Universidade Federal de Campina Grande - UFCG
Centro de Engenharia Elétrica e Informática - CEEI
Departamento de Engenharia Elétrica - DEE

Bruno Nascimento Gomes de Oliveira

Paradigmas de Programação em Sistemas de Automação Industrial

Campina Grande, Brasil

Outubro de 2024

Bruno Nascimento Gomes de Oliveira

Paradigmas de Programação em Sistemas de Automação Industrial

Universidade Federal de Campina Grande - UFCG

Centro de Engenharia Elétrica e Informática - CEEI

Departamento de Engenharia Elétrica - DEE

Orientador: George Acioli Júnior, D.Sc.

Campina Grande, Brasil

Outubro de 2024

Bruno Nascimento Gomes de Oliveira

Paradigmas de Programação em Sistemas de Automação Industrial

Trabalho aprovado em: / /

George Acioli Júnior, D.Sc.
Orientador

Eisenhawer de Moura Fernandes
Convidado

Campina Grande, Brasil
Outubro de 2024

Agradecimentos

"Seja forte e corajoso! Não se apavore nem desanime, pois o Senhor, o seu Deus, estará com você por onde você andar"

Josué 1:9

Resumo

O projeto de sistemas de automação requer o uso de paradigmas de programação que otimizem o desempenho, a modularidade e a flexibilidade dos sistemas. Este trabalho apresenta uma análise comparativa de três paradigmas amplamente utilizados: Programação Orientada a Objetos (OOP), Arquitetura Orientada a Serviços (SOA) e Desenvolvimento Baseado em Componentes (CBD). Cada paradigma é aplicado em um sistema simulado de separação de caixas, utilizando o software Factory IO, que permite a interação com controladores programáveis e ambientes virtuais de simulação. A análise revela que, enquanto a OOP oferece maior controle sobre os detalhes da implementação, o CBD e a SOA permitem uma implementação mais modular e ágil, sendo mais adequados para ambientes distribuídos e que exigem flexibilidade. As vantagens e desvantagens de cada abordagem são discutidas com base nos resultados obtidos, destacando a importância de selecionar o paradigma correto para diferentes contextos de automação industrial.

Palavras-chaves: Automação Industrial, Programação Orientada a Objetos, Arquitetura Orientada a Serviços, Desenvolvimento Baseado em Componentes, CLP, IEC 61131-3, Factory IO. IEC 61499.

Abstract

The design of automation systems requires the use of programming paradigms that optimize performance, modularity, and flexibility of systems. This paper presents a comparative analysis of three widely used paradigms: Object-Oriented Programming (OOP), Service-Oriented Architecture (SOA), and Component-Based Development (CBD). Each paradigm is applied in a simulated box sorting system, using the Factory IO software, which allows interaction with programmable controllers and virtual simulation environments. The analysis reveals that while OOP offers greater control over implementation details, CBD and SOA enable a more modular and agile implementation, being more suitable for distributed environments that require flexibility. The advantages and disadvantages of each approach are discussed based on the obtained results, highlighting the importance of selecting the right paradigm for different industrial automation contexts.

Key-words: Industrial Automation, Object-Oriented Programming, Service-Oriented Architecture, Component-Based Development, PLC, IEC 61131-3, Factory IO, IEC 61499.

Lista de ilustrações

Lista de abreviaturas e siglas

CLP	Controladores Lógicos Programáveis
OOP	Programação Orientada a Objeto (*Object-oriented programming*)
SOA	Arquitetura Orientada a Serviço (*Service Oriented Architecture*)
CBD	Desenvolvimento Baseado em Componente (*Component-based development*)
UML	Linguagem de Modelagem Unificada (*Unified Modeling Language*)
PCS	Sistema de Controle do Processo
SOCRADES	*Service-oriented cross-layer infrastructure for distrinuted smart embedded devices*

Sumário

1 Introdução

Nos sistemas de automação industrial, a crescente complexidade dos processos e a demanda por maior eficiência têm impulsionado a necessidade de abordagens mais sofisticadas para a programação dos controladores lógicos programáveis (CLPs). Para enfrentar esses desafios, é fundamental compreender o conceito de paradigma de programação e sua relevância nesse contexto. (DAI et al., 2014)

Os paradigmas de programação são conjuntos de princípios e métodos que orientam a estrutura e a organização do código. Eles fornecem modelos e estratégias para criação de *software*, influenciando a maneira como os problemas são abordados e resolvidos.

Eles têm um impacto positivo nos sistemas de automação das plantas industriais, visto que trazem benefícios como eficiência no desenvolvimento, pois com o uso dessas técnicas, a criação do sistema é feita de forma mais organizada, o que permite que o trabalho dos desenvolvedores seja mais produtivo, visto que a capacidade de dividir o código em módulos menores simplifica o processo de desenvolvimento. A reutilização de código, por sua vez, permite que partes do software sejam utilizadas em diferentes contextos sem a necessidade de reescrever o código. Isso promove uma maior consistência e eficiência, pois componentes ou módulos podem ser aproveitados em diferentes projetos ou partes do sistema. Outro benefício importante é a flexibilidade, que é atingida, pois esses paradigmas permitem que os sistemas sejam ajustados e ampliados de maneira mais fácil e eficiente, sendo um aspecto crucial para sistemas que precisam se adaptar a novas demandas ou mudanças no ambiente.

Além desses benefícios, uma importante característica dessa abordagem é ser útil na modelagem de sistemas de controle distribuído, a qual é uma tendência na indústria atual, retirando a necessidade de um controlador central.

Nesse contexto, três principais paradigmas serão abordados:

- Programação Orientada a Objeto (OOP);
- Arquitetura Orientada a Serviço (SOA);
- Desenvolvimento baseado em Componente (CBD).

2 Programação Orientada a Objeto

2.1 Contextualização

A Programação Orientada a Objetos (OOP) é baseada em quatro pilares principais: encapsulamento, herança, polimorfismo e abstração. Esses princípios permitem uma maior modularização do código, o que facilita a manutenção, escalabilidade e reutilização. Nas linguagens de programação como Java e C++, amplamente utilizadas em diversos setores, a OOP proporciona uma estrutura flexível e concisa, reduzindo a repetição de código e melhorando a organização geral do desenvolvimento.

Logo, conhecendo suas vantagens e diante do aumento da complexidade das plantas industriais, a OOP foi vista como uma abordagem eficaz para lidar com esses desafios no desenvolvimento, gerando componentes flexíveis e reutilizáveis, ao utilizar os conceitos de encapsulamento, herança e polimorfismo. (WERNER, 2009)

Portanto, esse paradigma foi adicionado na terceira edição da norma IEC 61131-3 (a qual define as linguagens de programação padrão para CLPs), publicada em 2013. Para permitir a utilização desse paradigma, vários novos elementos foram adicionados.

2.2 Elementos da OOP

Antes da introdução dos conceitos da OOP, a norma IEC 61131-3 já continha um conceito básico semelhante ao de uma classe, conhecido como bloco de função. Um bloco de função representa um módulo de software que possui um estado interno e uma rotina que manipula esse estado, podendo ser instanciado diversas vezes dentro de um programa.

Logo, a extensão para compatibilidade com a OOP fez com que os blocos de função se aprimorassem para um bloco de função híbrido semelhante a uma classe. Com isso, é possível que o bloco de função híbrido tenha um número arbitrário de métodos e propriedades (que junto com a assinatura do próprio bloco de função) forma a visão externa de um bloco de função. (WITSCH; VOGEL-HEUSER, 2009)

A introdução dos métodos dentro dos blocos de função híbridos permite uma organização mais clara e modular das tarefas executadas pelos CLPs, enquanto as propriedades possibilitam o controle eficiente do acesso e manipulação dos estados internos do bloco. Essas mudanças trazem ao desenvolvimento de sistemas os benefícios da OOP, como a modularidade, a reutilização de código e a flexibilidade na atualização e expansão de funcionalidades.

2.2.1 Métodos

Em um bloco de função tradicional, toda a funcionalidade é concentrada em uma única rotina que manipula o estado interno do bloco de função, mesmo que esse bloco precise realizar diversas tarefas distintas. Para controlar essa única rotina, é necessário passar valores específicos nos *inputs* do bloco ao invocá-la, o que pode resultar em um código menos claro e mais suscetível a erros.

Com a adoção da OOP, esse código pode ser dividido em métodos separados, cada um responsável por uma tarefa específica.Isso oferece maior estruturação do código, uma vez que cada método isola e organiza uma parte da funcionalidade, facilitando o desenvolvimento e a manutenção do sistema, como mostra a figura 1 em uma comparação entre as duas formas de executar a mesma ação de *Start Pump* e *Reset Pump*.

Figura 1 – Diferença entre a chamada de métodos com a OOP

1) *Start the pump*
   Pump1(start := TRUE, Direction := Forward, Reset := FALSE); (*classical approach*)
   Pump1.Start(Direction := Forward); (*object-oriented approach*)
2) *Reset the pump*
   Pump1(start := FALSE, Reset := TRUE); (*classical approach*)
   Pump1.Reset(); (*object-oriented approach*).

Fonte: (WERNER, 2009)

Essa abordagem traz vantagens como uma maior estruturação, já que em vez de agrupar todas as funcionalidades em uma única rotina, a OOP permite que as tarefas sejam separadas em métodos diferentes. Isso torna o código mais modular e fácil de entender, pois cada método lida com uma responsabilidade específica. E uma maior readaptabilidade, pois a nomeação dos métodos reflete claramente sua função. Por exemplo, um método chamado *pump()* deixa explícito o que ele deve fazer, enquanto que, na abordagem clássica, essa lógica é frequentemente determinada apenas pelos *inputs*, o que dificulta a leitura do código.

Os métodos podem ser entendidos como funções declaradas dentro de um bloco de função com o tipo do resultado, parâmetros e variáveis locais. Como no caso das funções, as variáveis locais não mantêm seu estado de uma chamada da função para a próxima. Além disso, os métodos têm acesso implícito às variáveis internas do bloco de função,

permitindo que eles alterem o estado do bloco ao longo da execução.

No padrão IEC 61131-3, essa flexibilidade é ampliada, permitindo que os métodos não apenas executem operações tradicionais, como em outras linguagens orientadas a objetos, mas também influenciem o estado interno do bloco de função, modificando o comportamento e o estado da instância do bloco em execuções futuras. Outra possibilidade é a implementação dos métodos em qualquer linguagem de programação dentro da norma.

2.2.2 Propriedades

As propriedades são mecanismos que permitem o acesso controlado às variáveis internas de um bloco de função. Diferentemente das variáveis comuns, uma propriedade não é diretamente uma variável dentro do bloco de função, mas um meio de acessar e manipular essas variáveis internas de forma controlada. As propriedades proporcionam uma visão abstrata e externa dos dados internos do bloco de função, permitindo um controle mais seguro e organizado sobre como os dados são lidos ou alterados.

Quando uma propriedade é lida externamente (geralmente de fora do bloco de função), o acessador *get* é chamado implicitamente para retornar o valor da variável interna associada. Da mesma forma, ao tentar modificar uma propriedade, o acessador *set* é chamaso para definir o novo valor. Esses acessadores *get* e *set* são similares a métodos, mas são especificamente projetados para controlar o acesso a variáveis locais de maneira encapsulada.

Dentro da implementação dos acessadores, é possível definir regras adicionais, como a validação de valores ou operações aritméticas antes de alterar ou retornar o valor da propriedade. Por exemplo, o acessador *set* pode ser configurado para garantir que apenas valores dentro de um intervalo permitido sejam atribuídos à propriedade, garantindo a integridade do estado interno do bloco de função. Isso oferece uma camada extra de controle e segurança ao manipular variáveis sensíveis no sistema.

2.2.3 Herança

Na OOP, a herança permite que um novo bloco de função seja construído com base em um bloco de função já existente. Isso significa que o novo bloco de função (chamado de "bloco de função filho") herda todas as variáveis, métodos e propriedades do bloco de função anterior (chamado de "bloco de função pai"). Além disso, o bloco de função filho pode adicionar suas próprias variáveis e métodos, ou sobrescrever (modificar) os métodos herdados do bloco de função pai, caso seja necessário.

Um bloco de função A pode ser uma generalização de outro bloco de função B (ou B herda de A). Essa relação de herança entre dois blocos de função é especificada pela palavra chave "EXTENDS". Isso tem dois efeitos principais:

- O bloco de função estendido B herda todos os métodos e propriedades e variáveis de A (incluindo a implementação de métodos e propriedades);
- O bloco de função estendido é compatível com o tipo de dado de A (mas o inverso não é verdade).

A herança pode ser usada para reutilizar a mesma implementação em blocos de função diferentes. Se uma implementação de um ou mais métodos não for adequada para um bloco de função estendido, ele pode sobrescrever esses métodos e/ou propriedades com sua própria implementação ao definir um método/propriedade com o mesmo nome/assinatura. Devido a isso, é possível:

- construir estruturas de herança hierárquicas (porém, um bloco de função só pode herdar de UM outro diretamente);
- estruturar partes de implementação com métodos e;
- caso necessário, sobrescrever métodos nos blocos de função estendidos.

2.2.4 Interfaces

Outra forma de definir tipos de dados nos blocos de função é por meio de interfaces. Interfaces são especificações abstratas que definem um conjunto de métodos e propriedades que devem ser implementados por um bloco de função, mas sem fornecer uma implementação concreta desses métodos. E, ao contrário de uma classe ou bloco de função tradicional, uma interface não contém variáveis internas. Ela serve apenas como um "contrato", especificando o que o bloco de função deve fazer, mas não como.

Elas permitem que diferentes blocos de função compartilhem a mesma estrutura de métodos e propriedades, mas com implementações específicas para cada contexto. Ou seja, essa implementação pode ser feita de maneira independente da de outros blocos que também implementam a mesma interface. Isso permite maior liberdade ao desenvolver novos blocos de função, garantindo que eles sigam o mesmo "contrato"sem que as implementações internas sejam idênticas.

Além disso, os blocos de funções podem implementar uma ou mais interfaces. Logo, ele é obrigado a ter, no mínimo, aqueles métodos e propriedades que estão definidos nas interfaces que implementa.

Ao implementar uma interface, um bloco de função define sua conformidade com essa interface. Como resultado, pode ser usado em todos os lugares onde a interface é exigida como tipo de dado.

Uma das grandes vantagens da interface é que ela evita os problemas associados à herança múltipla. No modelo tradicional de herança, um bloco de função só pode herdar de um único bloco de função pai diretamente, o que limita a flexibilidade de reaproveitamento de código. No entanto, um bloco de função pode implementar várias interfaces ao mesmo tempo, permitindo que ele adote diferentes comportamentos e métodos definidos por diversas interfaces sem enfrentar os conflitos que surgem com a herança múltipla.

A palavra-chave "IMPLEMENT"é usada no código para indicar que um bloco de função é uma subclasse de uma ou mais interfaces. Isso significa que o bloco de função deve implementar todos os métodos e propriedades definidos pela interface. A novidade em relação à herança é que, enquanto a herança carrega a implementação de métodos do bloco de função pai, a interface apenas define os métodos, exigindo que o bloco de função forneça sua própria implementação para eles.

2.2.5 Referência Semântica

Assim como em um bloco de função, o nome de uma interface pode ser utilizado como um tipo na declaração de variáveis. Esse conceito é chamado de referência semântica. Quando uma variável é declarada como sendo do tipo de uma interface, isso não cria um novo objeto, mas estabelece uma referência para uma instância existente de um bloco de função que implementa essa interface.

Em outras palavras, a declaração de uma variável do tipo interface não resulta na criação de uma nova instância, mas sim em uma referência a uma instância de um bloco de função já existente, que implementa aquela interface. Essa abordagem proporciona flexibilidade no uso de blocos de função, pois permite que a mesma variável de interface se refira a diferentes instâncias de diferentes tipos de blocos de função que implementam a mesma interface.

Um exemplo seria uma interface chamada *Motor*, que define métodos como *Start(), Stop()* e *ChangeSpeed()*. Você pode declarar uma variável do tipo *Motor* que não cria diretamente uma instância de um motor, mas faz referência a um bloco de função específico que implementa essa interface, como um bloco de função A que controla um motor de esteira. Mais tarde, a mesma variável de tipo *Motor* pode ser usada para se referir a um bloco de função B, que controla um motor de bomba. Como ambos os blocos de função implementam a interface *Motor*, você pode utilizar a mesma variável para controlar diferentes tipos de motores, bastando mudar a referência para a instância correta.

2.3 Mapeamento OOP para Diagramas de Classe UML

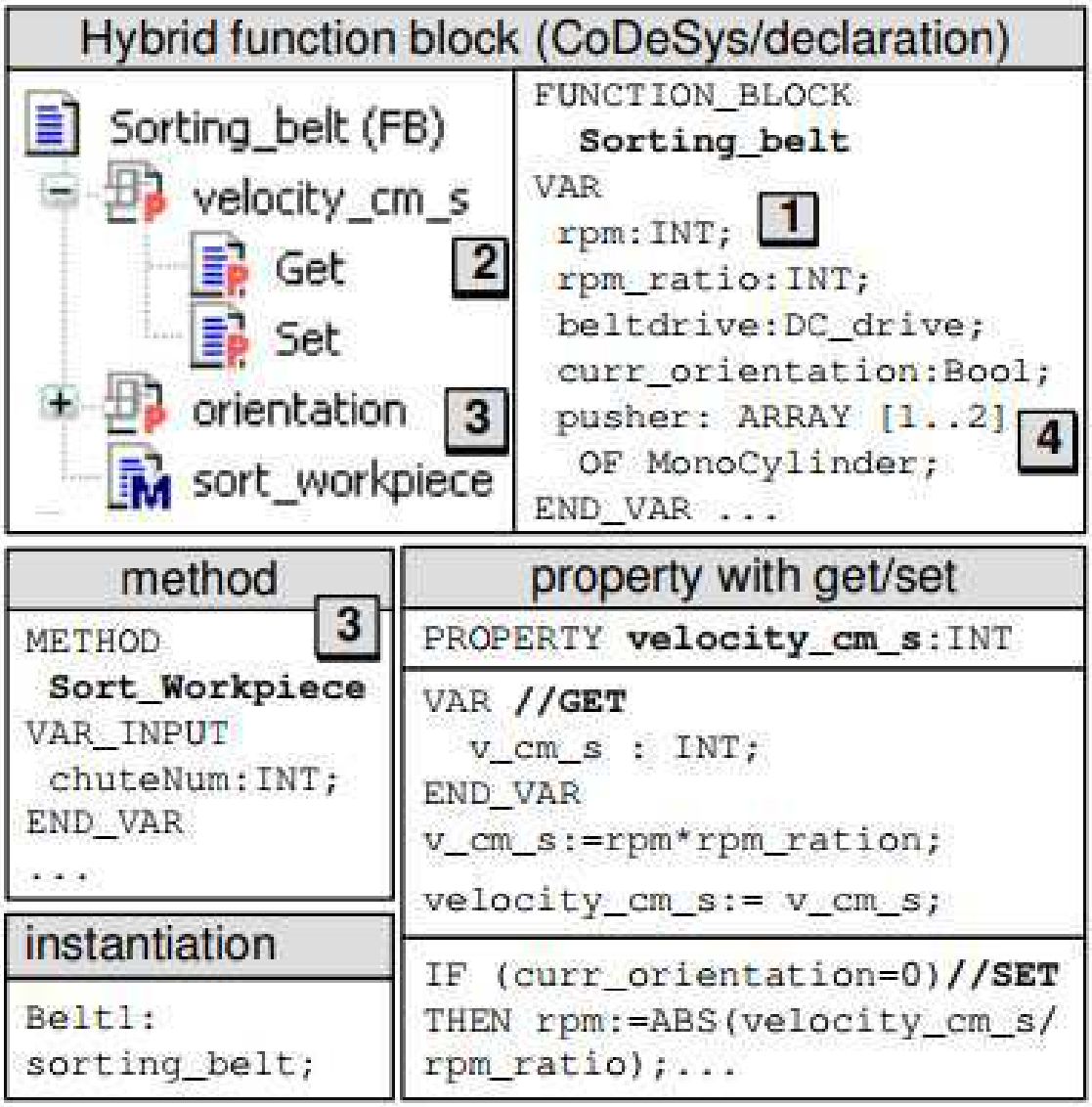
2.3.1 Mapeamento de um bloco de função para classes UML

UML (*Unified Modeling Language*) é uma linguagem de modelagem amplamente utilizada para visualização de sistemas de *software*. Ela permite que ideias abstratas e sistemas complexos sejam representados de maneira gráfica, facilitando sua compreensão. No contexto da automação industrial, a UML desempenha um papel crucial ao representar blocos de função e suas interações dentro de sistemas maiores, trazendo benefícios como:

- Facilidade na compreensão de ideias e sistemas - Os diagramas UML tornam ideias abstratas e sistemas de software complexos mais fáceis de entender através da visualização;
- Transforma códigos complexos em um diagrama visual - A construção de um *software* muitas vezes requer milhares de linhas de código complexas, com relacionamentos e hierarquias dentro delas. A interpretação deste código pode ser difícil e demorado, o uso de UML simplifica este processo ao representar os ambientes de codificação em diagramas visuais fáceis de entender;
- Permite ter o panorama geral de um sistema - Um diagrama UML ajuda a criar uma visão geral e abrangente do sistema como um todo.

Na figura 2, um bloco de função aparece na estrutura do projeto. O bloco de função possui propriedades (2 na figura) e métodos (3 na figura). As variáveis locais aparecem em 1, onde outros blocos de função são instanciados. No canto inferior direito da figura 2, um exemplo para o uso de propriedades e seus acessadores é fornecido.

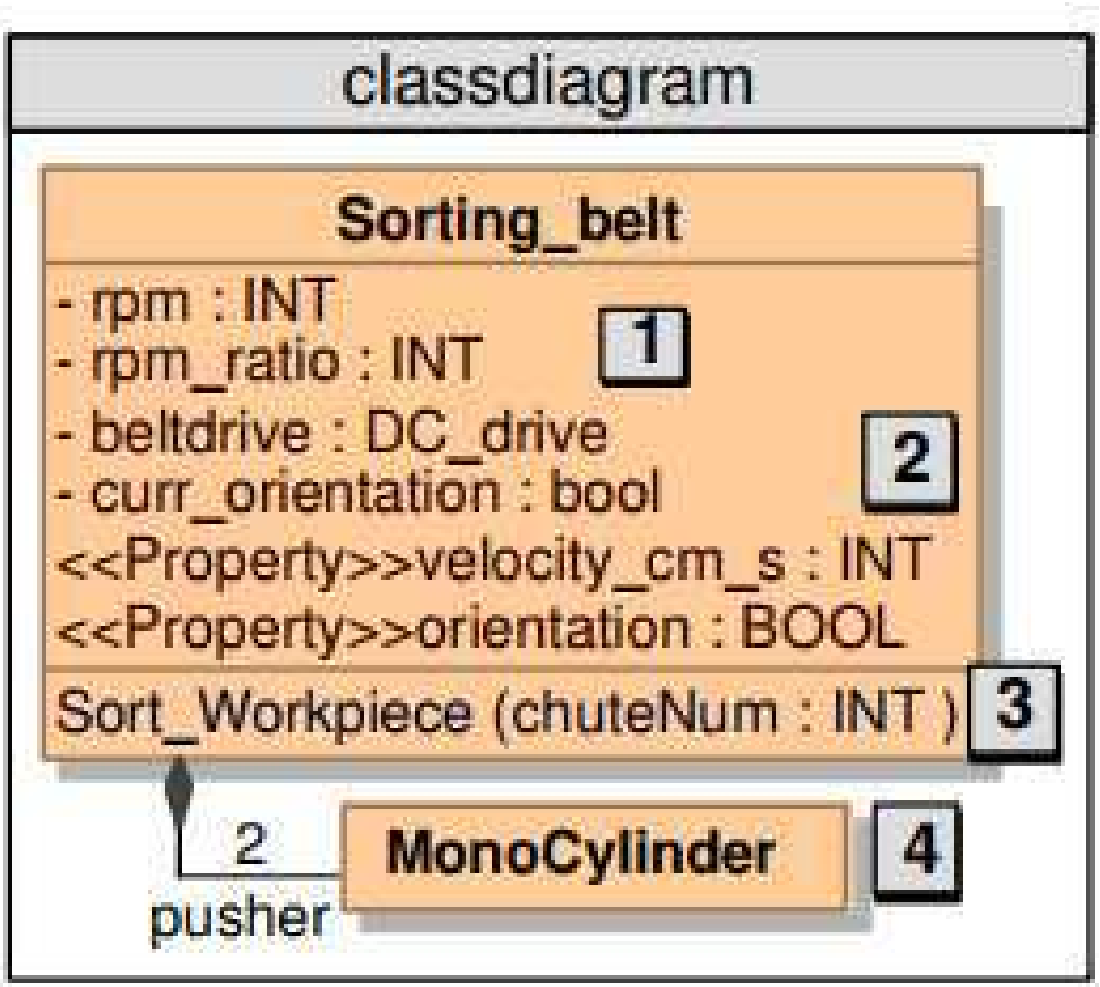
Figura 2 – Bloco de Função Híbrido



Fonte: (WITSCH; VOGEL-HEUSER, 2009)

Na figura 3, o mesmo bloco de função é mostrado como uma classe em um diagrama de classes UML.

Figura 3 – Bloco de função representado em UML



Fonte: (WITSCH; VOGEL-HEUSER, 2009)

Os números na figura 2 correspondem aos da figura 3. A classe é dividida em três seções. Na primeira seção, fica o nome do bloco de função. Na segunda seção, tem-se as variáveis do bloco de função, onde as variáveis locais são marcadas com um -", e as variáveis de entrada, com "+", já as variáveis de saída, com "+"também, porém, com *readonly* como sufixo. Já, na seção inferior, há os métodos com suas assinaturas, caso a implementação seja vazia, a assinatura é marcada com *«abstract»*, o que significa que é necessário que as classes que herdam dessa implementem esse método.

Na figura 3, está presente a classe 'MonoCylinder'. Assim, no caso que uma classe contem a instância de outra classe, uma relação entre elas pode ser desenhada.

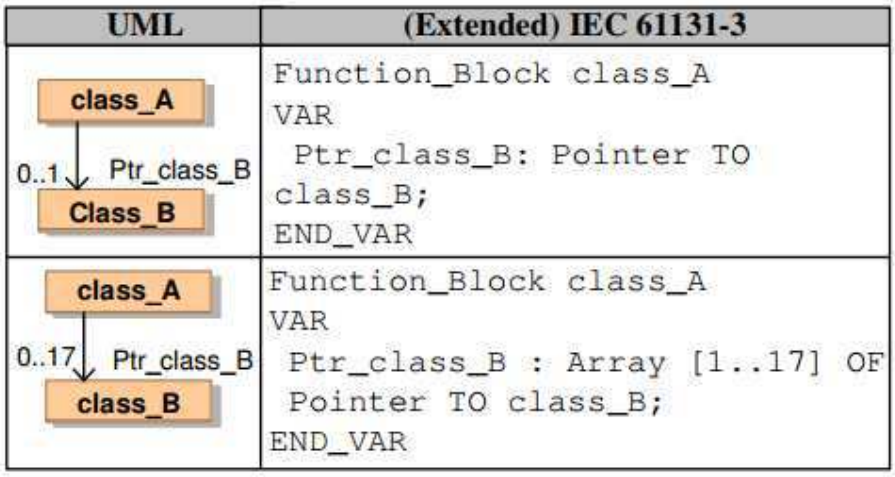
2.3.2 Mapeamento de relações entre blocos para diagrama de classes UML

O diagrama UML também é útil para visualizar e especificar relações entre classes (ou blocos de funções). Essas relações são expressas por bordas diretas ou indiretas entre as classes. Para as relações em UML, tem-se:

- Associação;
- Composição;
- Generalização;
- Implementação.

As relações de associação expressam um "conhecimento"uni ou bidirecional entre diferentes classes. Elas são representadas por uma seta simples direcionada ou não direcionada em um diagrama UML e mapeada para um ponteiro ou um "*array* de ponteiros. Ao lado desta seta, são fornecidas informações de cardinalidade e um nome para a função de relação. A cardinalidade de uma associação expressa a quantidade máxima de associação que pode ser estabelecida (usando esta relação). O limite superior da cardinalidade corresponde ao tamanho da matriz para o ponteiro.

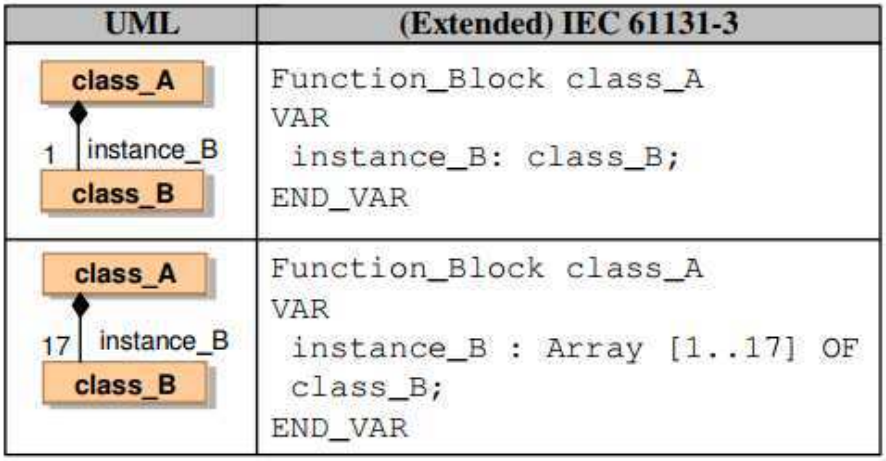
Figura 4 – Associação



Fonte: (WITSCH; VOGEL-HEUSER, 2009)

As relações de composição expressam contenção ou instanciação de uma variável do tipo correspondente. A relação é desenhada como uma aresta com um diamante preenchido no lado da classe que a contém. Na extremidade oposta da relação também é dado um nome de função e cardinalidade. Caso a cardinalidade seja maior que um, uma matriz de elementos (com um tamanho da cardinalidade) é adicionada à classe que a contém.

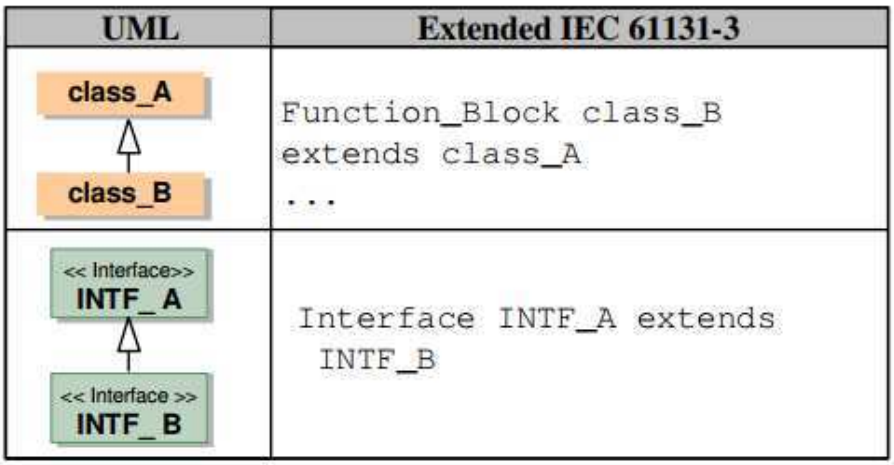
Figura 5 – Composição



Fonte: (WITSCH; VOGEL-HEUSER, 2009)

A relação de generalização/herança é expressa usando uma seta com sua ponta fechada. A seta aponta para classe superior. Nessa relação, nem a cardinalidade nem um nome para o papel são necessários. Além disso, esse tipo de relação pode existir entre classes ou entre interfaces.

Figura 6 – Herança

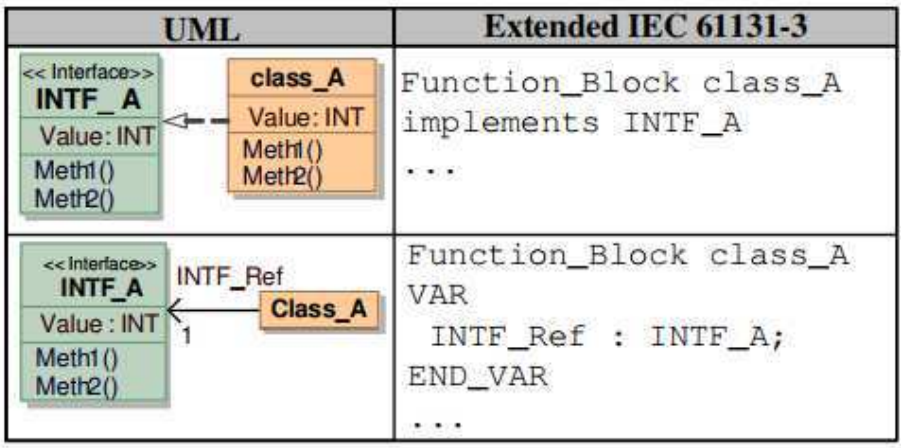


Fonte: (WITSCH; VOGEL-HEUSER, 2009)

A relação de implementação entre uma classe e a interface implementada é expressa por uma seta com uma linha tracejada.

Já para expressar a instância de uma variável, a qual é definida por uma interface como um tipo de dado, usa-se a relação de associação. A semântica de um ponteiro é dada implicitamente porque variáveis instanciadas de um tipo de interface são tratadas como ponteiros.

Figura 7 – Implementação



Fonte: (WITSCH; VOGEL-HEUSER, 2009)

3 Arquitetura Orientada a Serviço

3.1 Contextualização

O estudo (BLOCH; FAY; HOERNICKE, 2016) analisa diferentes abordagens da Arquitetura Orientada a Serviço (SOA) para a automação de processos modulares. Nesse contexto, a ideia de modularização consiste em dividir um sistema de produção em diferentes módulos, onde cada um realiza pelo menos uma função específica dentro do processo.

Uma planta modular é composta por dispositivos, instrumentos e controles de processo, organizados em segmentos chamados de módulos. Cada módulo executa uma parte específica do processo. Nesse sentido, um módulo pode ser entendido como um elemento tecnicamente e organizacionalmente limitado que realiza uma tarefa específica.

Para garantir o controle de cada módulo, sugere-se a utilização de um controle baseado em estado. Cada módulo deve transmitir uma descrição do seu estado para o PCS (Sistema de Controle do Processo), o qual controla os módulos por meio de mudanças de estado.

Nessa solução, uma nova estrutura para automação de processos modulares é requerida. Então o fornecimento de serviços como funções de processo é sugerido. Nele, cada serviço completa uma parte da função de controle dentro de um módulo.

Para que as descrições abstratas dos serviços sejam utilizadas, cada serviço encapsulado deve ter sua própria máquina de estados.

Nesse cenário, implementar um controle baseado em serviços exige que a informação gerada dentro dos módulos seja transmitida ao de forma adequada.

Cada serviço deveria ser controlado por mudanças de estados desencadeadas pelo controlador. E ,para isso, uma resposta de cada módulo é necessária. Onde, nessa respostas, é conter o estado ativo de cada serviço e os possíveis estados futuros e por quais comando eles são acessíveis.

A fonte dos comandos é o PCS, o qual deve orquestrar todos os serviços de todos os módulos para um processo geral.

3.2 Arquitetura

De forma geral, a SOA descreve um conjunto de componentes "caixa preta", os quais provêm serviços por meio de uma interface que é implementada usando serviços de comunicação. (BLOCH; FAY; HOERNICKE, 2016)

Para esclarecer o conceito de "serviço", o World Wide Web Consortium (W3C) define um serviço como um serviço web. Esse serviço consiste em um conjunto de componentes que possuem uma descrição de interface, podem ser publicados, descobertos e executados.

Essa definição pode ser aplicada no contexto de automação de processo modular, porque as funções de processo podem ser invocadas e descobertas e operar após a invocação.

Portanto, um serviço pode ser entendido como um mecanismo que permite o acesso a uma ou mais capacidades fornecidas por uma entidade e utilizadas por outras, por meio de uma interface que segue o padrão de solicitação-resposta. No contexto da automação de processos modulares, os serviços representam funções de processo encapsuladas, como o controle de temperatura, movimentação de materiais ou operação de máquinas. Esses serviços podem ser invocados e descobertos e operam conforme a solicitação.

3.3 Modelos de Referência

Os modelos de referência oferecem uma visão padronizada de como os serviços devem ser estruturados, organizados e integrados dentro de um sistema SOA. Eles definem não apenas a arquitetura geral do sistema, mas também as interações entre os diferentes componentes, as diretrizes para a composição de serviços e as melhores práticas para a implementação. Portanto, esses modelos são de vital importância no desenvolvimento de sistemas que utilizam SOA. Neste contexto, três abordagens diferentes são:

- Modelo de Referência OASIS;
- *Service-oriented cross-layer infrastructure for distrinuted smart embedded devices* (SOCRADES);
- Implementação da SOA com blocos de função.

3.3.1 OASIS

Dentro do Modelo de Referência OASIS, um serviço é definido como um "mecanismo que habilita o acesso a uma ou mais capacidades". Esses serviços são fornecidos por uma entidade e utilizados por outras, sendo o acesso implementado por meio de uma interface de serviço, que segue o padrão de solicitação-resposta.

Três conceitos fundamentais que estão envolvidos em uma interação com serviços são definidos:

- Visibilidade;

- Interação;
- Efeito no mundo real;

O conceito de visibilidade explicita que para performar uma interação entre provedor e consumidor do serviço, os dois precisam se "enxergar". Assim, os requisitos de consciência, alcançabilidade e vontade precisam ser cumpridos.

A consciência descreve o conhecimento dos mútuo do provedor e consumidor da existência da outra parte. Para atingir isso, descrições de serviço são necessárias para que o consumidor saiba se o serviço é capaz de cumprir a tarefa necessária.

A vontade, por sua vez, refere-se à disposição das entidades de cooperar. Na automação modular, assume-se que os serviços estejam dispostos a cooperar, mas pode haver restrições, como requisitos de segurança funcional ou conflitos com outros serviços, que podem impedir a execução de um serviço.

Já a alcançabilidade descreve a possibilidade de comunicação entre provedor e consumidor do serviço. Ela é assumida no contexto de automação modular de processo modular pela rede e os protocolos de comunicação, como, por exemplo, o OPC-UA.

A interação entre provedor e consumidor de serviços envolve dois modelos principais. De um lado, o modelo de informação caracteriza os possíveis dados que podem ser trocados entre provador e consumidor. Esse modelo é separado em estrutural e semântico.

- Estrutural: define o tipo e a forma do dado em diferentes níveis estruturais.
- Semântica: descreve toda informação necessária para interpretação do dado.

Por outro lado, o modelo de comportamento contêm todo o conhecimento sobre as ações realizadas dentro de cada serviço. Os aspectos temporais do processo também são parte do modelo de comportamento. Essa informação está contida em dois modelos mais detalhados: modelo de ação e modelo de processo.

O modelo de ação define as ações executadas pelos serviços. Em que, nesse contexto, essas ações podem ser entendidas como comandos para atuadores.

O modelo de processo contém as relações e propriedades de cada ação. No contexto da automação modular, o processo se refere à execução de uma ação e muda conforme as ações são alteradas.

Além disso, o modelo de processo também oferece espaço para definir as relações e propriedades necessárias para a orquestração de serviços. Na automação de processos modulares, essa orquestração é crucial, pois os serviços podem ser executados em paralelo ou em sequência.

Por exemplo, se um serviço de enchimento depende de um serviço de mistura, a orquestração garante que a mistura seja realizada antes que o enchimento ocorra.

Os efeitos no mundo real de um serviço podem incluir três elementos diferentes:

- Informação solicitada;
- Mudança de estado de uma entidade;
- Uma combinação dos dois.

Os consumidores invocam um serviço por uma dessas razões. Por exemplo, serviços que exibem valores, como leituras de sensores, podem ser vistos como resultado da primeira categoria, onde o efeito é meramente informacional.

3.3.2 SOCRADES

O projeto *Service-oriented cross-layer infrastructure for distributed smart embedded devices* (SOCRADES) foi uma iniciativa europeia de pesquisa e desenvolvimento voltada para a aplicação do paradigma de manufatura baseado em SOA. A principal base desse projeto está no uso de serviços web para integrar dispositivos distribuídos em sistemas industriais inteligentes.

Nessa abordagem, o foco está em 4 tópicos:

- Arquitetura orientada a serviço;
- Rede sem fio de sensores e atuadores;
- Integração com a empresa;
- Engenharia e gerenciamento de sistemas;

Dentro da estrutura SOA do SOCRADES, a inteligência do sistema é implementada por agentes físicos que são incorporados em dispositivos inteligentes. Esses dispositivos provêm serviços web como uma interface para comunicação entre si.

Todas as unidades estão cooperando entre si no mesmo nível hierárquico, assim um nível de controle de processo mais alto não é necessário. Essas unidades autônomas operam cooperativamente, e cada uma é inteligente e proativa.

Para gerenciar e compor todos os serviços web oferecidos por esses dispositivos, uma unidade de orquestração é conectada diretamente à rede. Essa unidade provê serviços web de orquestração, permitindo que os diferentes dispositivos cooperem de maneira coordenada e eficiente.

3.3.3 Implementação da arquitetura orientada a serviço com blocos de função

Nessa abordagem, serviços são vistos como elementos atômicos encapsulados que podem ser invocados por outros serviços ou por mensagens de solicitação-resposta. Nessa visão, é necessário o acesso a dados do processo como sinais do atuador e sensor, o qual é implementado também por serviços.

Uma característica essencial dessa implementação é que a interconectividade entre todos os dispositivos no sistema é necessária, porém, os serviços não precisam ser alocados em um hardware específico. Isso oferece flexibilidade no design do sistema, permitindo que serviços sejam movidos ou realocados sem impactar a funcionalidade global.

Conexão entre blocos de função representam mensagens, portanto representam a comunicação entre serviços. Essa comunicação é separada entre tipos de mensagem e parâmetros, onde mensagens são representadas por conexões de eventos e parâmetros por conexões de dados como descrito no padrão.

Essa abordagem se origina da automação de fábrica, onde as funcionalidades são decompostas em tarefas mais simples e básicas. Por exemplo, uma função como "empurrar peça de trabalho"é considerada uma função básica. Cada uma dessas funções é implementada como um serviço atômico. Para construir funcionalidades mais complexas, vários serviços atômicos podem ser combinados e orquestrados por uma entidade de coordenação central ou por meio de uma coreografia de serviços autônoma.

Em sistemas SOA, uma entidade de controle superior pode invocar serviços por meio de mensagens. Uma vez que um serviço é invocado, ele pode, por sua vez, invocar subserviços em uma ordem específica, criando uma cadeia de ações coordenadas.

A orquestração de serviços é específica para configuração de serviços implementados no componente e é executada por uma máquina de estado específica na entidade de orquestração.

4 Desenvolvimento Baseado em Componente

4.1 Contextualização

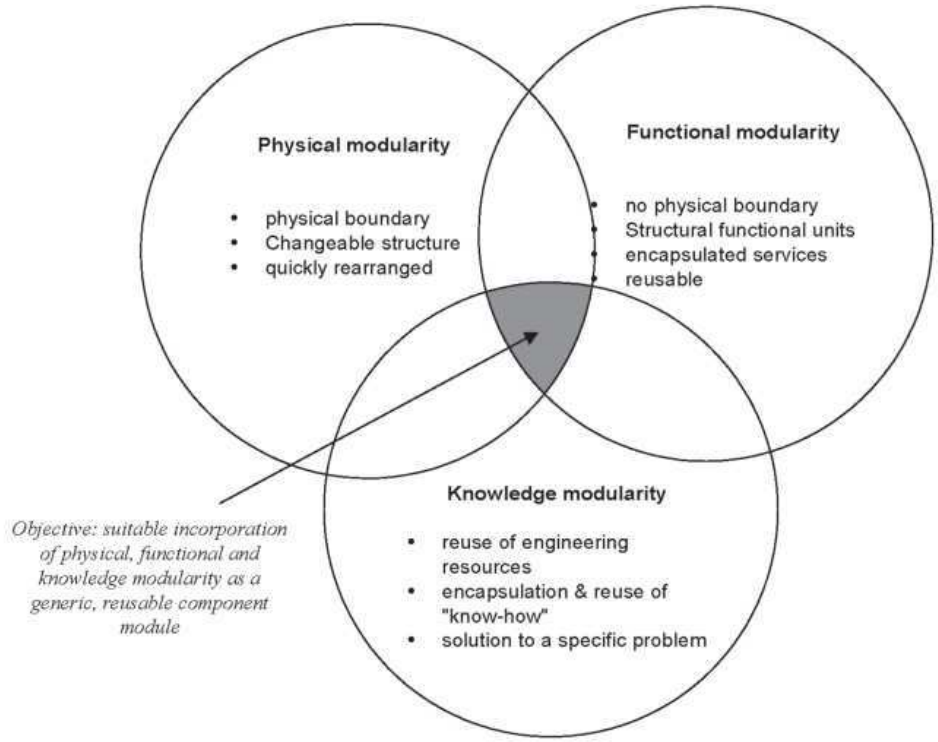
De acordo com (LEE; HARRISON; WEST, 2005), a emergência de um paradigma de programação ágil é impulsionada pela crescente necessidade das empresas de responder de forma rápida e flexível às demandas do mercado. Nesse cenário, torna-se essencial um sistema de manufatura capaz de ser reconfigurado rapidamente para acomodar mudanças tanto no produto quanto no processo.

Em reposta aos requisitos para uma manufatura ágil, existe uma necessidade por um sistema de controle que possa:

- Fornecer meios e capacidades para suportar flexibilidade, reconfiguração e reutilização;
- Permitir que novas funções e tecnologias de processo possam ser integradas rapidamente no sistema.

A modularização surge como uma abordagem eficaz para lidar com as complexidades e mudanças na manufatura. Esse conceito pode ser explorado sob três diferentes perspectivas: física, funcional e de conhecimento.

Figura 8 – Modularização



Fonte: (LEE; HARRISON; WEST, 2004)

Na modularização física, os blocos físicos ou módulos são usados para compor o produto final. E a principal características de tais sistemas modulares é o projeto de componentes de hardware genéricos que podem ser rearranjados para compor diferentes configurações de produção.

A modularização funcional estabelece uma ligação entre os módulos e as unidades de serviço, comumente chamadas de "blocos de função". Nesse contexto, o sistema de controle decompõe tarefas complexas em funções menores e mais simples, que podem ser armazenadas em bibliotecas de funções e reutilizadas em diferentes partes do sistema, promovendo a eficiência e a consistência do código.

A modularização de conhecimento, por sua vez, vê os módulos como "portadores de conhecimento". Esses módulos carregam informações sobre o problema, o contexto e a solução, e podem ser reutilizadas em diferentes fases de desenvolvimento ou em diferentes sistemas. Essa abordagem facilita a reutilização de recursos de engenharia e acelera o desenvolvimento de novos projetos, baseando-se no conhecimento já adquirido.

Lee, Harrison e West (2005) cita que para atender aos requisitos de agilidade, hardware de automação, software e serviços de controle é necessário que o conceito de modularidade não se limite a nenhum dos três aspectos na construção de uma unidade modular, mas que seja uma integração entre eles, como mostrado na figura 8.

Cengic, Ljungkrantz e Akesson (2006) avalia alguns requisitos para que uma ar-

quitetura baseada em componentes realmente atinja seus objetivos:

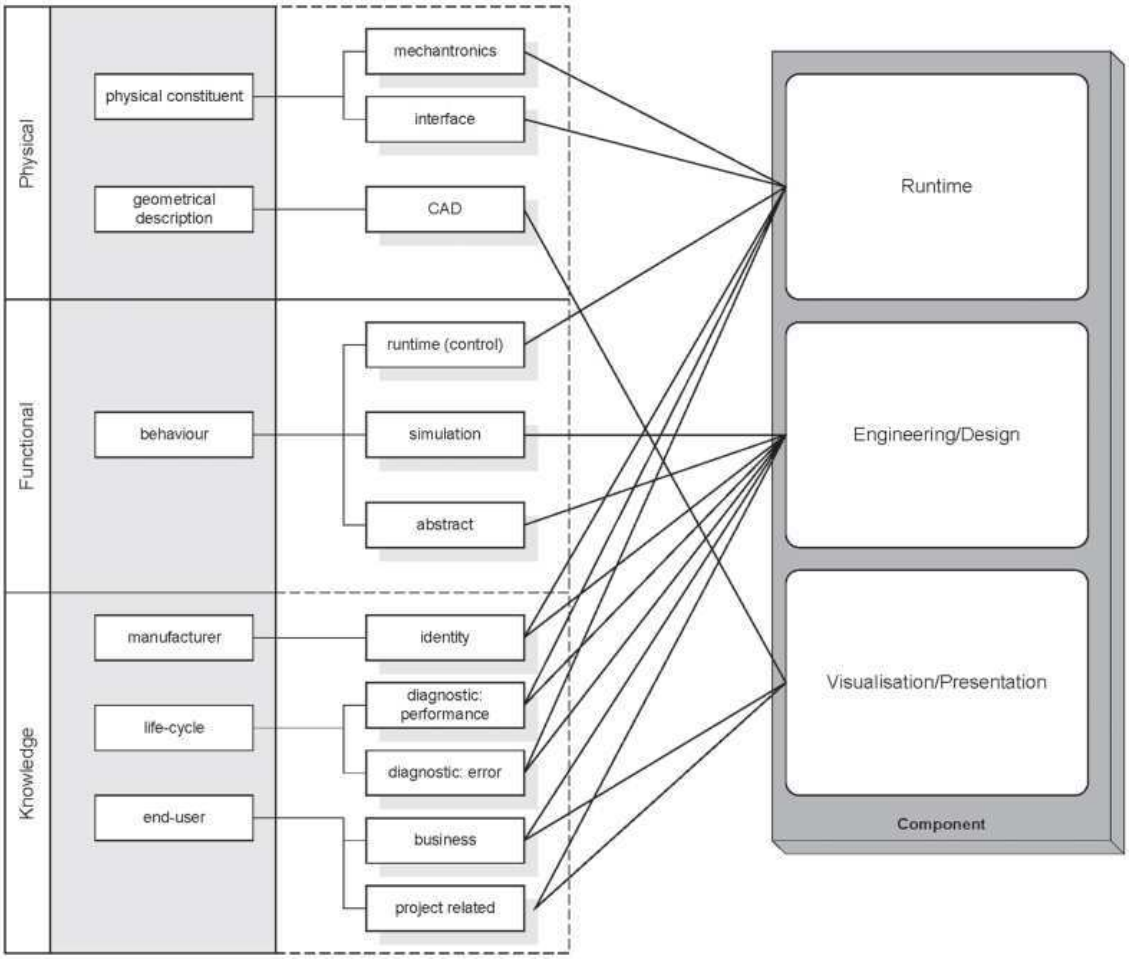
- Reutilização de Código - Componentes de automação devem prover mecanismos para reutilização de código de maneira eficiente;
- Extensibilidade - Os componentes de automação devem ser extensíveis em seu nível de abstração. Isso significa que qualquer componente deve poder servir como ponto de partida para novos componentes de automação, permitindo que sua funcionalidade seja estendida pelo usuário;
- Organização em Bibliotecas - Os componentes de automação devem ser organizáveis em bibliotecas de componentes, facilitando o acesso a componentes já verificados e prontos para uso;

4.2 Conceitos

O Desenvolvimento Baseado em Componente(CBD) é baseado na suposição de que "há uma suficiência de similaridade em muitos sistemas de software grandes para justificar o desenvolvimento de componentes reutilizáveis para explorar e satisfazer essa similaridade". (LEE; HARRISON; WEST, 2004)

Nessa abordagem, um componente encapsula os atributos físicos e funcionais necessários, conhecimento e características do sistema ao qual será integrado para que seja possível fazer essa integração imediata com outros componentes, e, assim, compor qualquer sistema.

Figura 9 – Mapeamento de Atributos



Fonte: (LEE; HARRISON; WEST, 2004)

Na figura 9, temos um mapeamento dos atributos de modularidade para um componente. Nesse sentido, um componente pode ser dividido em três aspectos:

- *Runtime*;
- *Engineering*;
- Visualização;

O aspecto de *Runtime* representa a visão de controle de um componente. Ele contém elementos físicos (como dispositivos de automação, controladores locais e circuitos de interface) e as funções de controle que determinam o comportamento do componente durante sua execução. Em sistemas de controle distribuído, esses aspectos são fundamentais para garantir que o componente possa operar de forma eficiente.

Por sua vez, o aspecto de *Engineering* prove informação sobre um componente que é necessária para o design e verificação do processo. Nesse mesmo aspecto, o comportamento de controle é representado de forma abstrata para que seja possível projetar o

sistema baseado em componente e validá-lo em um ambiente de engenharia. Esse ambiente fornece uma plataforma para que os integradores de sistemas projetem e construam um sistema de manufatura com base em uma biblioteca de componentes, sem ter que lidar com detalhes de baixo nível.

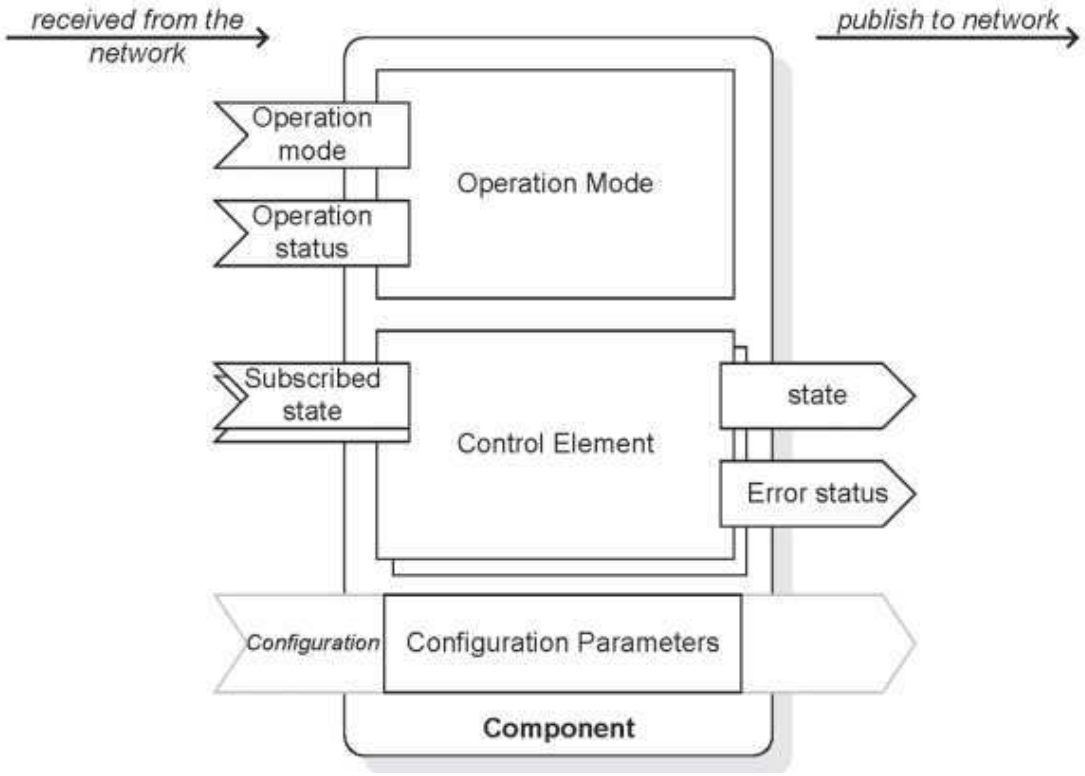
O aspecto de visualização, diferentemente, traz informações sobre o componente que são necessárias para propósitos de apresentação.Ele inclui dados de CAD, os quais podem ser usados para criar modelos visuais do componente, bem como descrições de como ele se comporta. Com essas informações, o mesmo pode ser representado e visualizado em 2D ou 3D.

O conceito usado para entender a programação baseada em componente, no entanto, tem seu foco no aspecto de *Runtime* e *Engineering*.

Logo, um componente é definido como uma unidade de software que pode ser utilizada para implementar aplicações de software de controle. Eles podem ser hierarquicamente incorporados a outros componentes para criar novos, ou combinados para formar aplicações baseadas em componentes. Eles encapsulam funcionalidades específicas e interações que podem ser reutilizadas em diferentes configurações de sistemas de controle, promovendo a modularidade, reutilização de código, e facilitando a manutenção e reconfiguração de sistemas de controle distribuído (CENGIC; LJUNGKRANTZ; AKESSON, 2006)

Sob a perspectiva lógica, um componente é formado por três partes, a interface, comportamento interno e invocação. A interface do componente estabelece os "conectores"de comunicação, chamados de interfaces de rede, que permitem ao componente enviar dados para o sistema e receber dados de outros componentes. A Figura 11 ilustra essas interfaces de rede, onde o componente recebe mensagens através das interfaces de entrada localizadas à esquerda do diagrama e envia mensagens de estado e status pelas interfaces de saída à direita.

Figura 10 – Interface de Rede



Fonte: (LEE; HARRISON; WEST, 2005)

O comportamento interno define como o componente funciona internamente. As operações básicas dos dispositivos de automação podem ser programadas pelo fornecedor do componente e encapsuladas dentro dele. O comportamento encapsulado lida com o controle, o qual é representado por máquinas de estados finitas. Essas máquinas de estado controlam as transições entre diferentes estados operacionais, como "motor ligado"e "motor desligado", com base em condições específicas.

A invocação de um componente determina quando ele pode realizar suas operações. A condição para que uma operação seja invocada depende dos estados combinados de outros componentes no sistema.

Além disso, um componente pode encapsular um ou mais dispositivos de automação (como um sensor ou atuador), que vai ter seu comportamento controlado por um elemento de controle, fazendo com que o dispositivo de automação se comporte de acordo com o estado desse elemento na máquina de estados.

Um exemplo de associação entre um dispositivo de automação e um elemento de controle pode ser observado em uma esteira transportadora automatizada utilizada em uma linha de produção industrial. Nesse caso, o dispositivo de automação seria o motor da esteira, responsável por movimentar fisicamente a esteira e transportar itens de um ponto a outro na linha de produção. Esse é o dispositivo físico que executa a tarefa de

movimentação. O controle desse motor é realizado por um elemento de controle, cuja função é monitorar e comandar o funcionamento do motor da esteira. Esse elemento de controle gerencia a ativação e desativação do motor, ao responder a eventos como sinais de sensores que indicam a presença de itens na esteira. O comportamento do elemento de controle é modelado por uma máquina de estados finitas, que define estados como "motor desligado", "motor em operação"e "motor em pausa". A transição entre esses estados ocorre com base em condições predefinidas, como a detecção de um item pela esteira ou a necessidade de interromper o funcionamento para evitar sobrecargas.

Por exemplo, o motor da esteira é ativado pelo elemento de controle quando um sensor detecta a presença de um item pronto para ser transportado. O elemento de controle mantém o motor em operação até que outro sensor indique que o item chegou ao destino

Dessa forma, o dispositivo de automação (motor) realiza a tarefa física, enquanto o elemento de controle gerencia seu comportamento de forma inteligente e autônoma, com base em uma máquina de estados finitas, garantindo que a operação seja realizada de forma eficiente e adaptável às condições da linha de produção.

5 Metodologia

A metodologia foi dividida em etapas que permitem explorar o conceito de desenvolvimento de cada um desses paradigmas.

Após a etapa de apresentação teórica de cada, uma aplicação, seguindo as características apresentadas anteriormente, será mostrada com o objetivo de avaliar suas características e a alcançabilidade de seus objetivos.

Para o desenvolvimento dessas aplicações, é necessário usar ferramentas que englobam as normas IEC 61131-3 (para OOP) e a IEC 61499 (para CDB e SOA). Além disso, para simular as plantas industriais a serem controladas, foi um utilizado o software Factory IO com o objetivo de simular esses ambientes. Para isso, foi escolhida uma aplicação simples apenas com o objetivo de avaliar os pontos positivos e negativos de cada um.

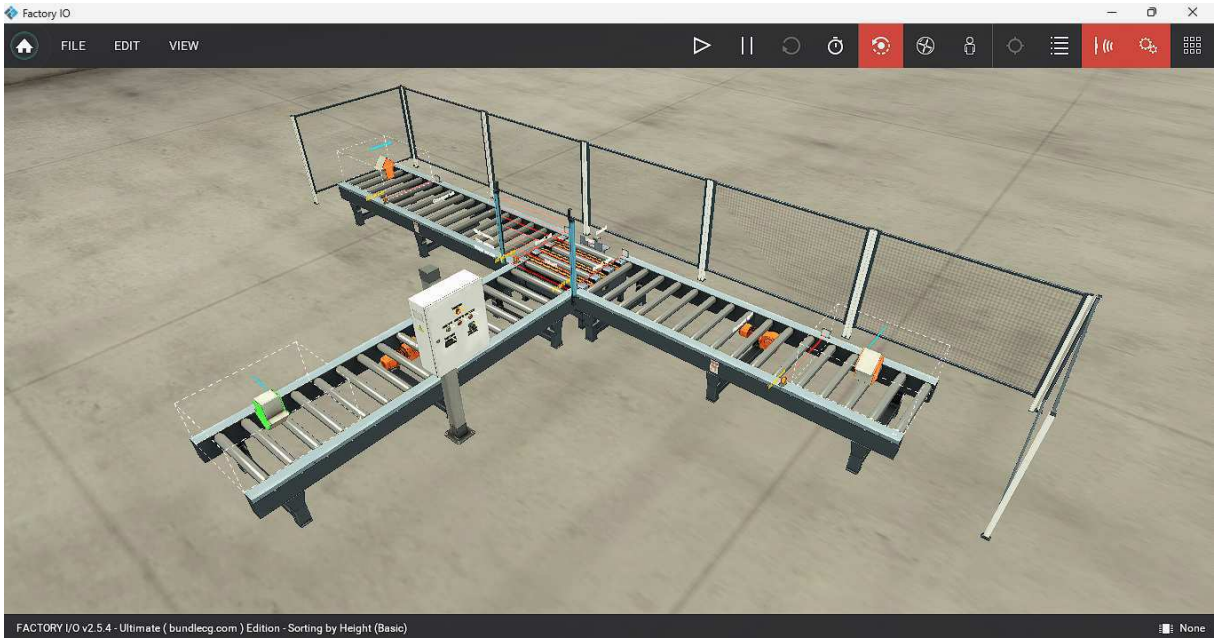


Figura 11 – Planta no Factory IO

Essa planta faz a separação de caixas com base em seus tamanhos. A lista de sensores e atuadores presentes pode ser vista na figura 12, e seu mapeamento, nas figura 13 e 14.

Figura 12 – Lista de Sensores e Atuadores

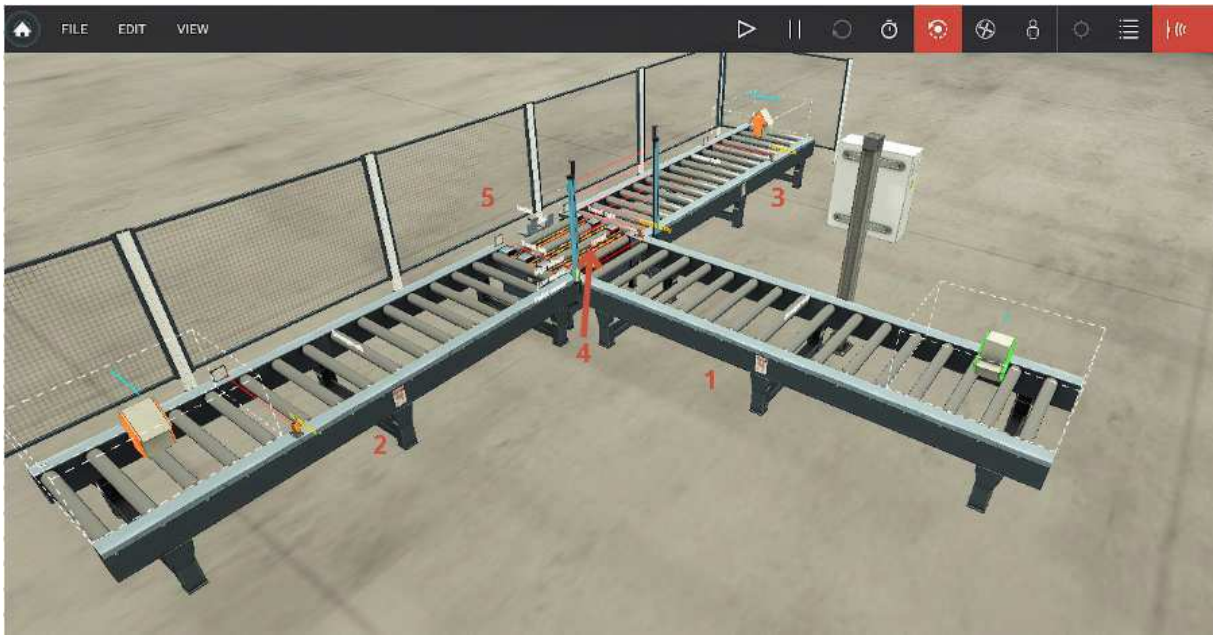


Figura 13 – Mapeamento dos atuadores

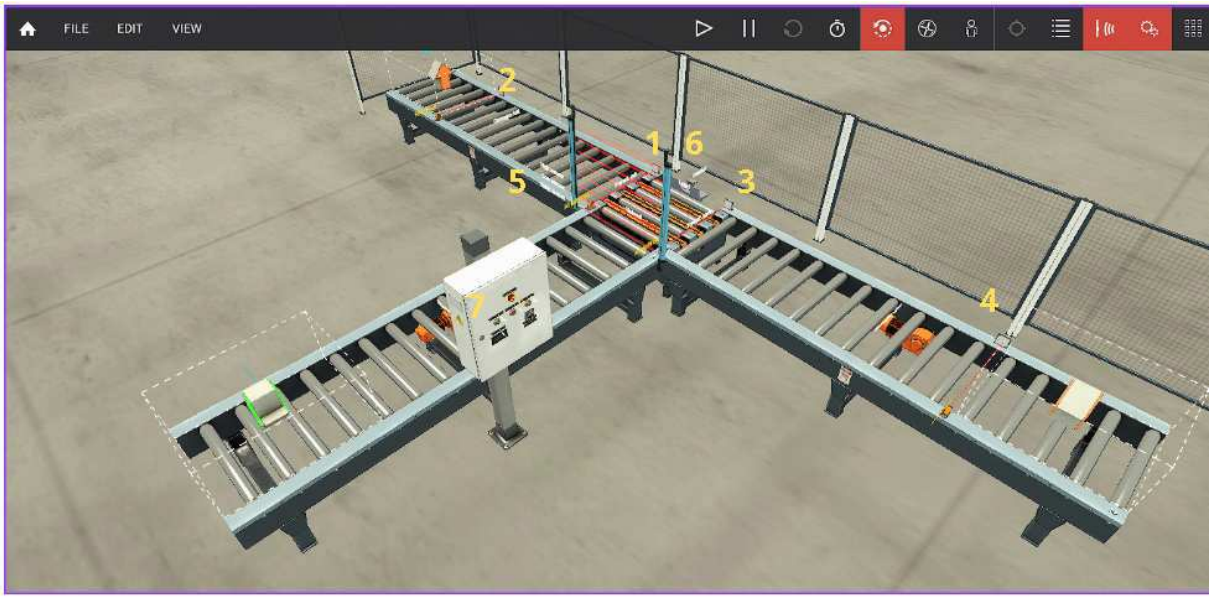


Figura 14 – Mapeamento dos sensores

Ao chegar uma caixa na esteira, e acionando o botão *Start*(7), a esteira(1) começa a se mover. Após isso, a caixa passa pelo sensor(5) que identifica o tamanho da caixa, nesse momento também é acionado o *Load*(4) que leva a caixa para o atuador *Transf. left/Transf. right* que, a partir da informação do sensor 5, decide para onde a caixa vai. Porém, esse atuador só é ativado quando a caixa ativa o sensor *Loaded*(6), e é nesse momento que a esteira de entrada(1) e o *Load*(4) param para que o atuador seja acionado, junto da esteira da esquerda ou direita(2/3).

Quando a caixa passa pelo sensor de entrada de da direita ou esquerda (1/3), a esteira principal(1) é acionada novamente, e todo processo se repete.

5.1 Programação Orientado a Objeto

A aplicação desenvolvida com a programação orientada a objeto foi projetada no software CodeSys, que está de acordo com a norma IEC 61131-3. Logo, permite a programação orientada a objeto. Além disso, o próprio ambiente do CodeSys disponibliza um CLP virtual para que seja possível rodar a aplicação.

Para comunicação, é usado o protocolo OPC-UA, para fazer a conexão do CLP virtual, que está com a aplicação, com a planta a ser controlada. Dessa forma, é possível ter acesso ao estado dos sensores e enviar os comandos para os atuadores.

Para essa aplicação, duas interfaces são criadas com o objetivo de que qualquer tipo de classe que implemente essas interfaces tenha métodos obrigatórios para implementar. As interfaces criadas são esteira e atuador, a esteira cria 3 métodos obrigatórios para qualquer classe que a implemente, enquanto que o atuador, 2.

Figura 15 – Interfaces

A partir das interfaces, cria-se classes que as implementam, o que pode ser visto pela palavra chave "IMPLEMENT", na declaração da classe.

Figura 16 – Classes

Como nota-se pela imagem 19, os métodos das interfaces, que inicialmente são métodos vazios, quando implementados por uma classe precisam ter alguma aplicação na prática.

```
FUNCTION_BLOCK esteira_FB IMPLEMENTS esteira_
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
    estado:BOOL;
END_VAR
```

Figura 17 – Criação da classe implementando a interface

Cada instância da classe esteira, tem uma variável interna chamada "estado"que é do tipo "BOOL"e dita o estado da esteira, podendo ser ligada ou desligada.

Para mudar o estado dessa variável, os métodos "ligar"e "desligar"são invocados dentro do programa, esses métodos, por sua vez, chamam o método "mudar status", passando o sinal correto, para efetivamente mudar o estado da variável.

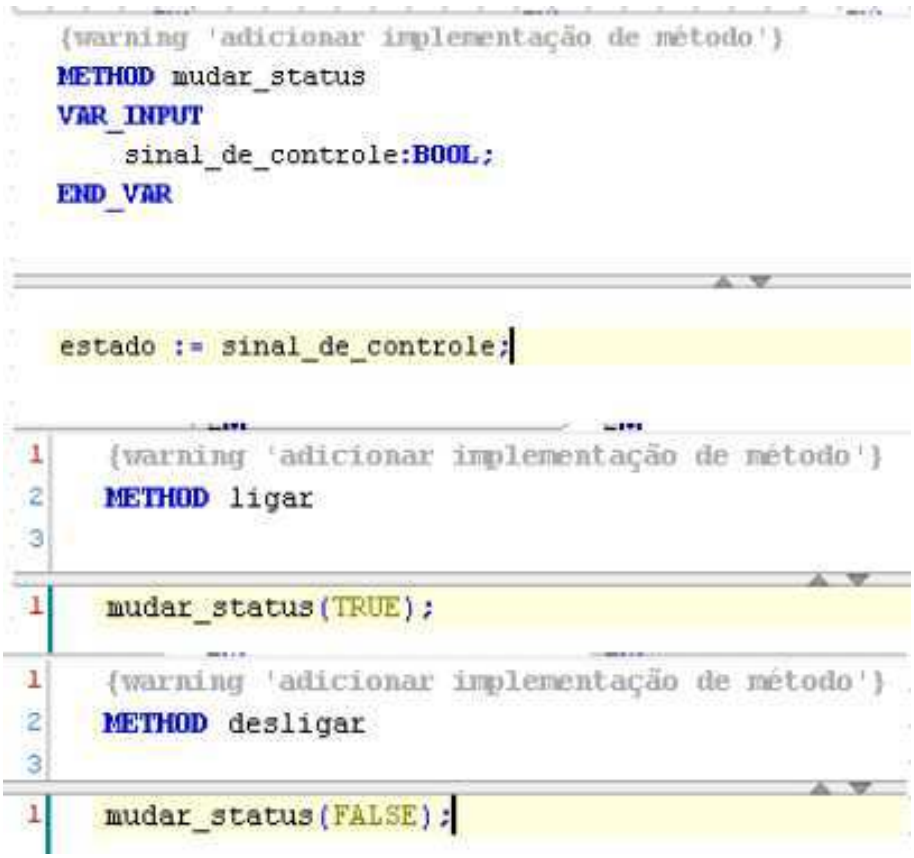


Figura 18 – Implementação dos métodos de esteira

O mesmo procedimento é feito para a classe "atuador", a diferença é que as mudanças na variável interna são feitas diretamente nos métodos "ligar"e "desligar".

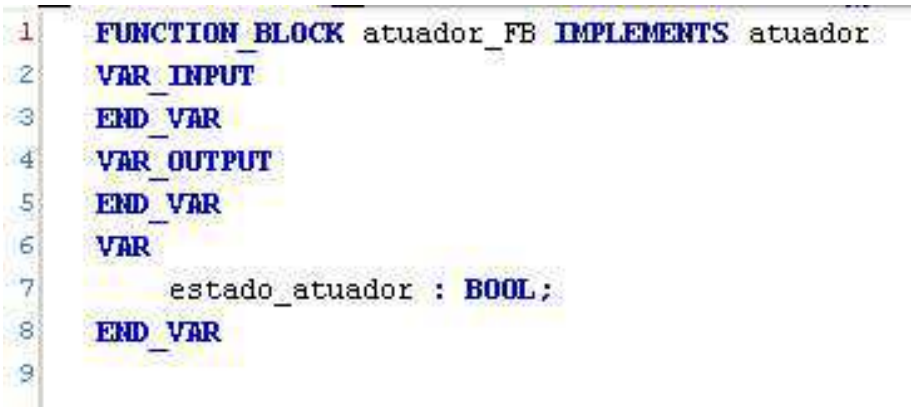


Figura 19 – Classe Atuador

```
1  {warning 'adicionar implementação de método'}
2  METHOD desligar

1  estado_atuador := FALSE;

1  {warning 'adicionar implementação de método'}
2  METHOD ligar
3

1  estado_atuador := TRUE;
```

Figura 20 – Métodos da classe atuador

A partir das classes, a programação do CLP é facilitada, pois, mesmo numa aplicação simples como a mostrada acima, há uma maior clareza no programa, um exemplo disso é que no lugar de mudar o valor de uma variável diretamente no programa no ligamento/desligamento de uma esteira, ao chamar o método ligar/desligar fica claro qual ação está sendo feita, o que facilita na leitura e na clareza do programa.

```
1   PROGRAM PLC_PRG
2   VAR
3       // Esteiras
4       esteira1 : esteira_FB;
5       esteira_right : esteira_FB;
6       esteira_left : esteira_FB;
7       load       : esteira_FB;
8
9       //Atuadores
10      atuador_right : atuador_FB;
11      atuador_left : atuador_FB;
12
```

Figura 21 – Criação das instâncias

Na imagem 21, é possível ver a instância das classes esteira e atuador, na *main* do programa.

```
14
15      load.desligar();
16
17      esteira1.desligar();
18      esteira_right.desligar();
19      esteira_left.desligar();
20
21      atuador_right.desligar();
22      atuador_left.desligar();
```

Figura 22 – Uso dos métodos

Nesse trecho de código, é visto como é feito a chamada dos métodos de cada instância no código principal.

5.2 Desenvolvimento Baseado em Componente

As aplicações baseadas em componente são feitas seguindo a norma IEC 61499, visto que não segue necessariamente uma lógica sequencial, dessa forma, o uso dos eventos é essencial para programação.

Como o CodeSys é destinado a aplicações sob a norma IEC 61131-3 não é possível utilizá-lo para o desenvolvimento segundo esse paradigma. Assim, o software utilizado neste caso é o "4DIAC-IDE", cujo ambiente permite a programação sob a norma IEC 61499, possibilitando a criação de blocos para simular os componentes da aplicação. Além do IDE, existe também o "4diac FORTE", o qual permite rodar a aplicação no próprio 'Windows', ou em algumas outra plataforma, como a "RaspBerry".

A comunicação da aplicação com a simulação no Factory IO é feita com o protocolo OPC UA, porém para que a aplicação rodando no FORTE tenha a possibilidade de se comunicar com esse protocolo é necessário fazer sua "*build*"incluindo a biblioteca OPC-UA.

Outro fator importante na comunicação é que existe uma questão com a comunicação entre o Factory IO e o 4DIAC. O que acontece é que para fazer a comunicação, é usado uma lógica de "PubSub", onde é possível se inscrever a variáveis dentro da planta com o bloco "Subscriber", e também é possível publicar variáveis no servidor pelo bloco "Publisher". Porém, o bloco "Subscriber"deveria ativar o evento "IND"apenas quando a variável mudasse seu valor, mas o que acontece na prática é que o próprio Factory IO escreve a todo momento no servidor, o que faz com que esse evento seja "gatilhado"a todo

momento. Isso é um detalhe que necessita de atenção, pois muda a lógica de programação, e dificulta a implementação do paradigma.

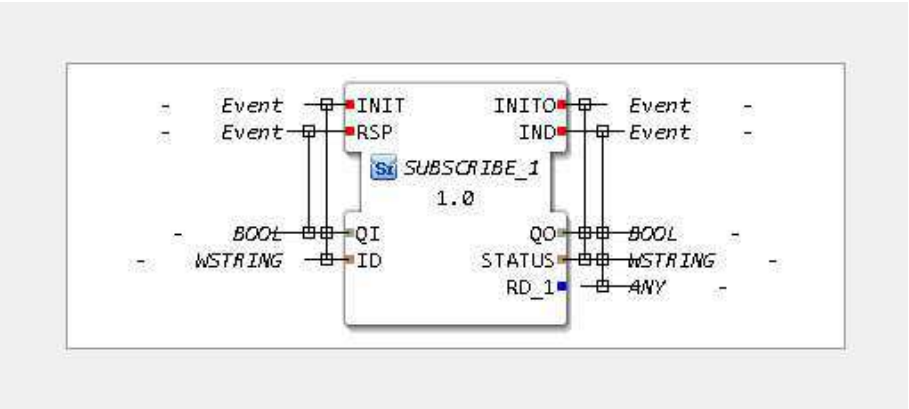


Figura 23 – *Subscriber*

Nessa aplicação, o componente é construído como um bloco de função dentro da IDE. Os componentes construídos para essa aplicação têm sua própria máquina de estado que funciona como o elementos de controle, já o dispositivo de automação varia de acordo com o componente. Para essa aplicação, foram criados quatro tipos de blocos de função que funcionam como os componentes:

- Esteira
- Esteira Lateral
- Atuador
- *Load*

A divisão entre as duas esteiras se deve ao fato de que o controle para o ligamento e desligamento entre elas é feito dentro do elemento de controle(da máquina de estados) pelo fato do evento "ind"ser acionado a todo momento, e como a lógica de ativação das duas difere entre si, foi necessário a criação de dois tipos diferentes. Porém, dessa forma, também é possível visualizar outro elemento importante desse paradigma, que é o fato de que um componente pode servir de base para criação de outro, assim como foi feito para o tipo esteira lateral que deriva de esteira.

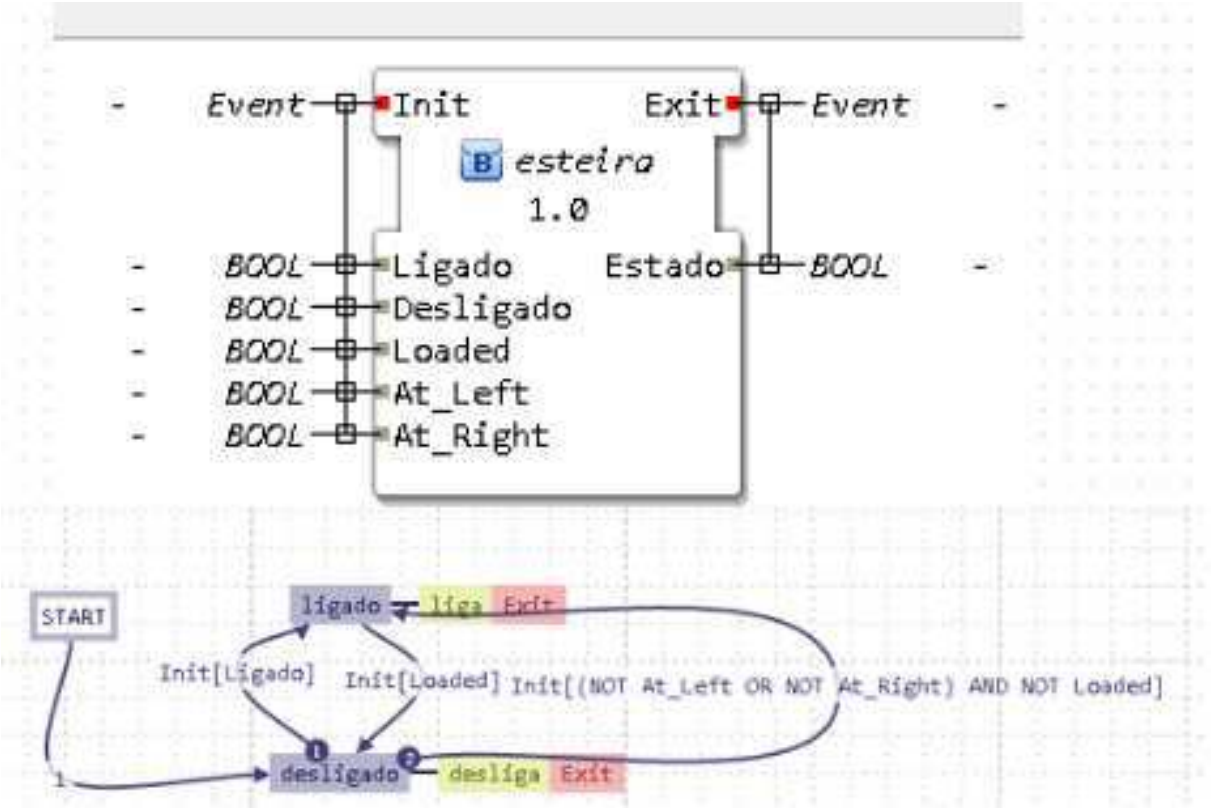


Figura 24 – Componente esteira e máquina de estado

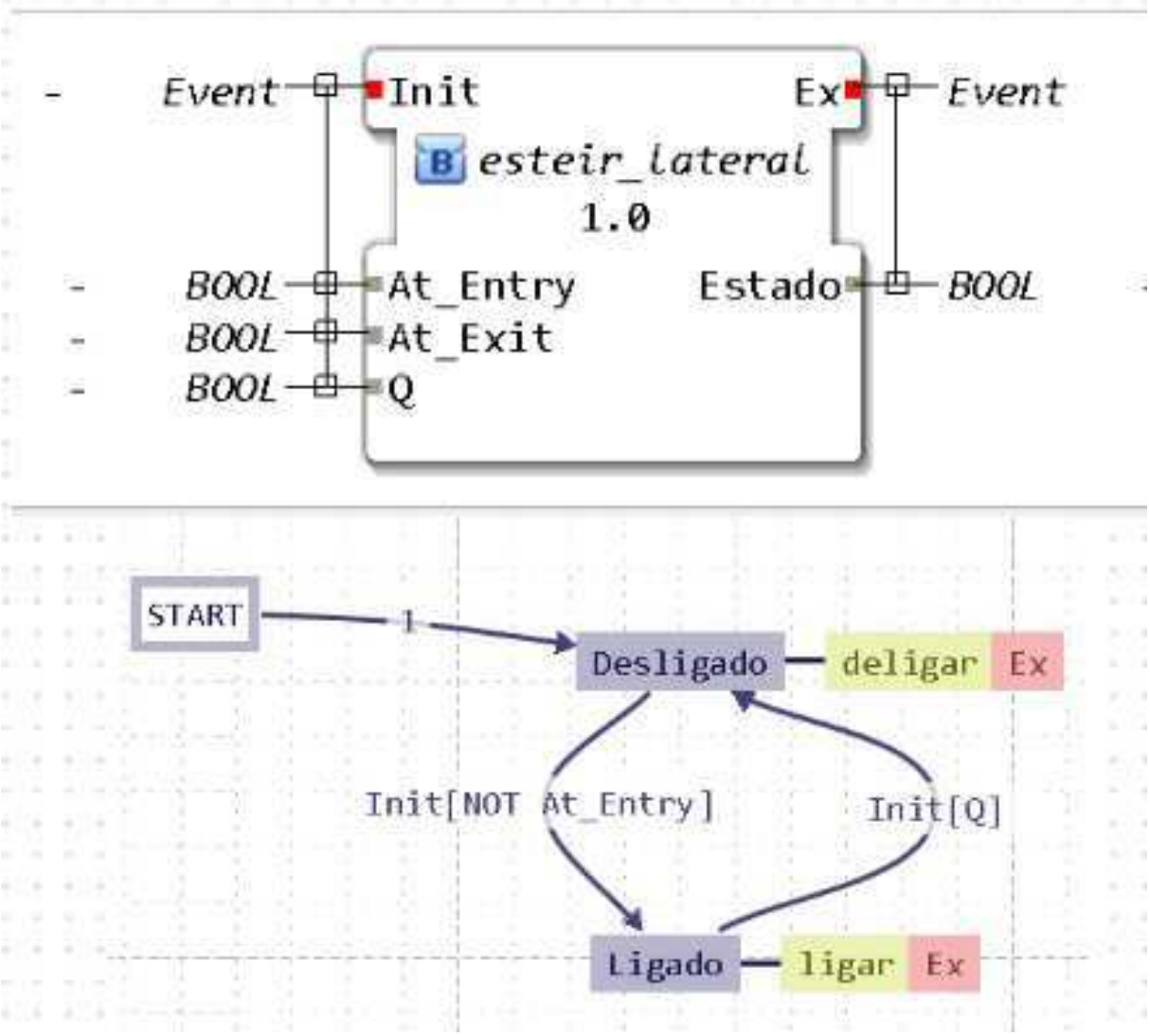


Figura 25 – Componente esteira lateral e máquina de estado

O atuador funciona com base no sinal dos sensores "*Low*"e "*High*", a partir disso, ele escolhe para onde vai mandar a caixa. O *Load*, por sua vez, funciona quando é preciso "puxar"uma caixa para o atuador, momento esse que é quando passa pelos sensores de "*Low*"e "*High*", e se desliga quando o sensor "*Loaded*"é acionado.

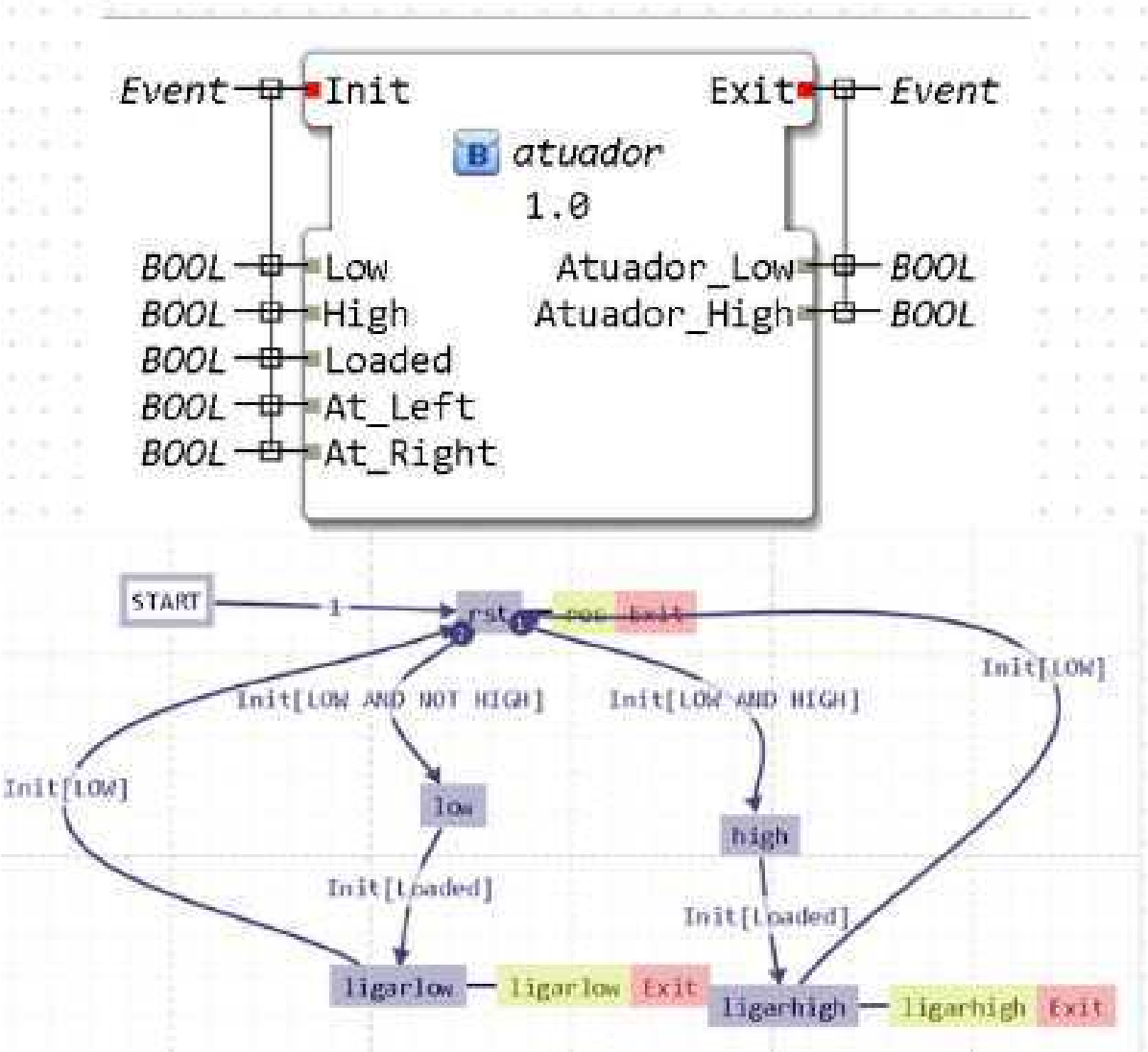


Figura 26 – Componente Atuador

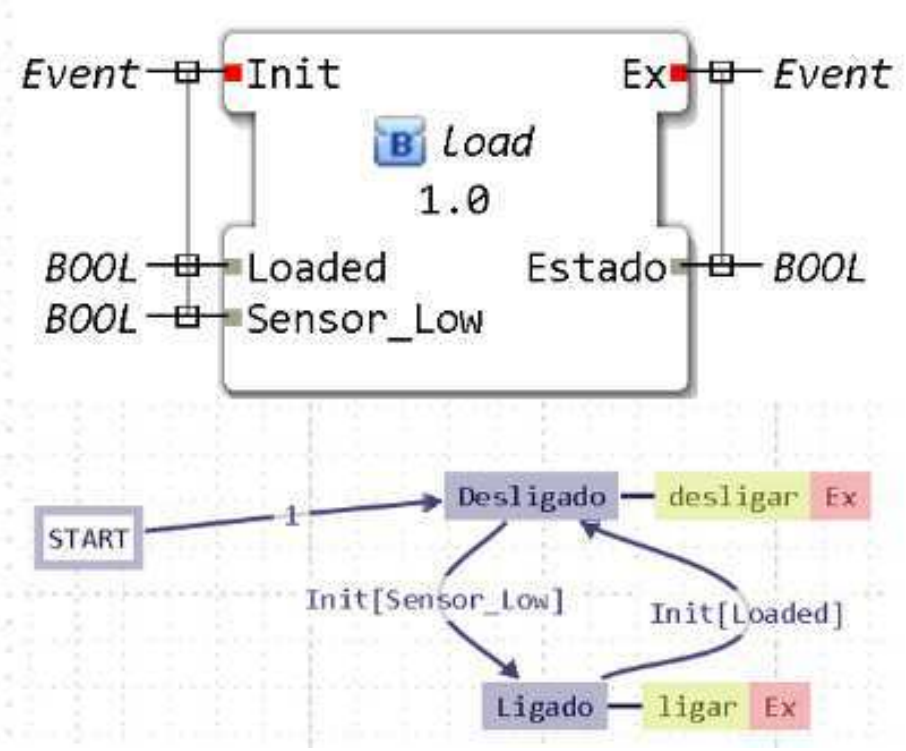


Figura 27 – Componente *Load*

Outro elemento que pode ser entendido como um componente são os sensores dessa aplicação, porém não foi necessário criar um novo tipo, já que o *Subscriber* já faz sua função de ler os valores dos sensores na planta quando a aplicação está rodando, e está "desligado"quando a aplicação não está rodando. Logo, pode-se entender que esse elemento é controlado por uma máquina de estado com os estados de "medindo"quando a planta está ligada e "sem medir", caso contrário.

Depois da criação dos blocos e da implementação deles num diagrama de blocos, é dado o *Deploy* da aplicação para o FORTE, o qual pode foi feito tanto para uma máquina *Windows* quanto para uma *RaspBerry*. E a partir daí, é feito o mesmo processo necessário para rodar a aplicação no CodeSys que é ligar as tags que foram disponibilizadas pela comunicação OPC nos seus locais adequados dentro do Factory IO.

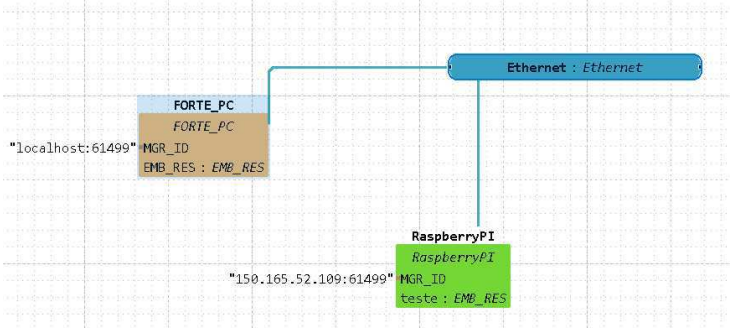
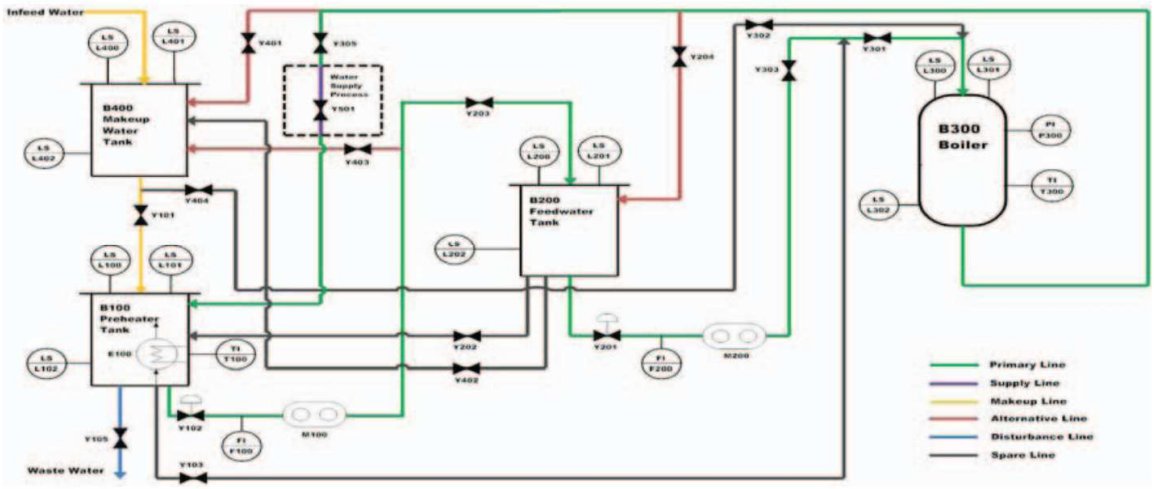


Figura 28 – Conexão do Computador e da Raspberry

Figura 29 – Sistema de Produção



Fonte: (DAI et al., 2014)

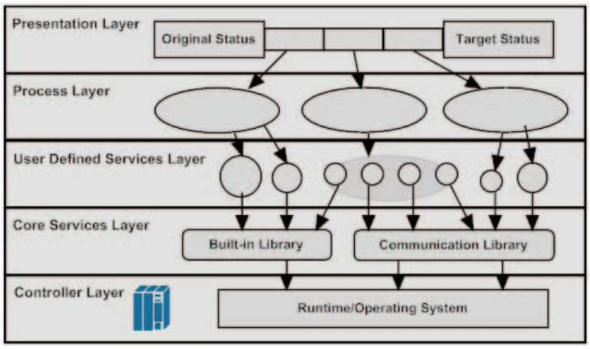
5.3 Arquitetura Orientado a Serviço

Um estudo de caso proposto por (DAI et al., 2014) usa um sistema de produção de calor como mostrado na figura 29.

O processo se dá da seguinte maneira:

- Água fria entra para o tanque de água de reposição (B400);
- A água do tanque de reposição vai alimentar o tanque de pré aquecimento (B100) pela válvula de controle Y101 quando o nível da água do tanque de pré aquecimento estiver baixa;
  - Existe um aquecedor no tanque de pré aquecimento que esquenta a água.
- A água quente vai ser bombeada para o tanque de alimentação (B200) por meio da válvula de controle Y102 e da bomba M100;
- Quando a caldeira (B300) estiver pronta, a água do tanque de alimentação será bombeada para a caldeira através da válvula Y201 e da bomba M200;
  - Tem um indicador de pressão e um sensor de temperatura na caldeira para evitar sobre aquecimento e sobre pressão;
  - Quando qualquer emergência acontecer, a válvula Y204 é aberta para diminuir a pressão e a temperatura na caldeira;
  - Em cada tanque, há um conjunto de indicadores de nível alto (Lx01 - Analógico, Lx00 - Digital) e baixo (Lx02 - Analógico) para medir o nível e detectar condições anormais

Figura 30 – Camadas de Serviços



Fonte: (DAI et al., 2014)

- A água pressurizada no tanque da caldeira será fornecida ao cliente através da válvula de abastecimento Y305. E, por fim, a água pode ser descarregada abrindo a válvula Y105

No domínio da computação, SOA é apresentado em uma arquitetura em camadas de serviços compostos. Um conceito parecido pode ser ser aplicado no domínio da automação industrial.

A quinta camada é a camada de controle, que consiste do sistema operacional e do ambiente de execução para os controladores. Já a quarta camada, camada de serviços principais, inclui *bulit-in functions e handlers* de comunicação.

A camada de serviços definidos pelo usuário(terceira camada) tem funções ou blocos de funções que são desenvolvidas nessa camada, que podem atuar como consumidores de serviços,invocando serviços dos provedores da camada de serviços principais.

A segunda camada é a camada do processo, que contém informações dos processos físicos individuais controlados por funções de automação, como enchimento de água.

A camada de apresentação forma os processos individuais em um sistema completo, usando diagramas de sequência. Essa camada possui conhecimento do sistema de automação inteiro.

5.3.1 Camadas

Na camada de serviços principais, funções integradas como o evento cíclico. inversor e acesso de I/O como Entradas Analógicas e Saídas Analógicas e Entradas Digitais e Saídas Digitais são implementadas.

Já na camada de serviços definidos por usuários, há quatro tipos de serviços a medição de sensor analógico, controle de atuador analógico, medição de sensor digital e o controle de atuador digital. Nessa aplicação, os blocos de função de medição analógico e

digital fazem leituras de proximidade de nível de água, sensores de temperatura, sensores de pressão e sensores de medição de fluxo, e geram alarmes para cada variável do processo.

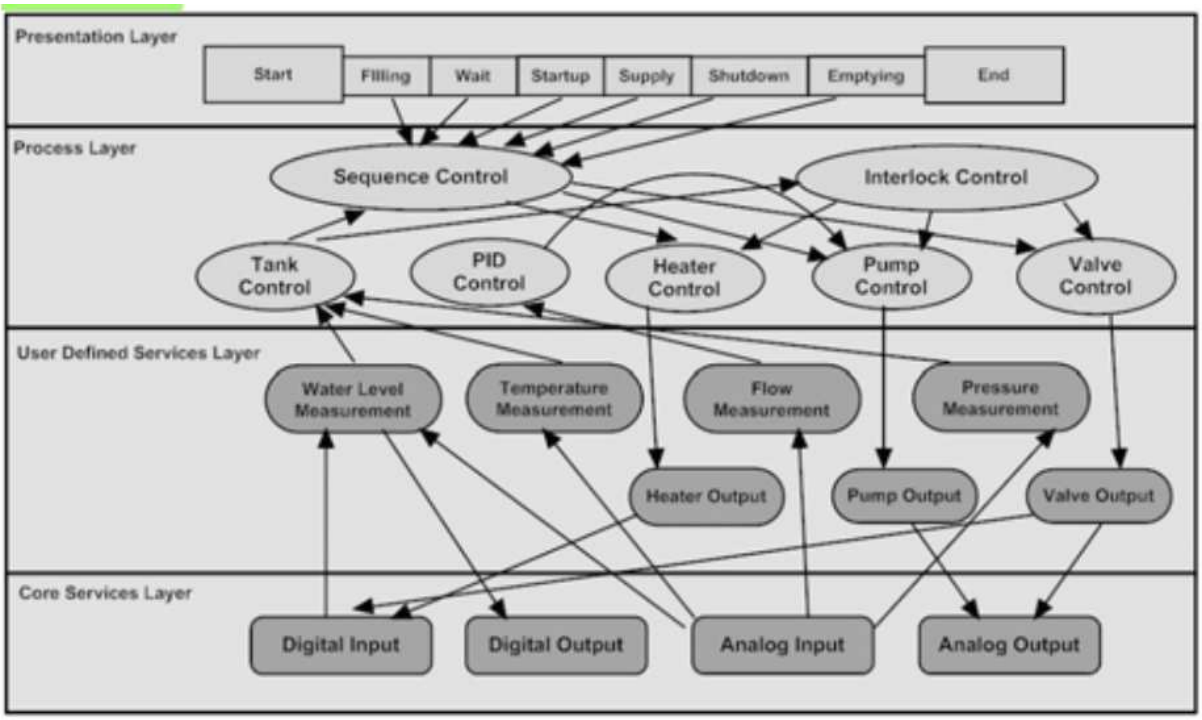
Cada bloco de função de serviço de atuador recebe duas entradas uma do nível superior (modo automático) e uma da Interface Homem-Máquina (HMI, modo manual).

Na camada de processo, os serviços são agrupados por funcionalidades dentro do processo, controle de tanque, controle PID, controle de aquecedor, controle de bomba e controle de válvula. O serviço de controle do tanque coleta sinais de alarme dos serviços de medição de sensores e gera status para o tanque, como se o tanque está pronto para entrada e saída de água, se o tanque pode ser aquecido e se o tanque está sobre pressurizado. O serviço de controle do PID lê os valores de medidas do serviço de medição de vazão e recalcula o valor de controle para os serviços de controle da válvula e bomba. Os serviços de controle do aquecedor, bomba e válvula checam que o valor de controle está dentro do range e produzem comandos de saída para os serviços de controle dos atuadores.

Dois outros serviços são definidos o serviço de controle de sequência e serviço de intertravamento. O serviço de controle de sequência controla todas as válvulas, aquecedores e bombas baseado no *feedback* coletado dos serviços de controle de tanque na camada de processo.

A sequência de funcionamento é definida em cinco passos: Enchimento, Espera, Inicialização, Desligamento e Esvaziamento.Finalmente, tem-se a visão geral da configuração do sistema HPP.

Figura 31 – Visão Geral dos Serviços



Fonte: (WITSCH; VOGEL-HEUSER, 2009)

6 Análise dos Resultados

A partir dos experimentos, evidenciou-se que a utilização de paradigmas de programação em sistemas de automação industrial traz uma série de benefícios essenciais, dado o aumento da complexidade das plantas industriais e as crescentes exigências de mercado.

Contudo, a implementação de cada um dos paradigmas permite uma compreensão mais profunda de suas capacidades e limitações, bem como das vantagens e desvantagens que cada abordagem oferece em relação às outras.

No que diz respeito à OOP, uma de suas principais vantagens é a capacidade de encapsular dados e métodos dentro de uma classe, o que permite uma organização clara e modular do sistema. Essa abordagem facilita a reutilização de código e a clareza na estrutura do programa, pois as funcionalidades são abstraídas em classes que representam os componentes do sistema. No entanto, a lógica de atuação dos métodos ou a modificação do valor de propriedades ainda precisa ser definida manualmente dentro do código, o que pode aumentar a complexidade da implementação.

Por outro lado, no caso do CBD, a lógica de atuação de cada componente já está encapsulada dentro do próprio bloco de função. Isso gera uma maior simplicidade no momento da implementação, pois o desenvolvedor precisa apenas conectar os sinais de entrada e saída necessários para o funcionamento do componente. Dessa forma, o CBD permite uma implementação mais rápida e menos suscetível a erros na configuração dos componentes.

Em relação SOA, ela se mostrou uma abordagem eficaz para a modularização e integração de sistemas em ambientes de automação distribuída. A SOA foi implementada com foco na separação de funcionalidades em serviços encapsulados, que podem ser invocados e orquestrados por meio de uma comunicação padronizada, como o protocolo OPC-UA. Ela promove uma reconfigurabilidade ágil e uma manutenção simples.

Portanto, a escolha do paradigma depende das necessidades do projeto, sendo que cada abordagem oferece vantagens que podem ser mais adequadas para diferentes contextos de automação industrial.

7 Conclusão

A escolha adequada de paradigmas de programação é essencial para o sucesso na automação industrial, especialmente diante da crescente complexidade das plantas e das demandas de mercado por maior flexibilidade e eficiência. A aplicação da Programação Orientada a Objetos (OOP) mostrou ser uma abordagem poderosa, fornecendo organização e modularidade ao encapsular dados e métodos em classes. No entanto, essa abordagem pode aumentar a carga de trabalho de desenvolvimento devido à necessidade de definir manualmente a lógica dos métodos.

Por outro lado, o Desenvolvimento Baseado em Componentes (CBD) provou ser mais eficaz em termos de simplicidade na implementação, pois encapsula a lógica de controle dentro dos próprios componentes. Essa abordagem reduz a chance de erros de configuração e simplifica o processo de desenvolvimento. No entanto, a diversidade de componentes pode aumentar a complexidade do sistema como um todo.

A Arquitetura Orientada a Serviços (SOA) foi eficaz na modularização de sistemas distribuídos, permitindo a invocação de serviços através de comunicação padronizada e promovendo a flexibilidade e a manutenção de sistemas. No entanto, a orquestração e sincronização de serviços entre os módulos exigem maior esforço no planejamento da arquitetura do sistema.

Assim, conclui-se que cada paradigma tem suas vantagens e desvantagens, e a escolha do mais adequado depende das necessidades específicas de cada projeto de automação industrial. A modularidade e a reconfigurabilidade oferecidas pelo CBD e pela SOA são essenciais em sistemas dinâmicos e distribuídos, enquanto a OOP é mais vantajosa quando se necessita de maior controle detalhado sobre as funcionalidades internas do sistema.

Referências

BLOCH, H.; FAY, A.; HOERNICKE, M. Analysis of service-oriented architecture approaches suitable for modular process automation. In: IEEE. *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. [S.l.], 2016. p. 1–8. Citado na página 13.

CENGIC, G.; LJUNGKRANTZ, O.; AKESSON, K. A framework for component based distributed control software development using iec 61499. In: IEEE. *2006 IEEE Conference on Emerging Technologies and Factory Automation*. [S.l.], 2006. p. 782–789. Citado 2 vezes nas páginas 19 e 22.

DAI, W. et al. Service-oriented distributed control software design for process automation systems. In: IEEE. *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. [S.l.], 2014. p. 3637–3642. Citado 3 vezes nas páginas 1, 38 e 39.

LEE, S.; HARRISON, R.; WEST, A. A component-based control system for agile manufacturing. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, Sage Publications Sage UK: London, England, v. 219, n. 1, p. 123–135, 2005. Citado 3 vezes nas páginas 18, 19 e 23.

LEE, S.-M.; HARRISON, R.; WEST, A. A. A component-based distributed control system for assembly automation. In: IEEE. *2nd IEEE International Conference on Industrial Informatics, 2004. INDIN'04. 2004.* [S.l.], 2004. p. 33–38. Citado 3 vezes nas páginas 19, 20 e 21.

WERNER, B. Object-oriented extensions for iec 61131-3. *IEEE Industrial Electronics Magazine*, IEEE, v. 3, n. 4, p. 36–39, 2009. Citado 2 vezes nas páginas 2 e 3.

WITSCH, D.; VOGEL-HEUSER, B. Close integration between uml and iec 61131-3: New possibilities through object-oriented extensions. In: IEEE. *2009 IEEE Conference on Emerging Technologies & Factory Automation*. [S.l.], 2009. p. 1–6. Citado 7 vezes nas páginas 2, 8, 9, 10, 11, 12 e 40.