Framework Orientado a Objetos para Desenvolvimento de Aplicações que utilizam Arduino em Automação

Christian Alan Krötz[1], Reiner Franthesco Perozzo[1]

[1]Curso de Ciência da Computação – Centro Universitário Franciscano (UNIFRA)
97010-491 – Santa Maria – RS – Brasil

christian.alan@unifra.br, reiner.perozzo@unifra.br

Abstract. *This paper presents a proposal to create an object-oriented framework capable of assisting software developers in creating home automation projects using the Arduino as a central control platform. Three features stand out among the main ones of the proposed framework: (i) the possibility of reusing projects, (ii) reducing the complexity of automation in the use of some devices and (iii) the framework ability to evolve according to the needs of the environment.*

Resumo. *Este trabalho apresenta uma proposta de criação de um framework orientado a objetos capaz de auxiliar desenvolvedores de software na criação de projetos de automação predial/residencial que utilizem o Arduino como mecanismo central de controle. Dentre as principais características do framework proposto, destacam-se: (i) a possibilidade de reutilização de projetos, (ii) a redução da complexidade de automatização no uso de alguns dispositivos e (iii) a capacidade de evolução do framework de acordo com a necessidade do ambiente.*

1. Introdução

A automação pode ser definida como um conjunto de técnicas que fundamentam um sistema capaz de efetuar ações no meio em que atuam. A ação recebida se comporta de forma mais apropriada para aquela ação, mantendo uma relação direta entre valores de entrada e saída gerenciada por controladores com algoritmos e circuitos eletrônicos que validam essa ação [Moraes e Castrucci 2001]. Apesar de possuírem características semelhantes, a automação predial e a residencial se diferenciam em alguns aspectos: a automação predial segue um conjunto de leis, destinado aos empreendimentos comerciais e seguindo um padrão. Por outro lado, na automação residencial, o usuário tem uma interação com o ambiente quase constante, cujos objetivos podem ser diferenciados e customizados para um determinado cenário [Bolzani 2004].

Os ambientes de automação predial/residencial podem conter os mais variados requisitos, que vão desde o desenvolvimento de cenários automatizados visando o conforto dos moradores até os voltados para a segurança patrimonial. Atualmente, há uma diversidade de dispositivos eletrônicos desenvolvidos para essa área, os quais incluem sensores, atuadores e gerenciadores de automação, também conhecidos como controladores. Diante dessa variedade de dispositivos, programar esses ambientes pode ser algo complexo, uma vez que a implementação e a comunicação entre um controlador e seus dispositivos de automação podem variar de acordo com fabricantes ou

tecnologias envolvidas. Dessa forma, a utilização de um *framework* pode acelerar o desenvolvimento de um projeto de automação, pois além desses projetos serem reaproveitados, a complexidade da implementação para alguns dispositivos pode ser abstraída por esse *framework*. Isso o torna a espinha dorsal de uma aplicação, pois é ele que possui todos os dados do sistema, podendo ser customizado pelo desenvolvedor e utilizado na construção de um novo projeto dentro do mesmo domínio de aplicação [Gamma 2000].

1.1 Objetivo Geral

Este trabalho tem como objetivo geral desenvolver um *framework* orientado a objetos para auxiliar desenvolvedores na implementação de projetos de automação para ambientes prediais/residenciais, que utilizem o Arduino como dispositivo central de controle.

1.2 Objetivos Específicos

Dentre os objetivos específicos do *framework* podem-se destacar os seguintes:

- Mapear classes de dispositivos mais usados em automação predial/residencial.
- Implementar o *framework* em linguagem C++, utilizando padrões de projeto.
- Oferecer a reutilização de projetos em automação predial/residencial.
- Reduzir a complexidade de automatização no uso de alguns dispositivos que são mapeados previamente pelo *framework*.
- Permitir que o *framework* consiga evoluir de acordo com novos requisitos de projeto.

2. Referencial Teórico

Esta Seção apresenta a revisão bibliográfica que engloba as tecnologias e conceitos a serem utilizados na proposta, além dos trabalhos relacionados que contribuíram como base na elaboração da mesma.

2.1 Revisão Bibliográfica

Esta Subseção relata as tecnologias e os conceitos utilizados para a elaboração desta proposta. São apresentados os temas relacionados com *framework*, Arduino, automação em geral e a automação predial/residencial.

2.1.1 Framework

O processo de construção de *software* pode ser complexo e relativamente árduo em função dos diversos fatores que estão envolvidos nesse contexto. Para tentar diminuir essa complexidade e tornar o trabalho um pouco mais leve, vem sendo buscada a reutilização de *software*, que também não é uma tarefa trivial. Contudo, pode ser que um novo problema não necessite de uma nova solução completa e algo concretizado no passado possa ser útil em uma outra abordagem [Gamma 2000].

Com isso, o *framework* é uma técnica da Orientação a Objetos com a finalidade de, além de reutilizar código, reutilizar projetos. Ele possui três características das

linguagens de programação orientadas a objetos: (i) abstração; (ii) polimorfismo; e (iii) herança. Sendo assim, o *framework* se torna a espinha dorsal de uma aplicação, pois é nele que se descreve a arquitetura de um sistema, os tipos de objetos e suas interações, podendo ser customizado pelo desenvolvedor e reutilizado em uma nova aplicação dentro de um mesmo domínio [Gamma 2000].

Segundo Gamma (2000), na linguagem orientada a objetos, o *framework* é composto por interfaces e classes que contém as suas instâncias (que são serviços já implementados pelo *framework* e que realizam determinadas funcionalidades no sistema) ou, ainda, podem ser implementadas (através da herança como uma de suas características) pelos desenvolvedores que irão inserir os seus códigos inerentes na aplicação. Então, o *framework* pode ser uma ferramenta abrangente ou específica, tudo irá depender da sua implementação.

Dessa forma a utilização do *framework* apresenta alguns benefícios como: (i) modularização: encapsulamento da implementação através de interfaces; (ii) reutilização: reutilização do *framework* para a criação de novos sistemas; (iii) extensibilidade: capacidade de estender interfaces. Porém, para se obter esses benefícios, é de suma importância investir na qualidade do projeto do *framework*. Pois a construção de um *framework* pode ser muito mais complexa do que uma aplicação clássica, devido à alta capacidade de se criar mecanismos para que esse *framework* possa ser reutilizado. De qualquer modo, os benefícios de sua adoção podem ser evidenciados conforme novos projetos vão surgindo e a reutilização se torna frequente [Gamma 2000].

2.1.2 Arduino

O Arduino é uma plataforma de prototipagem eletrônica, de *hardware* livre, com suporte de entrada e saída de dados embutida. Isso permite ao usuário desenvolver objetos interativos que estão conectados a ele, como sensores e atuadores. O seu *hardware* é simples se comparada com outras plataformas computacionais, tal como o *Personal Computer* (PC). Porém, essa simplicidade oferece um potencial na facilidade de uso, uma vez que a plataforma é flexível e a documentação é abrangente [Mcroberts 2015].

Diversos circuitos eletrônicos, para as mais diversas áreas de atuação podem ser prototipados com essa tecnologia. Além disso, a plataforma permite que componentes de *hardware* possam ser acoplados ao sistema de modo a oferecer novas funcionalidades. Esses componentes são conhecidos como *Shields* e oferecem desde a possibilidade de conexão com diferentes interfaces de redes de comunicação de dados, até a aquisição de sinais, analógicos ou digitais, específicos. A plataforma oferece um *Integrated Development Environment* (IDE) o qual permite o desenvolvimento de aplicações, conhecidas como *sketches*, em linguagem Arduino (similar a linguagem C). Porém, outras linguagens de programação e/ou algumas extensões podem ser adotadas para que se consiga atingir uma aplicação de acordo com as suas características ou requisitos de projeto. Um exemplo é o uso da linguagem C++, a qual contempla o uso dos conceitos presentes na Orientação a Objetos [Mcroberts 2015].

2.1.3 Automação (sensores, controladores e atuadores)

A palavra *automation* foi criada pelo *marketing* da indústria de equipamentos na década de 60, com o intuito de enfatizar a participação do computador no controle automático industrial [Moraes e Castrucci 2001]. Ou seja, a automação é um conjunto de técnicas que, aplicadas a um sistema, efetua as ações recebidas no meio em que atuam. Essa ação recebida pelo sistema se comporta da forma mais apropriada, mantendo uma relação direta entre o valor de entrada e saída. Essa relação direta faz referência aos controladores do sistema, aonde que algoritmos e circuitos eletrônicos validam a ação recebida, formando assim um sistema de realimentação.

Com o conceito de *softwares* associado ao sistema, se tem a possibilidade de controle das ações, podendo produzir diferentes resultados. Entretanto, a automação emprega ações em que a saída depende da realimentação da entrada na sua malha de controle, resultando assim em um sistema de controle inteligente, aonde se associa o conceito de malha fechada ao da aberta. Todavia, em ambos os sistemas deve-se seguir as leis básicas que estabelecem a teoria geral dos sistemas, sendo que um sistema utiliza, principalmente, de realimentação das informações para o seu controle, implicando em três componentes: (i) sensores: encarregados de obter os dados do ambiente; (ii) atuadores: encarregados de executar as ações no ambiente; (iii) controladores: encarregados por processar os dados e tomar decisões [Silveira e Santos 1999].

A automação pode, ainda, ser dividida em alguns ramos como (i) automação industrial: automação de uma máquina/processo; (ii) automação comercial: utilização de *softwares* para a otimização de processos comerciais; (iii) automação residencial: buscando uma melhoria no conforto e segurança de residências.

2.1.3.1 Automação predial/residencial

A automação residencial foi baseada na industrial. Contudo, em função da diferença de infraestrutura enfrentada por ambas, ao longo dos últimos anos vem sendo elaborados novos sistemas para ambientes que não dispõem de um amplo espaço para centrais controladoras. Essa busca pela automação em ambientes de pequeno e médio porte se deu no início da década de 80. Após alguns anos, com inúmeras automações prediais/residências realizadas, devido ao surgimento da Internet e a utilização de computadores pessoais, criou-se assim uma nova cultura de acesso a informação, elevando com isso os projetos dos ambientes convencionais, em que as funções efetuadas corriqueiramente se tornassem algo integrado e efetuadas de forma conjunta [Bolzani 2004].

Apesar de existirem características muito semelhantes entre a automação predial e a residencial, elas se diferenciam em alguns aspectos: a automação predial atua na parte elétrica, hidráulica e ar-condicionado, trabalhando com sistemas distribuídos aonde segue-se um conjunto de leis locais, destinado a segurança dos empreendimentos e obtendo-se, assim, maior eficiência dos recursos disponíveis. Por ter que seguir à risca as leis determinadas, se criou um padrão para instalações prediais, visando atender uma gama maior de usuários. Em contrapartida, esse estilo de automação nem sempre fica visível ao usuário, dificultando assim a sua interação com o ambiente. O que já é bem diferente quando se trata da automação residencial, em que o usuário tem uma interação com o ambiente quase que constante. Devido a essa interação pode existir um sistema

para cada tipo de usuário, devido ao seu desejo e a sua necessidade no ambiente, tornando-se inviável a criação de uma automação padrão [Bolzani 2004].

Com isso, a automação residencial permite tornar automática uma série de ações no interior de um ambiente, oferecendo a integração de diferentes equipamentos, visando um maior conforto e controle do ambiente. É possível, ainda, alterações nas funcionalidades do ambiente, devido a automação conseguir coletar dados através de sensores e/ou dispositivos instalados e podendo efetuar um melhor gerenciamento de equipamentos com minimização de custos. A automação predial/residencial está associada com, praticamente, todas as atividades que possam ser realizadas no meio em que se encontra como, por exemplo:

1. Poder ligar/desligar ou controlar a luminosidade do ambiente;
2. Ser capaz de ligar/desligar ou ajustar a temperatura do ar-condicionado;
3. Abrir/fechar a persiana da janela;
4. Ligar/desligar uma Televisão;
5. Preparar o banho antes mesmo de chegar em casa.

2.2 Trabalhos Relacionados

Esta Subseção relata os trabalhos relacionados que são inerentes ao tema. São apresentadas abordagens que contribuem para a elaboração da proposta do trabalho.

2.2.1 Uma Abordagem Orientada a Objetos Aplicada a Automação:

Visando um código mais enxuto, inteligível e de fácil manutenção, Perozzo *et.al* (2015) tem como estratégia a utilização da linguagem orientanda a objetos como vantagem na construção de cenários computacionais microcontrolados voltados ao gerenciamento de ambientes prediais/residenciais. Para melhor visualizar essa vantagem foi apresentado um cenário de iluminação, no qual ilustra, claramente, a modularidade e o ganho em termos de reaproveitamento de código quando a estratégia é comparada com a programação estruturada clássica.

Em contrapartida da utilização da orientação a objetos, se tem um investimento maior em tempo de programação na etapa inicial, mas que se torna algo vantajoso com o passar do tempo, dada a fácil manutenção no código e a linguagem mais parecida com as classificações realizadas pelo ser humano no mundo real.

2.2.2 Frameworks para Integração entre Ambientes Inteligentes e o Sistema Brasileiro de TV Digital Terrestre

No trabalho de Perozzo (2010), foi proposto um *framework* para integrar um ambiente inteligente com o sistema brasileiro de TV digital, por meio de um *middleware* chamado Ginga. Tendo como objetivo construir aplicações orientadas a objetos, a qual gera automaticamente código para as aplicações suportadas pelo *middleware* que irão acessar serviços e dispositivos disponíveis no ambiente inteligente (ambiente automatizado ciente ao contexto). Podendo, com isso, o usuário ser capaz de gerenciar todo e qualquer serviço e/ou dispositivo de automação existente em seu meio através do controle remoto da TV.

Por se tratar de um *framework* orientado a objetos e que gera, automaticamente, o código para plataformas alvo (*set-top-boxes* ou televisores com o *middleware* Ginga

embarcado), ele possibilita a reutilização de código e projeto, permitindo assim a criação de aplicações com base em um determinado *hardware* suportado, então, pelo *middleware*. Além disso, o *framework* permite que novos dispositivos de automação sejam inseridos e controlados no ambiente automatizado a partir de um mapeamento computacional que ocorre, criando-se classes orientada a objetos para cada dispositivo. Assim, as suas características e funcionalidades são implementadas pelo *framework* e disponibilizadas para as aplicações que o utilizam.

2.2.3 Framework Web para Automação

No trabalho de Prado (2012), é proposto um *framework web* com o objetivo de auxiliar o desenvolvimento de aplicações, principalmente para ambientes residenciais, onde que o desenvolvedor não precise ter nenhum conhecimento sobre o funcionamento dos dispositivos ou sensores que ele deseje utilizar. Além de auxiliar no controle do ambiente em geral, o sistema também facilitará para o acréscimo de novas ferramentas já automatizadas.

O *framework web* para automação está baseado em um servidor *web* (Apache), que fornece uma aplicação *Python*, para o ambiente automatizado, comunicando-se com o Arduino para a execução de determinada tarefa.

2.2.4 Considerações sobre os trabalhos relacionados

Na abordagem de Prado (2012) foram identificadas algumas dificuldades durante a construção do projeto, em função do não conhecimento do programador com a linguagem adotada para a sua construção. Além disso, houve insucesso na sincronia das mensagens com o Arduino. Já, no trabalho de Perozzo (2010) é possível a criação de aplicações com base em um determinado *hardware*. Entretanto, eles devem ser suportados pelo *middleware* Ginga.

Por outro lado, Perozzo *et.al* (2015) propõe uma estratégia que serve como base para a proposta deste trabalho, uma vez que a programação orientada a objetos oferece algumas vantagens se comparada com a estruturada e seus conceitos podem ser empregados na construção de um *framework.*

De uma forma geral, os trabalhos citados convergem para o mesmo ponto, em que a criação de *frameworks* utilizando a técnica de programação orientada a objetos, oferece a capacidade de reutilização não só de código, mas também, a reutilização de projetos. Isso deve impactar nas reduções de tempo, de custo e de complexidade na construção de novos projetos para um mesmo domínio de aplicação.

3. Proposta

Devido ao atual estágio tecnológico das plataformas computacionais disponíveis atualmente e a crescente demanda por sistema de automação predial/residencial observa-se uma diversidade de aplicações desenvolvidas para esse segmento. Como estratégia de controle da automação nesses ambientes observa-se, também, uma crescente utilização do Arduino [McRoberts 2015], que é uma plataforma de prototipagem eletrônica, de *hardware* livre, com suporte de entrada e saída de dados e que permite ao usuário desenvolver sistemas interativos que estão conectados a ele, como sensores e/ou atuadores.

Diante desse contexto, a presente proposta visa criar um conjunto de classes que facilite e agilize o desenvolvimento de novas aplicações para automação, em que a plataforma Arduino esteja inserida como estratégia central de controle. Basicamente, o *framework* proposto irá conter algumas das classes dos dispositivos mais utilizados em automação predial/residencial, abstraindo para os desenvolvedores a complexidade da implementação e acelerando o processo de desenvolvimento de *software* através da reutilização de projetos, como ilustram as Figuras 1 e 2.

Figura 1: Visão Geral do *Framework*

A Figura 2 representa o diagrama de processos do *framework*.

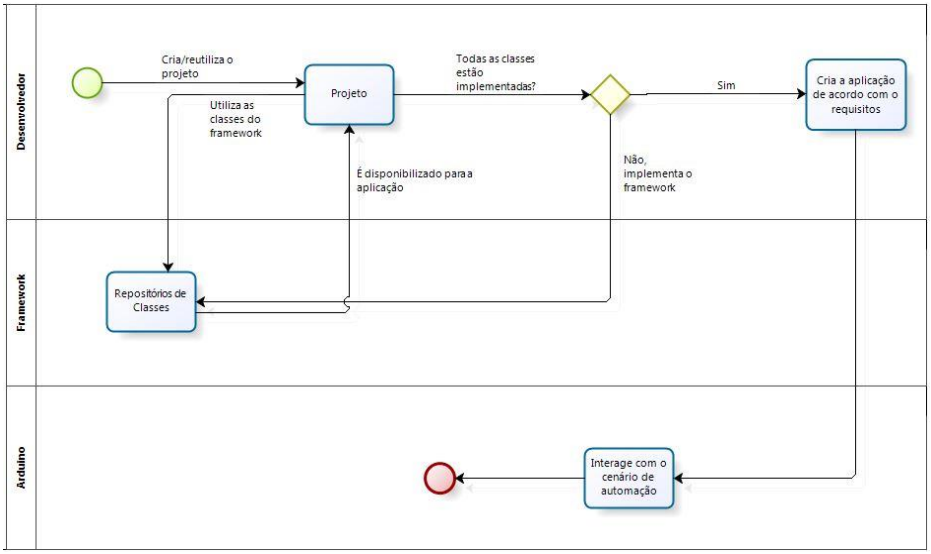


Figura 2: Diagrama de Processos

No Diagrama de Processos o Desenvolvedor cria um novo projeto a partir do *framework* proposto neste trabalho. Caso alguma classe necessária no projeto não esteja presente no *framework,* o desenvolvedor pode implementá-la, ampliando e gerando uma evolução para esse *framework*. Com o projeto finalizado é criada a aplicação de acordo com os requisitos. Por fim, a aplicação é enviada ao Arduino, o qual irá interagir com o cenário de automação em questão.

Considerando esse cenário, esta proposta está alicerçada pela metodologia FDD, pelo fato dela ser uma metodologia ágil e atender as necessidades presentes neste trabalho durante todo o período de seu desenvolvimento, a qual é dividida em cinco processos: (i) desenvolver um modelo abrangente, aonde é realizado um estudo sobre o escopo do sistema e seu contexto para se ter um modelo abrangente do *framework*. (ii)

construir a lista de funcionalidades, consiste na criação das funcionalidades que o *framework* irá disponibilizar. (iii) planejar por funcionalidades, definida a ordem/dependência entre as funcionalidades, é estipulada uma data para o término de desenvolvimento de cada funcionalidade. (iv) detalhar por funcionalidade, são produzidos os diagramas de sequência para cada funcionalidade de acordo com os seus atributos e classes. (v) construir por funcionalidade, os desenvolvedores implementam os itens necessários para que suas classes suportem a funcionalidade atribuída a aquele item. Sendo assim, neste trabalho os três primeiros processos da metodologia estão contemplados na Subseção de Projeto e os dois processos restantes na Subseção de Implementação.

3.1 Projeto

Esta Subseção contém os três primeiros processos da metodologia FDD. Inicialmente é realizado um estudo sobre o escopo do sistema e seu contexto para se ter um modelo abrangente do *framework*. Sendo realizados estudos detalhados sobre o trabalho e como ele é modelado, podendo ser representado por um diagrama de classe, por um diagrama de domínio ou qualquer outra forma que ilustre a sua estrutura [Heptagon 2014].

Diante desse contexto, é ilustrado na Figura 4 o diagrama de domínio do *framework* proposto, com uma visão conceitual para o projeto do *software.* Nesse diagrama é apresentada a ideia de que um desenvolvedor utiliza um repositório de classes (que contém as classes de dispositivos já implementadas, como os sensores e atuadores). Essas classes, por sua vez, representam os dispositivos de automação no cenário do mundo real e com esse repositório o desenvolvedor pode criar uma aplicação que será executada pelo Arduino, o qual, finalmente, irá interagir com os dispositivos de automação presentes no ambiente.

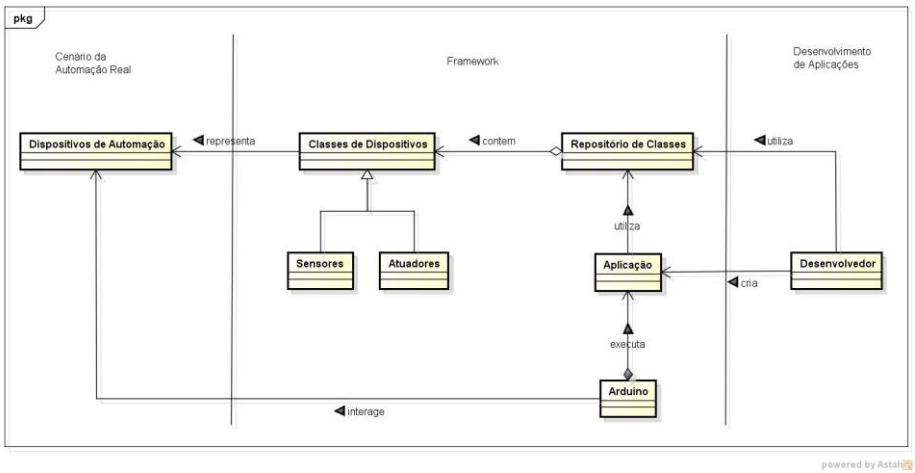


Figura 3: Diagrama de Domínio

Na tentativa de ilustrar o *framework* proposto contemplado uma visão mais refinada de sua parte estrutural (atributos, métodos e relacionamentos), é apresentado na Figura 4 o diagrama de classes do sistema, no qual já apresenta o padrão de projeto *Factory Method* a ser utilizado na implementação do *framework*. A escolha desse padrão de projeto é resultado de uma necessidade em se trabalhar com classes abstratas,

oferecendo a possibilidade de disponibilizar o *framework* com alguns métodos previamente implementados, auxiliando assim na criação de novos dispositivos.

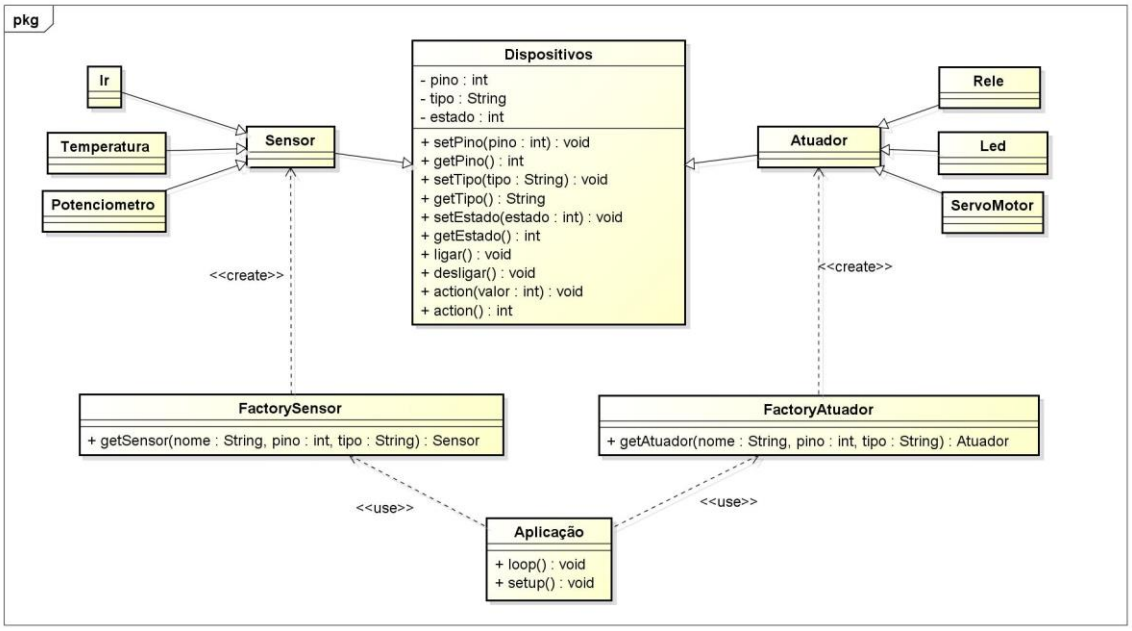


Figura 4: Diagrama de Classes

Considerando o diagrama de classes ilustrado na Figura 4 tem-se, na Tabela 1 disponíveis no Apêndice A, a descrição dos atributos e dos métodos que compõem as classes do *framework.* Cabe destacar que, em algumas classes não existem atributos e/ou métodos em função do conceito de herança presente no paradigma da orientação a objetos e, por isso, não estão listados na Tabela.

Uma vez definido o diagrama de classes, há algumas funcionalidades que o *framework* irá disponibilizar. Com isso, o propósito das funcionalidades do *software* e de seus requisitos são divididos em partes, as quais são descritas como:

- Disponibilização do *framework* para projetos de automação residencial;
- Possibilidade de ampliação do *framework* de acordo com novos requisitos de projeto;
- Oferecer uma aplicação com a estrutura base para o início de projetos no Arduino;
- Proporcionar agilidade na criação de projetos a partir do *framework* proposto, juntamente com a aplicação base;

3.2 Implementação

Esta Subseção vai ao encontro dos dois últimos processos da metodologia FDD. No qual visa produzir-se um pacote de atividades para cada funcionalidade, sendo que para cada classe que compõem o *framework* é estabelecido um pacote de funcionalidade para aquela classe. Sendo, então, desenvolvido o diagrama de sequência para cada funcionalidade [Heptagon 2014].

Buscou-se, então, desenvolver um diagrama de sequência mais abrangente, englobando assim diversos pacotes de atividades produzidos, seguindo uma linha da

utilização do *framework* para a apresentação de algumas funcionalidades disponíveis no *framework*, conforme ilustra a Figura 5.

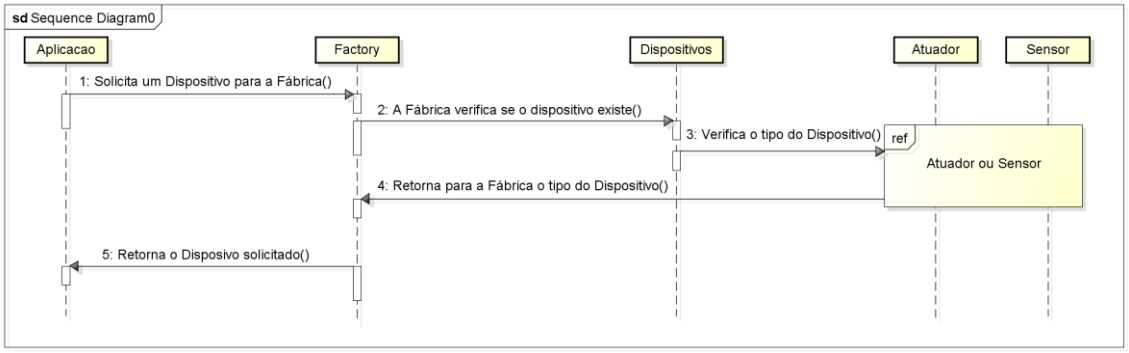


Figura 5: Diagrama de Sequência

Numa tentativa de facilitar a compreensão da implementação do *framework* é obtida como exemplo, a criação de um dispositivo de automação do tipo Led, ou seja: as fases necessárias e as ações realizadas no *framework* desde o momento da solicitação de um dispositivo de automação por uma aplicação até a configuração do dispositivo, a construção e a disponibilização desse dispositivo para a aplicação.

Dessa forma, para a criação de um dispositivo de automação orientado a objetos em C++ do tipo Led é necessário a criação de dois arquivos, um com a extensão ".h" e o outro com a extensão ".cpp", ilustrados na Figura 6, sendo o arquivo ".h" ilustrado do lado esquerdo da imagem responsável por definir os atributos e métodos do dispositivo, e o arquivo ".cpp" ilustrado do lado direito sendo responsável por conter a implementação dos métodos definidos no arquivo ".h". Os códigos aqui apresentados já contemplam a estrutura inicial do *framework*, podendo serem utilizados em projetos futuros ou até mesmo podendo serem alterados de acordo com a necessidade do projeto.

```h
#ifndef Led_h
#define Led_h

#include "Atuador.h"

class Led : public Atuador{
    public:
        Led(int pino, char *tipo);

        void action(int intervalo);
        void ligar();
        void desligar();

    private:
};
#endif
```

.h

```cpp
#include "Arduino.h"
#include "Atuador.h"
#include "Led.h"

Led::Led(int pino, char *tipo){
    Atuador::setPino(pino);
    Atuador::setTipo(tipo);

    if (tipo == "OUTPUT") pinMode(pino, OUTPUT);
    else pinMode(pino, INPUT);
}

void Led::action(int intervalo){
    digitalWrite(Atuador::getPino(), HIGH);
    delay(intervalo);
    digitalWrite(Atuador::getPino(), LOW);
    delay(intervalo);
}

void Led::ligar(){
    digitalWrite(Atuador::getPino(), HIGH);
}

void Led::desligar(){
    digitalWrite(Atuador::getPino(), LOW);
}
```

.cpp

Figura 6: Classe LED

Desta forma, o arquivo ".h" contém apenas métodos, sendo eles todos públicos (*public*), podendo ter ainda atributos públicos e ou atributos/métodos privados (*private*)

e ou protegidos (*protected*). Já no arquivo ".cpp" é feita a implementação dos métodos definidos no arquivo ".h" sendo que no método construtor da classe (Led) é realizada toda a configuração do dispositivo, através dos métodos setPino e setTipo. Nos demais métodos são realizadas ações com o dispositivo, identificando-o através do método getPino, e então realizando ações através de comandos existentes na plataforma Arduino (HIGH, LOW e DELAY).

Tendo, então, como base a classe Led disponibilizada pelo *framework*, na Figura 7 é apresentado como uma aplicação, que seria criada por um desenvolvedor, se utiliza do *framework*. Inicialmente, na primeira linha é incluída a biblioteca da fábrica de atuadores. Nas linhas três e quarto são definidos os atributos "fd" que representa a fábrica de atuadores (Figura 8) e um ponteiro "led" do tipo Atuador (Figura 9). Dentro do método setup (linha nove) a Aplicação por sua vez solicita para a fábrica de atuadores um dispositivo Led, já informando o pino em que o Led irá se conectar no Arduino e o tipo dele (I/O). Tendo como retorno da fábrica um dispositivo do tipo Atuador com as características de um Led, já herdadas as características de Atuador e também de Dispositivos (Figura 10), contemplando o Led com todas as características de Atuador e Dispositivos. Já finalizando a aplicação, dentro do método loop, é executada uma ação específica do dispositivo, que no caso faz com que o Led pisque a cada um segundo.

LED

```
1  #include <FactoryAtuador.h>
2
3  FactoryAtuador fd = FactoryAtuador();
4  Atuador *led;
5
6  void setup() {
7    Serial.begin(9600);
8
9    led = fd.getAtuador("Led", 13, "OUTPUT");
10
11   Serial.println("Iniciando teste...");
12 }
13
14 void loop() {
15   led->action(1000);
16 }
```

Figura 7: Aplicação de um dispositivo Led

```h
#ifndef FactoryAtuador_h
#define FactoryAtuador_h

#include "Atuador.h"

class FactoryAtuador{
    public:
        Atuador *getAtuador(char *nome, int pino, char *tipo);

    private:
};
#endif
```

.h

```cpp
#include "Arduino.h"
#include "FactoryAtuador.h"
#include "Atuador.h"
#include "Led.h"
#include "ServoMotor.h"
#include "Rele.h"

Atuador *FactoryAtuador::getAtuador(char *nome, int pino, char *tipo){
    if (nome == "Led"){
        Led *led = new Led(pino, tipo);
        return led;
    }
    if (nome == "Servo"){
        ServoMotor *servo = new ServoMotor(pino, tipo);
        return servo;
    }
    if (nome == "Rele"){
        Rele *rele = new Rele(pino, tipo);
        return rele;
    }
}
```

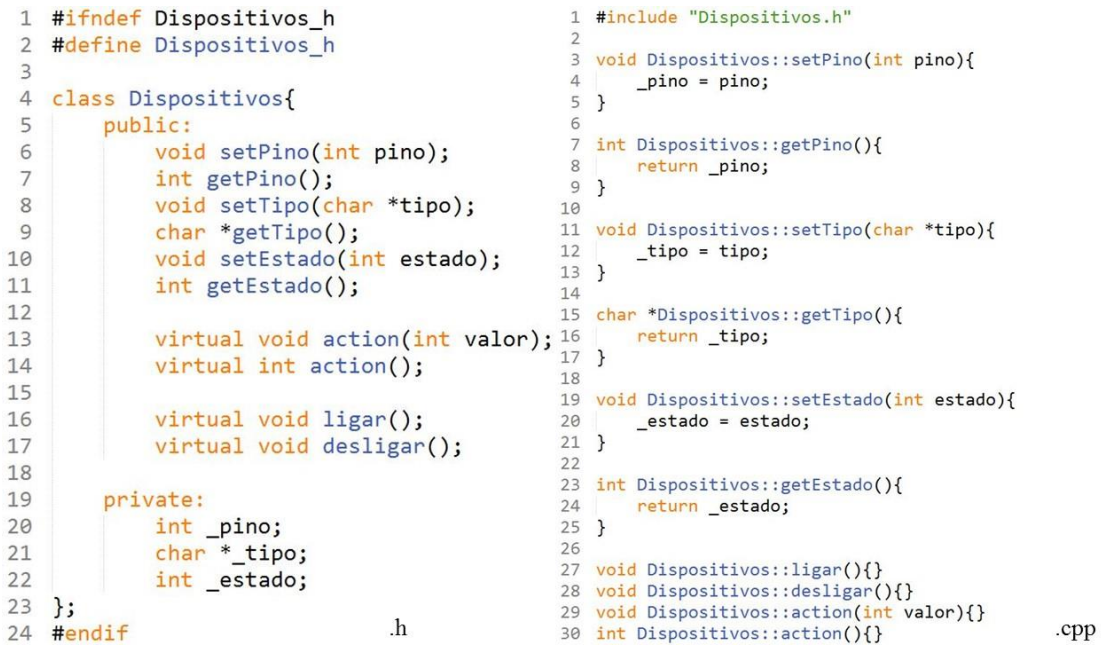
.cpp

Figura 8: Classe FactoryAtuador

```h
#ifndef Atuador_h
#define Atuador_h

#include "Dispositivos.h"

class Atuador : public Dispositivos{
    public:

    private:
};
#endif
```

.h

Figura 9: Classe Atuador



```h
#ifndef Dispositivos_h
#define Dispositivos_h

class Dispositivos{
    public:
        void setPino(int pino);
        int getPino();
        void setTipo(char *tipo);
        char *getTipo();
        void setEstado(int estado);
        int getEstado();

        virtual void action(int valor);
        virtual int action();

        virtual void ligar();
        virtual void desligar();

    private:
        int _pino;
        char *_tipo;
        int _estado;
};
#endif
```

.h

```cpp
#include "Dispositivos.h"

void Dispositivos::setPino(int pino){
    _pino = pino;
}

int Dispositivos::getPino(){
    return _pino;
}

void Dispositivos::setTipo(char *tipo){
    _tipo = tipo;
}

char *Dispositivos::getTipo(){
    return _tipo;
}

void Dispositivos::setEstado(int estado){
    _estado = estado;
}

int Dispositivos::getEstado(){
    return _estado;
}

void Dispositivos::ligar(){}
void Dispositivos::desligar(){}
void Dispositivos::action(int valor){}
int Dispositivos::action(){}
```

.cpp

Figura 10: Classe Dispositivos

Tomando como exemplo a aplicação de um Led que utiliza o *framework* com padrão de projeto, as demais classes implementadas neste trabalho seguem a mesma lógica, porém, com algumas características específicas para cada dispositivo. Dessa forma, as demais implementações estão disponíveis no Apêndice A.

4. Cenário de teste e validação

Considerando que este trabalho tem como objetivo desenvolver um *framework* orientado a objetos que auxilie desenvolvedores na criação de projetos para a automação de ambientes prediais/residenciais, os quais tenham o Arduino como mecanismo central de controle para as suas aplicações optou-se, para a validação da proposta, construir dois cenários de automação baseados em uma *Protoboard* (placa de prototipação microeletrônica). Dessa forma, o objetivo dos testes é criar duas aplicações para o Arduino, uma que utiliza o *framework* desenvolvido e a outra sem utilizar o *framework*. Isso, para cada um dos cenários de testes. Os cenários de validação e testes são compostos por LEDs, relés, *infrared* (IR) e sensores de temperatura para simular o comportamento de automação e, assim, validar o *framework* proposto. No final, serão comparados os códigos das aplicações (com e sem o uso do *framework*). O objetivo desses cenários de testes é visualizar as vantagens de se utilizar um *framework* orientado a objetos na construção de aplicações, uma vez que os conceitos de modularidade, flexibilidade e facilidade de manutenção comparados com aplicações clássicas estruturadas podem ser evidenciadas.

4.1 Cenário 1

Na construção do cenário de teste 1 (Figura 11), a *Protoboard* possui 4 ligações com a plataforma Arduino, sendo uma ligação ilustrada pela cor vermelha, duas pela cor verde e uma pela cor preta. A ligação de cor vermelha refere-se ao pino de 5V (5 volts), responsável por emitir uma corrente para alimentar os dispositivos. As duas ligações de

cor verde conectam o sensor de temperatura ao pino A0 e o relé ao pino 7 do Arduino, os quais são responsáveis por receber os dados dos dispositivos ou emitir dados aos dispositivos. Por último, a ligação de cor preta refere-se ao pino *ground (*GND) comum para que todos os dispositivos consigam fechar o circuito. Com isso, o circuito prototipado permite que o Arduino controle, de forma independente, cada um dos dispositivos ilustrados no cenário de teste.

A ideia do Cenário de Testes 1, ilustrado na Figura 11, remete-se a uma sala de estar, em que os dispositivos se comportam da seguinte maneira: o sensor de temperatura, ao captar uma temperatura superior a estipulada pelo desenvolvedor, acionará o relé que, por sua vez, ligará algum disponível para climatizar o ambiente, que no caso é meramente ilustrado por um ar condicionado.

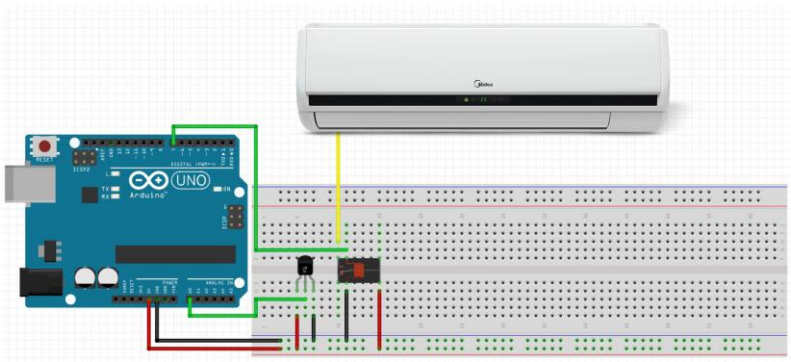


Figura 11: Cenário de Teste 1

A Figura 12, demostra dois estilos diferentes de programação aplicados em uma mesma plataforma. Do lado esquerdo há um código estruturado e do outro há um código orientado a objetos, que reaproveita as características e os comportamentos comuns a todos os objetos criados. Com isso, tem-se um código mais inteligível e de fácil manutenção. Entretanto, devido a utilização de padrões de projeto, não se tem uma diferença notável na extensão/quantidade de linhas de código.

cenario1

```
int analogPin = 0;
int valAnalog;
int temperatura;
int rele = 7;

void setup() {
  Serial.begin(9600);
  pinMode(rele, OUTPUT);
}

void loop() {
  valAnalog = analogRead(analogPin);
  temperatura = (5 * valAnalog * 100) / 1024;

  Serial.println(temperatura);
  if(temperatura >= 25){
    digitalWrite(rele, HIGH);
  }
  else{
    digitalWrite(rele, LOW);
  }
  delay(1000);
}
```

(a)

cenario1-OO

```
#include <FactoryAtuador.h>
#include <FactorySensor.h>

FactoryAtuador atuador = FactoryAtuador();
FactorySensor sensor = FactorySensor();

Atuador *rele;
Sensor *temperatura;

void setup() {
  Serial.begin(9600);

  rele = atuador.getAtuador("Rele", 7, "OUTPUT");
  temperatura = sensor.getSensor("Temperatura", A0, "INPUT");
}

void loop() {
  Serial.println(temperatura->action());
  if(temperatura->action() >= 25){
    rele->ligar();
  }else{
    rele->desligar();
  }
  delay(1000);
}
```

(b)

Figura 12: (a) Estruturada (b) Orientada a Objetos

4.2 Cenário 2

Na construção do Cenário de Teste 2 (Figura 13 ilustrada no Apêndice A), além das ligações citadas no cenário 1 (Figura 11) a *Protoboard* possui mais três ligações com a plataforma Arduino, sendo ambas as ligações ilustradas pela cor verde. Essas ligações conectam o segundo relé ao pino 3, o Led ao pino 8 e o sensor de movimento ao pino 9 do Arduino, os quais são responsáveis por receber os dados dos dispositivos ou emitir dados aos dispositivos. A ideia é que o Arduino controle, independentemente, cada um dos dispositivos ilustrados no Cenário de Teste 2.

O princípio do Cenário de Testes 2, ilustrado na Figura 13, acrescenta alguns dispositivos a mais dos ilustrados na Figura 11 criando, assim, dois ambientes incluindo a sala de estar e um banheiro, sendo que os dispositivos se comportam da seguinte maneira: (i) o sensor de presença, ao detectar um movimento no interior do banheiro, acionará (ii) o relé que, por sua vez, ligará algum disponível para climatizar o ambiente, que no caso é meramente ilustrado por um exaustor e (iii) ascenderá o Led para iluminar o ambiente. Assim, quando o sensor de movimento não detectar mais movimentos no ambiente o relé e o Led serão desligados.

A Figura 14 apresentada no Apêndice A, assim como a Figura 12, demostra dois estilos de programação diferentes aplicados em uma mesma plataforma. Seguindo a mesma ordem e estilo da Figura 12.

5. Resultados e Discussões

Baseado nos testes realizados é possível destacar que, diferentes aplicações podem sim ser criadas com base no *framework* proposto, o qual utiliza padrões de projeto na sua implementação. Resultando em uma padronização para a criação dos dispositivos. Entretanto, houveram algumas dificuldades encontradas para se trabalhar com a orientação a objetos em C++ juntamente com a utilização de padrões de projeto, pelo fato de ter que retornar os objetos dos dispositivos solicitados a fábrica pela aplicação, tendo que se utilizar de ponteiros para se trabalhar com o objeto retornado. Uma vez que a compilação das classes não resulta, necessariamente, na conformidade da aplicação/execução, sendo necessário testar a execução na plataforma Arduino. Uma vez executado, a compilação estava correta, mas não se tinha uma resposta. Sendo assim, foi necessário desenvolver aplicações que retornassem mensagens via serial para se ter uma garantia que o código desenvolvido estava, finalmente, correto.

6. Conclusão

Este trabalho apresentou uma proposta de desenvolvimento de um *framework* orientado a objetos para auxiliar desenvolvedores na criação de aplicações para ambientes automatizados que tenham o Arduino como mecanismo central de controle. Sendo realizadas pesquisas sobre o tema que envolveram o uso e a criação de *frameworks* orientados a objetos, bem como aplicações voltadas para o cenário de automação residencial com a finalidade de contribuir com a proposta apresentada.

Diante das diversas possibilidades existentes para o desenvolvimento de aplicações na área de automação residencial, optou-se pela construção de um *framework,* justamente pela possibilidade de reutilização de código e de projetos. Além disso, pode existir uma redução da complexidade de implementação presente em

determinados dispositivos de automação que se deseja utilizar em um ambiente, uma vez que a comunicação e a interação entre o Arduino e os dispositivos de automação reais são abstraídos pelo *framework,* a partir da construção de classes de dispositivos lógicos que implementam essas funcionalidades. Cabe ressaltar que a minimização da complexidade de implementação de dispositivos pode ser obtida pelos desenvolvedores de aplicações que utilizarem o *framework* proposto. Entretanto, a construção desse *framework*, assim como relatado por Gamma (2000), pode ser complexa e exigir conhecimentos específicos para essa área de aplicação. Ademais, na estrutura proposta, é contemplada a possível evolução do *framework* de acordo com a necessidade de novos dispositivos de automação.

Referências

Bolzani, Caio, A., M. (2004) "Residências Inteligentes", Livraria da Física, São Paulo.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. (2000) "Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos", Bookman, Porto Alegre.

Heptagon, TI Ltda. (2014) "FDD – Feature Driven Development", acesso em: 22 de Setembro de 2015, disponível em: http://www.heptagon.com.br/fdd.

McRoberts, Michael. (2015) "Arduino Básico", Novatec, São Paulo.

Moraes, C. C. D.; Castrucci, P. D. L. (2001) "Engenharia de Automação Industrial", LTC Editora, Rio de Janeiro.

Perozzo, Reiner, F.; Pereira, Carlos, E. (2010) "Framework para integração entre ambientes inteligentes e o sistema brasileiro de TV digital terrestre", Universidade Federal do Rio Grande do Sul, Porto Alegre-RS.

Perozzo, Reiner, F.; Zamberlan, Alexandre, O.; Oliveira, Alessandro, A. M.; Kurtz, Guilherme, C. (2015) "Uma abordagem orientada a objetos aplica à automação", Trabalho de Pesquisa, Centro Universitário Franciscano, Santa Maria-RS.

Prado, Caio, V., D. (2012) "Framework Web para Automação", Trabalho Final de Graduação – Curso de Ciência da Computação, Universidade Vila Velha, Vila Velha-ES.

Silveira, Paulo, R.; Santos, Winderson, E. (1999) "Automação e Controle Discreto", Editora Érica, São Paulo.

Apêndice A

Dispositivos	
Descrição	Classe abstrata responsável pela implementação dos métodos a serem utilizados.
Atributos	
pino	Responsável por representar o pino que será usado no Arduino.
tipo	Esse atributo é responsável por representar o tipo (I/O) de dispositivo que será conectado no pino.
estado	Representa a situação (HIGH/LOW) em que se encontra o dispositivo que no momento.
Métodos	
getPino	Método que retorna o pino no qual o dispositivo foi conectado.
setPino	Método responsável por configurar o pino do dispositivo.
getTipo	Método que retorna o tipo do dispositivo.
setTipo	Método responsável por configurar o tipo do dispositivo.
getEstado	Método que retorna o estado do dispositivo.
setEstado	Método responsável por configurar o estado do dispositivo.
ligar	Responsável por ativar um dispositivo.
desligar	Responsável por desativar um dispositivo.
action	Responsável por efetuar uma ação especifica do dispositivo em questão, de acordo com a característica específica do mesmo.
Atuador	
Descrição	Classe que herda de Dispositivos.
Sensor	
Descrição	Classe que herda de Dispositivos.
Rele	
Descrição	Classe responsável pela representação do dispositivo relé que herda as características de Atuador.
Led	
Descrição	Classe responsável pela representação do dispositivo Led que herda as características de Atuador.
ServoMotor	
Descrição	Classe que faz referência a um motor servo que herda as características de Atuador.
Ir	
Descrição	Classe responsável pela representação do dispositivo IR (*infrared*) que herda as características de Sensor.
Temperatura	

Descrição	Classe responsável pela representação do dispositivo temperatura que herda as características de Sensor.
Protenciometro	
Descrição	Classe responsável pela representação do dispositivo potenciômetro que herda as características de Sensor.
FactoryAtuador	
Descrição	Classe responsável por criar e retornar um dispositivo do tipo Atuador.
Métodos	
getAtuador	Responsável por retornar um dispositivo desejado do tipo Atuador.
FactorySensor	
Descrição	Classe responsável por criar e retornar um dispositivo do tipo Sensor.
Métodos	
getSensor	Responsável por retornar um dispositivo desejado do tipo Sensor.
Aplicação	
Descrição	Classe desenvolvida pelo desenvolvedor que irá utilizar o *framework* (com suas respectivas classes) proposto neste trabalho.
Métodos	
loop	Método responsável por executar, consecutivamente, todo o código desenvolvido na aplicação.
setup	Esse método é responsável por inicializar e definir os valores *a priori* dos atributos.

Tabela 1: Métodos e Atributos

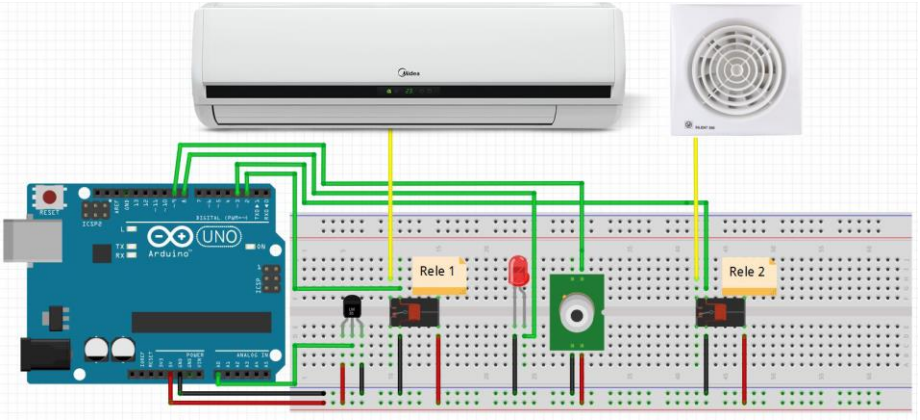


Figura 13: Cenário de Teste 2

cenario2

```ino
int analogPin = 0;
int valAnalog;
int temperatura;
int ir = 9;
int acionamento;
int releSala = 2;
int releBanheiro = 3;
int led = 13;

void setup() {
  Serial.begin(9600);

  pinMode(releSala, OUTPUT);
  pinMode(releBanheiro, OUTPUT);
  pinMode(led, OUTPUT);
  pinMode(ir, INPUT);
}

void loop() {
  // SALA
  valAnalog = analogRead(analogPin);
  temperatura = (5 * valAnalog * 100) / 1024;

  Serial.println(temperatura);
  if(temperatura >= 25){
    digitalWrite(releSala, HIGH);
  }
  else{
    digitalWrite(releSala, LOW);
  }

  // BANHEIRO
  acionamento = digitalRead(ir);
  if(acionamento == LOW){
    digitalWrite(led, HIGH);
    digitalWrite(releBanheiro, HIGH);
  }else{
    digitalWrite(led, LOW);
    digitalWrite(releBanheiro, LOW);
  }
  delay(1000);
}
```

(a)

cenario2-OO

```ino
#include <FactoryAtuador.h>
#include <FactorySensor.h>

FactoryAtuador atuador = FactoryAtuador();
FactorySensor sensor = FactorySensor();

Atuador *releSala;
Atuador *releBanheiro;
Atuador *led;
Sensor *temperatura;
Sensor *ir;

void setup() {
  Serial.begin(9600);

  releSala = atuador.getAtuador("Rele", 2, "OUTPUT");
  releBanheiro = atuador.getAtuador("Rele", 3, "OUTPUT");
  led = atuador.getAtuador("Led", 8, "OUTPUT");

  temperatura = sensor.getSensor("Temperatura", A0, "INPUT");
  ir = sensor.getSensor("Ir", 9, "INPUT");
}

void loop() {
  // SALA
  Serial.println(temperatura->action());
  if(temperatura->action() >= 25){
    releSala->ligar();
  }else{
    releSala->desligar();
  }

  // BANHEIRO
  if(ir->action() == LOW){
    led->ligar();
    releBanheiro->ligar();
  }else{
    led->desligar();
    releBanheiro->desligar();
  }
  delay(1000);
}
```

(b)

Figura 14: (a) Estruturada e (b) Orientada a Objetos

```h
#ifndef Atuador_h
#define Atuador_h

#include "Dispositivos.h"

class Atuador : public Dispositivos{
    public:

    private:
};
#endif
```

.h

Figura 15: Classe Sensor

```h
#ifndef FactorySensor_h
#define FactorySensor_h

#include "Sensor.h"

class FactorySensor{
    public:
        Sensor *getSensor(char *nome, int pino, char *tipo);

    private:
};
#endif
```

.h

```cpp
#include "Arduino.h"
#include "FactorySensor.h"
#include "Sensor.h"
#include "Ir.h"
#include "Temperatura.h"
#include "Potenciometro.h"

Sensor *FactorySensor::getSensor(char *nome, int pino, char *tipo){
    if (nome == "Ir"){

        Ir *ir = new Ir(pino, tipo);
        return ir;
    }
    if (nome == "Temperatura"){

        Temperatura *temperatura = new Temperatura(pino, tipo);
        return temperatura;
    }
    if (nome == "Potenciometro"){

        Potenciometro *potenciometro = new Potenciometro(pino, tipo);
        return potenciometro;
    }
}
```

.cpp

Figura 16: Classe FactorySensor

```h
#ifndef Ir_h
#define Ir_h

#include "Sensor.h"

class Ir : public Sensor{
    public:
        Ir(int pino, char *tipo);
        int action();

    private:
};
#endif
```

.h

```cpp
#include "Arduino.h"
#include "Sensor.h"
#include "Ir.h"

Ir::Ir(int pino, char *tipo){
    Sensor::setPino(pino);
    Sensor::setTipo(tipo);

    if (tipo == "OUTPUT") pinMode(pino, OUTPUT);
    else pinMode(pino, INPUT);
}

int Ir::action(){
    Sensor::setEstado(digitalRead(Sensor::getPino()));
    if (Sensor::getEstado() == LOW)
        return 0;
    else
        return 1;
}
```

.cpp

Figura 15: Classe *infrared* (Ir)

```h
#ifndef Potenciometro_h
#define Potenciometro_h

#include "Sensor.h"

class Potenciometro : public Sensor{
    public:
        Potenciometro(int pino, char *tipo);
        int action();

    private:
};
#endif
```

.h

```cpp
#include "Arduino.h"
#include "Sensor.h"
#include "Potenciometro.h"

Potenciometro::Potenciometro(int pino, char *tipo){
    Sensor::setPino(pino);
    Sensor::setTipo(tipo);

    if (tipo == "OUTPUT") pinMode(pino, OUTPUT);
    else pinMode(pino, INPUT);
}

int Potenciometro::action(){
    return analogRead(Sensor::getPino());
}
```

.cpp

Figura 16: Classe Potenciômetro

```h
#ifndef Rele_h
#define Rele_h

#include "Atuador.h"

class Rele : public Atuador{
    public:
        Rele(int pino, char *tipo);

        void ligar();
        void desligar();

    private:
};
#endif
```

.h

```cpp
#include "Arduino.h"
#include "Atuador.h"
#include "Rele.h"

Rele::Rele(int pino, char *tipo){
    Atuador::setPino(pino);
    Atuador::setTipo(tipo);

    if (tipo == "OUTPUT") pinMode(pino, OUTPUT);
    else pinMode(pino, INPUT);
}

void Rele::ligar(){
    digitalWrite(Atuador::getPino(), HIGH);
}

void Rele::desligar(){
    digitalWrite(Atuador::getPino(), LOW);
}
```

.cpp

Figura 17: Classe Relé

```h
#ifndef ServoMotor_h
#define ServoMotor_h

#include "Atuador.h"

class ServoMotor : public Atuador{
    public:
        ServoMotor(int pino, char *tipo);
        void action(int graus);

    private:
};
#endif
```

.h

```cpp
#include "Arduino.h"
#include "Atuador.h"
#include "ServoMotor.h"

ServoMotor::ServoMotor(int pino, char *tipo){
    Atuador::setPino(pino);
    Atuador::setTipo(tipo);

    if (tipo == "OUTPUT") pinMode(pino, OUTPUT);
    else pinMode(pino, INPUT);
}

void ServoMotor::action(int graus){
    analogWrite(Atuador::getPino(), graus);
}
```

.cpp

Figura 18: Classe Servo Motor

```h
#ifndef Temperatura_h
#define Temperatura_h

#include "Sensor.h"

class Temperatura : public Sensor{
    public:
        Temperatura(int pino, char *tipo);
        int action();

    private:
};
#endif
```

.h

```cpp
#include "Arduino.h"
#include "Sensor.h"
#include "Temperatura.h"

Temperatura::Temperatura(int pino, char *tipo){
    Sensor::setPino(pino);
    Sensor::setTipo(tipo);

    if (tipo == "OUTPUT") pinMode(pino, OUTPUT);
    else pinMode(pino, INPUT);
}

int Temperatura::action(){
    return (analogRead(Sensor::getPino())*5*100)/1024;
}
```

.cpp

Figura 19: Classe Temperatura