, D., BASS, L., STAFFORD, J., NORD, R., IVERS, J., & LITTLE, R. Documenting software architectures, views and beyond. Addison-Wesley, 2002.

CLEMENTS, P.; BACHMANN, F.; BASS, L.; GARLAN, D.; IVERS, J.; LITTLE, R.; NORD, R.; STAffORD, J. Documenting software architectures: Views and beyond. SEI series in software engineering, 2 ed. Boston, MA: Addison-Wesley, 2011.

COMINOS, P.; MUNRO, N. PID controllers: recent tuning methods and design to specification. IEE Proceedings-Control Theory and Applications, v. 149, n. 1, p. 46-53, 2002.

DAYANG N. A. JAWAWI; ROSBI M. and SAFAAI D., "A Component-Oriented Programming for Embedded Mobile Robot Software". International Journal of Advanced Robotic Systems, Vol. 4, No. 2, 2007.

DELAFOSSE, M.; CLERENTIN, A.; DELAHOCHE, L.; BRASSART, E., "Uncertainty and Imprecision Modeling for the Mobile Robot Localization Problem," Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on , vol., no., pp.4550,4555, 2005

DIAO, Y., HELLERSTEIN, J. L., PAREKH, S., GRIFFITH, R., KAISER, G., & PHUNG, D., Self-managing systems: A control theory foundation. In Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the (pp. 441-448). IEEE, 2005.

DOBSON, S., DENAZIS, S., FERNANDEZ, A., GAITI, D., GELENBE, E., MASSACCI, F., NIXON, P., SAFFRE, F., SCHMIDT, N., ZAMBONELLI, F.: A survey of autonomic communications. ACM Transactions Autonomous Adaptive Systems (TAAS), 223–259, 2006.

EELES, P." Capturing Architectural Requirements," IBM DeveloperWorks, 15 Nov. 2005. Disponível em; www-128.ibm.com/developerworks/rational/library/4706.html. Acesso em 08 de abril de 2015.

EDWARDS, G., GARCIA, J., TAJALLI, H., POPESCU, D., MEDVIDOVIC, N., SUKHATME, G., & PETRUS, B.. Architecture-driven self-adaptation and self-management in robotics systems. In Software Engineering for Adaptive and Self-Managing Systems. SEAMS'09. ICSE Workshop on (pp. 142-151). IEEE, 2009.

FIERRO, Rafael, Aveek DAS, John SPLETZER, Joel ESPOSITO, Vijay KUMAR, James P. OSTROWSKI, George PAPPAS "A framework and architecture for multi-robot coordination." The International Journal of Robotics Research 21, no. 10-11, 977-995, 2002.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. Design patterns: Elements of reusable object-oriented software. Addison Wesley, 1995.

GARLAN, D.; SHAW, Mary. An introduction to software architecture. Advances in software engineering and knowledge engineering, vol. 1 p. 1-40, Singapure 1994.

GARLAN, D., Software Architecture: a Roadmap. In 22th InternationalConferenceon The Future of Software Engineering(ICSE 2000), pages 91–101, New York, NY, USA. ACM Press, 2000.

GARLAN, D., CHENG, S.W., SCHMERL, B.: Increasing system dependability through architecture-based self-repair. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677. Springer, Heidelberg, 2003.

GARLAN, David, SHANG-WEN Cheng, AN-CHENG Huang, BRADLEY Schmerl, and STEENKISTE, Peter. "Rainbow: Architecture-based self-adaptation with reusable infrastructure." Computer 37, no. 10, 2004.

GEORGAS, John C.; TAYLOR, Richard N. Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In: Proceedings of the international workshop on Software engineering for adaptive and self-managing systems. p. 105-112, ACM, 2008.

GOOGLE CODE, https://code.google.com/hosting/search?q=leJOS&sa=Search, último acesso em abril de 2015.

HAYES-ROTH, Barbara; PFLEGER, Karl, LALANDA, Philippe; MORIGNOT, Philippe; Balabanovic, Marko. "A domain-specific software architecture for adaptive intelligent systems." Software Engineering, IEEE Transactions on 21, no. 4, p. 288-301, 1995.

HEATH, Steve, Embedded systems design. "An embedded system is a microprocessor based system that is built to control a function or a range of functions." 2 ed. Newnes. p. 2, 2003.

HOFMEISTER, C.; NORD, Robert; SONI, Dilip. Applied software architecture. Addison-Wesley Professional, 2000.

IBM Corporation: An architectural blueprint for autonomic computing. White Paper, 4th edn., IBM Corporation. http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20 Paper%20V7.pdf . Acesso em abril de 2015.

KAZMAN, R., BASS, L., ABOWD, G., and WEBB, M. SAAM: A method for analyzing the properties of software architectures. In Proc. of the 16th Int. Conf. on Software Engineering, pages 81–90, Sorrento, Italy, 1994.

KAZMAN, R., KLEIN, M., CLEMENS, P. ATAM: Método para Arquitetura avaliação, o Relatório Técnico, CMU / SEI-2000-TR-004 ESC-TR-2000-004, Agosto de 2000.

KLEIN, M., KAZMAN, R., BASS, L., CARRIERE, J.,BARBACCI, M., LIPSON, H. Attribute-based architecture styles. In Proc. of the IEEE/IFIP First Workshop Conf. on Software Architecture (WICSA-1), p. 225-243, 1999.

KRAMER, James; SCHEUTZ, Matthias. Development environments for autonomous mobile robots: A survey. Autonomous Robots, v. 22, n. 2, p. 101-132, New York: Springer, 2007.

KEPHART, J. O. and CHESS ,D. M., “The vision of autonomic computing,” IEEE Computer, vol. 36, no. 1, pp. 41–50, 2003.

KRUCHTEN, Philippe. The rational unified process: an introduction. Addison-Wesley Professional, 2004.

KUMBASAR, T.; HAGRAS, H., "A Type-2 Fuzzy Cascade Control Architecture for Mobile Robots," Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on , vol., no., pp.3226,3231, 13-16 Oct. 2013

LADDAGA, R. Active software. In Proc. of Int. Workshop on Self-Adaptive Software. 11–26, 2000.

LEGO. Lego.com MINDSTORMS. 2014. Http://mindstorms.lego.com/. Último acesso em abril de 2015.

LeJOS NXJ, Java for LEGO MindStorms, NXJ Technology, http://www.lejos.org/nxj.php. Último acesso em abril de 2015.

LEE, Tae-You; SEO, Hyung-Rok; LEE, Byung-Hyug, SHIN, Dong-Ryeol. "A software component model and middleware architecture for intelligent mobile robot." In Computer and Automation Engineering (ICCAE), The 2nd International Conference on, vol. 4, pp. 453-456. IEEE, 2010.

LIU, H., PARASHAR, M.: Accord: a programming framework for autonomic applications. IEEE Transactions on Systems, Man, and Cybernetics 36(3), 341–352, 2006.

MACÍAS-ESCRIVÁ, F. D., HABER, R., TORO, R., & HERNANDEZ, V. Self-adaptive systems: A survey of current approaches, research challenges and applications. Expert Systems with Applications, 7267-7279, 2013.

MARTÍNEZ-FERNÁNDEZ, S., AYALA, C., FRANCH, X., MARQUES, H. and AMELLER, D. 2013. A Framework for Software Reference Architecture Analysis and Review. ESELAW-CIbSE 2013.

MEDEIROS, Adelardo A.D. "A Survey of Control Architectures for Autonomous Mobile Robots". JBCS - Journal of the Brazilian Computer Society, special issue on Robotics, vol. 4, n. 3, 1998.

METTALA, E. and GRAHAM M. H., eds., The Domain-Specific Software Architecture Program. No. CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June, 1992.

MOONZOO, Kim, LEE, Jaejoon, KANG, Kyo Chul, HONG, Youngjin, and Seokwon BANG. "Re-engineering software architecture of home service robots: A case study." In Proceedings of the 27th international conference on Software engineering, pp. 505-513. ACM, 2005.

NAKAGAWA, Elisa Yumi; OQUENDO, Flavio; BECKER, Martin. Ramodel: A reference model for reference architectures. In: Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), Joint Working IEEE/IFIP Conference on. IEEE, p. 297-301, 2012.

NAKAGAWA, E. Y., "Uma Contribuição ao Projeto Arquitetural de Ambientes de Engenharia de Software," Tese de Doutorado, Universidade de São Paulo, São Carlos, SP,

Brasil, 2006.

NAKAGAWA E. Y., e MALDONADO J. C., "Requisitos Arquiteturais como Base para a Qualidade de Ambientes de Engenharia de Software," IEEE Latin America Transactions, vol. 6, no. 3, pp. 260-266, July 2008.

NAKAGAWA, E.Y.; GUESSI, M.; MALDONADO, J.C.; FEITOSA, D.; OQUENDO, F., "Consolidating a Process for the Design, Representation, and Evaluation of Reference Architectures," *Software Architecture (WICSA), IEEE/IFIP Conference on* , vol., no., pp.143,152, 7-11 April 2014.

NORMEY-RICO, Julio E., ALCALÁ, I., GÓMEZ-ORTEGA, J., & CAMACHO, E. F. Mobile robot path tracking using a robust PID controller. Control Engineering Practice, v. 9, n. 11, p. 1209-1214, Elsevier, 2001.

OREBÄCK, Anders; CHRISTENSEN, Henrik I. Evaluation of architectures for mobile robotics. Autonomous Robots Journal, Springer, Vol. 14.1: 33-49, 2003.

OREIZY, Peyman, GORLICK Michael M., TAYLOR Richard N., Dennis HEIMBIGNER, Gregory JOHNSON, Nenad MEDVIDOVIC, Alex QUILICI, David S. ROSENBLUM, and Alexander L. WOLF. "An architecture-based approach to self-adaptive software." IEEE Intelligent systems 14, no. 3, p. 54-62, 1999.

OREIZY, Peyman, NENAD Medvidovic, and Taylor, Richard N. "Runtime software adaptation: framework, approaches, and styles." In Companion of the 30th international conference on Software engineering, pp. 899-910. ACM, 2008.

PEPER, C., SCHNEIDER, D.: Component engineering for adaptive ad-hoc systems. In: ACM International Workshop on Software Engineering for Adaptive and Self-Managing Systems, Leipzig, Germany, pp. 49–56, 2008.

PETERS, L.; PAULY, M.; ARGHIR, A. Servicebots-a scalable architecture for autonomous service robots. In: Fuzzy Systems, 2000. FUZZ IEEE 2000. The Ninth IEEE International Conference on. IEEE, 2000. p. 1013-1016.

PFEIFER, Rolf; LUNGARELLA, Max; IIDA, Fumiya. Self-organization, embodiment, and biologically inspired robotics. science, v. 318, n. 5853, p. 1088-1093, 2007.

PRESSMAN, Roger. Software Engineering: A Practitioner's Approach, 9 ed. New York: McGraw Hill, 2005.

QUIGLEY M., GERKEY B., CONLEY K., FAUST J., FOOTE T., LEIBS J., BERGER E., WHEELER R., and A. N. ROS: an open-source Robot Operating System. In International Conference on Robotics and Automation, 2009.

RAKITIN, Steven R. Software verification and validation for practitioners and managers. Artech House, Inc., 1997.

RICHTER,U., MNIF,M., BRANKE,J., MULLER-SCHLOER, C., SCHMECK, H.: Towards a generic observer/controller architecture for organic computing. In: Hochberger, C., Liskowsky, R. (eds.) INFORMATIK 2006: Informatik fur Menschen. GI-Edition – Lecture Notes in Informatics, vol. P-93, pp. 112–119. Gesellschaft fur Informatik, 2006.

SALEHIE, M. e TAHVILDARI, L., “Self-adaptive software: Landscape and Research challenges,” ACM Trans. Auton. Adapt. Syst., vol. 4, no. 2, pp.1–42, May 2009.

SHAW, M.; GARLAN, D. Software architecture. Perspectives on an emerging discipline. Upper Saddle River, NJ: Prentice Hall, 1996. 242 p.

SHEIKH, U.A.; JAMIL, M.; AYAZ, Y., "A comparison of various robotic control architectures for autonomous navigation of mobile robots," Robotics and Emerging Allied Technologies in Engineering (iCREATE), 2014 International Conference on , vol., no., pp.239,243, 22-24 April 2014.

SIEGWART, Roland; NOURBAKHSH, Illah Reza; SCARAMUZZA, Davide. Introduction to autonomous mobile robots. MIT press, 2011.

SODERSTROM, T., STOICA, P.: System Identification. Prentice-Hall, Englewood Cliffs, 1988

SOMMERVILLE, Ian. Engenharia de software. 8. ed. São Paulo: Pearson Addison-Wesley, 2007. xiv, 552 p.

THRUN, S., GUTMANN, J.-S., FOX, D., BURGARD, W., KUIPERS, B., Integrating topological and metric maps for mobile robot navigation: A statistical approach. in Proceedings of the National Conference on Artificial Intelligence (AAAI), p. 989-995,1998.

THRUN, Sebastian. Learning metric-topological maps for indoor mobile robot navigation. Artificial Intelligence, v. 99, n. 1, p. 21-71, 1998.

THRUN, Sebastian; BÜCKEN, Arno. Learning Maps for Indoor Mobile Robot Navigation. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1996.

THRUN, S. Robotic mapping: A survey. In G. Lakemeyer and B. Nebel (Eds.), Exploring Artificial Intelligence in the New Millennium. Morgan Kaufmann, San Francisco, CA, USA, pp. 1–35, 2003.

THRUN, S.; BURGARD, W.; FOX, D. Probabilistic robotics. Cambridge, Massachusetts, EUA: MIT Press, 2005.

TOWLE, B.A.; NICOLESCU, M., "Real-world implementation of an Auction Behavior-Based Robotic Architecture (ABBRA)," Technologies for Practical Robot Applications (TePRA), 2012 IEEE International Conference on , vol., no., pp.79,85, 23-24 April 2012.

TRUXAL, J. G. Computers in Automatic Control Systems. Proceedings of the IRE, 1961.

VROMANT, P., WEYNS, D., MALEK, S., & ANDERSSON, J. On interacting control loops in self-adaptive systems. In Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (pp. 202-207). ACM, 2011.

WASSERMAN, A.I.: Towards a discipline of software engineering. IEEE Software 13 p. 23–31, 1996.

WEYNS, D. On Patterns for Decentralized Control in Self-Adaptive Systems. In: R. Lemos, Self-Adaptive Systems (pp. 76-107). Springer, 2013a.

WEYNS, D., IFTIKHAR, U., & SÖDERLUND, J. Do External Feedback Loops Improve the Design of Self-Adaptive Systems? A Controlled Experiment. SEAMS. IEEE, 2013b.

WOLF, Denis F., Simões, E. V., Osório, F. S., & Onofre, T. Junior,'Robótica móvel inteligente: Da simulação às aplicações no mundo real'. In: XXIX Congresso da SBC. Jornada de Atualização em Informática, 2009.

ZHUANG, M.; ATHERTON, D.P., "Automatic tuning of optimum PID controllers," Control Theory and Applications, IEE Proceedings D , vol.140, no.3, pp.216,224, May 1993.

Anexo A

GUIA DE UTILIZAÇÃO DA ARQUITETURA DE REFERÊNCIA SARA^{MR}

9.5 Introdução

Este guia de utilização tem o objetivo de auxiliar na implementação das classes que são necessárias para construir uma aplicação para um RMA. O guia é dividido em três partes conforme é descrito a seguir:

- Aplicação base do robô: abrange a criação das classes que compõem os módulos básicos que se referem à montagem da parte física do robô com os sensores, atuadores e parâmetros de configuração; a representação do ambiente e os comportamentos do robô;
- Os *loops* de controle: abrange o conjunto de classes que devem ser estendidas para a implementação de um loop de controle. Vários *loops* de controle podem ser criados para uma única aplicação;
- A execução da aplicação: abrange a criação do método *Main* com a instanciação de todas as classes criadas.

9.6 Implementação da Aplicação Base do Robô

O objetivo dessa parte é criar a classe MyRobot para representar o robô, incluindo o ambiente e os comportamentos. Para iniciar a construção de uma aplicação base, uma opção é começar criando as classes espelho para representar os sensores e atuadores que constituem o

robô. Um exemplo de código que representa a implementação de um sensor ultrassônico, pode ser encontrado no Capítulo 6, no Código 6.1. Esse procedimento pode ser feito para todos os sensores e atuadores da parte física do robô. Posteriormente, essas classes serão instanciadas e adicionadas as coleções de sensores e atuadores da classe MyRobot.

Para configurar os parâmetros que definem valores como o tamanho do robô, tamanho das rodas, distância entre eixos etc., deve ser criada a classe HardwareCharacteristics. Um exemplo de código que representa essa implementação, pode ser encontrado no Capítulo 6, no Código 6.3.

O ambiente pode ser representado com a criação da classe EnvironmentMap. Um exemplo de código que representa essa implementação, pode ser visto no Capítulo 7, no Código 7.9.

O controle do robô é efetuado com o gerenciamento de vários comportamentos. Os comportamentos do robô devem ser implementados em classes separadas com funções bem distintas, por exemplo, no caso de um robô de patrulhamento em ambiente fechado, pode haver um comportamento para automatizar o caminho a ser percorrido pelo robô, assim como pode haver um outro comportamento exclusivo para desviar o robô de obstáculos. Um comportamento é implementado com a criação da classe MyBehavior. Um exemplo de código que representa a implementação de um comportamento, pode ser encontrado no Capítulo 6, no Código 6.5. Para gerenciar esses comportamentos é utilizada a classe Arbitrator, que já está implementada pela API Lejos e deve ser apenas instanciada. Essa classe possui uma coleção interna, na qual os comportamentos devem ser adicionados. Por padrão da arquitetura SARA^{MR}, todos os comportamentos implementados já possuem acesso à leitura de valores dos sensores e atuadores.

9.7 Implementação de Loops de Controle

Para criar um loop de controle é necessário concretizar todas as classes que compõem o módulo, a saber: Monitor, Analizer, Planner, Executor, KnowledgeBase, Event, Symptom, AdaptivePolice e ControlLoop.

Um evento (Event) pode ser um valor ou um conjunto de valores que indicam um estado particular da aplicação. Esses valores são publicados no DataPublisher do módulo de

integração que será visto mais adiante. Um sintoma (Symptom) é a interpretação de um evento. Exemplo de código que representam a implementação de um loop de controle, assim como exemplos de eventos e políticas de adaptação, podem ser encontrados no Capítulo 7, entre os códigos 7.1 e 7.5. Um exemplo de código para a implementação da classe ControlLoop, que faz o gerenciamento do *loop* de controle, pode ser encontrado no Capítulo 7, no Código 7.2.

Um módulo de integração deve ser criado para cada loop de controle. Exemplo de código que representam a implementação das classes DataPublisher e RemoteFacade, do módulo de integração, podem ser encontrados no Capítulo 7, nos códigos 7.6 e 7.7.

9.8 Executando a Aplicação

Em uma aplicação Java, a execução é efetuada no método main(). Nesse método, devem ser instanciadas todas as classes criadas para a aplicação base do robô e todas as classes criadas para *loops* de controle.

No código 9.1, é mostrado um exemplo de implementação do método main(). Nessa aplicação o robô possui dois sensores (ultrassônico e toque), dois comportamentos e dois *loops* de controle. A classe MyRobot foi instanciada com o nome patrolBot, observe que nó final do código, a aplicação é inicializada invocando o método start(), com o comando patrolBot.start(). Na sequência do código, os dois *loops* de controle também são inicializados.

```
public static void main(String[] args) {

        //Configure Robot
        HardwareCharacteristics hardwareCharacteristics = new HardwareCharacteristics();
        MySensors sensors = new MySensors();
        MyActuators actuators = new MyActuators(hardwareCharacteristics);
        MyRobot patrolBot = new MyRobot(sensors, actuators);

        //Instantiate Behaviors
        MyBehavior patrolling = new Patrolling(sensors,actuators);
        //patrolling.setOnOff(true);
        MyBehavior wallfollower = new WallFollower(sensors,actuators);
        wallfollower.setOnOff(true);
        Behavior [] robotBehaviors = {(Behavior) patrolling, (Behavior) wallfollower};

        //Binding feedback loop adjust distance
        List<MyBehavior> myBehaviors = new ArrayList<MyBehavior>();
        myBehaviors.add(wallfollower);
        DataPublisherApp dataPublisherApp = new DataPublisherApp(myBehaviors, null, 0, 500);
        RemoteApp remoteApp = new RemoteApp(myBehaviors, null);

        //Instantiate feedback loops adjust distance
        //Adjust distance
        KnowledgeBaseDistance knowledgeBaseDistance = new KnowledgeBaseDistance();
        EventDistance patternDistance = new EventDistance(1, "Pattern distance");
        patternDistance.setDistance(10);
```

```
        knowledgeBaseDistance.setEventPattern(patternDistance);
        MonitorDistance monitorDistance = new
        MonitorDistance(knowledgeBaseDistance, dataPublisherApp);
        AnalyzerDistance analyzerDistance = new AnalyzerDistance(knowledgeBaseDistance);
        ExecutorDistance executorDistance = new ExecutorDistance(remoteApp);
        PlannerDistance plannerDistance = new
        PlannerDistance(knowledgeBaseDistance, executorDistance);
        ControlLoop feedbackControlLoop = new
        ControlLoop(monitorDistance, analyzerDistance, plannerDistance);

        //Binding feedback loop adjust loop
        DataPublisherAdjustLoop dataPublisherAdjustLoop = new
        DataPublisherAdjustLoop(null, knowledgeBaseDistance, 0, 500);
        RemoteAdjustLoop remoteAdjustLoop = new RemoteAdjustLoop(null, knowledgeBaseDistance);


        //Adjust feedback loop adjust loop
        KnowledgeBaseAdjustLoop knowledgeBaseAdjustLoop = new KnowledgeBaseAdjustLoop();
        MonitorAdjustLoop monitorAdjustLoop = new
        MonitorAdjustLoop(knowledgeBaseAdjustLoop, dataPublisherAdjustLoop);
        AnalyzerAdjustLoop analyzerAdjustLoop = new
         AnalyzerAdjustLoop(knowledgeBaseAdjustLoop);
        ExecutorAdjustLoop executorAdjustLoop = new ExecutorAdjustLoop(remoteAdjustLoop);
        PlannerAdjustLoop plannerAdjustLoop = new
        PlannerAdjustLoop(knowledgeBaseAdjustLoop, executorAdjustLoop);
        ControlLoop controlLoopAdjustLoop = new
        ControlLoop(monitorAdjustLoop, analyzerAdjustLoop, plannerAdjustLoop);

        //Configure Robot Behaviors and start application, data publisher and feedback loop
        patrolBot.setBehaviors(robotBehaviors);
        patrolBot.start();
        dataPublisherApp.start();
        feedbackControlLoop.start();
        dataPublisherAdjustLoop.start();
        controlLoopAdjustLoop.start();

}
```

Código 9.1 Exemplo de implementação do método main()