UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO, DE CIÊNCIAS EXATAS E EDUCAÇÃO
DEPARTAMENTO DE ENG. DE CONTROLE, AUTOMAÇÃO E COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Eduardo Klein Fabeni

Aplicação de Design Patterns na automatização de projetos de comando e controle de transformadores

Blumenau
2023

Eduardo Klein Fabeni

Aplicação de Design Patterns na automatização de projetos de comando e controle de transformadores

Trabalho de Conclusão de Curso de Graduação em Engenharia de Controle e Automação do Centro Tecnológico, de Ciências Exatas e Educação da Universidade Federal de Santa Catarina como requisito para a obtenção do título de Engenheiro de Controle e Automação.
Orientador: Prof. Mauri Ferrandin, Dr.

Blumenau
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Fabeni, Eduardo Klein
Aplicação de design patterns na automatização de projetos de comando e controle de transformadores / Eduardo Klein Fabeni ; orientador, Mauri Ferrandin, 2023.
85 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Blumenau, Graduação em Engenharia de Controle e Automação, Blumenau, 2023.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Automatização de processos. 3. Design patterns. 4. Projetos de circuitos de comando e controle. I. Ferrandin, Mauri. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Eduardo Klein Fabeni

Aplicação de Design Patterns na automatização de projetos de comando e controle de transformadores

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Engenheiro de Controle e Automação" e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação.

Blumenau, 07 de 03 de 2023.

Banca Examinadora:

Documento assinado digitalmente
Mauri Ferrandin
Data: 20/06/2023 11:49:42-0300
CPF: ***.580.869-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Mauri Ferrandin, Dr.
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Carlos Roberto Moratelli
Data: 20/06/2023 13:21:22-0300
CPF: ***.999.369-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Carlos Roberto Moratelli, Dr.
Universidade Federal de Santa Catarina

Documento assinado digitalmente
Maiquel de Brito
Data: 20/06/2023 13:38:56-0300
CPF: ***.657.150-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Maiquel de Brito, Dr.
Universidade Federal de Santa Catarina

RESUMO

Esse trabalho apresenta o planejamento, desenvolvimento e resultados do projeto de automatização de processos em projetos de circuitos de comando e controle para transformadores, realizado na WEG Transmissão e Distribuição de Blumenau e aplicado às demais unidades do Brasil. O projeto teve como objetivo a identificação de processos que pudessem ser automatizados e que refletissem no aumento de produtividade do setor de projetos, de forma a acompanhar o rápido crescimento da empresa nos últimos anos. Antigas automatizações que já eram utilizados em produção foram estudadas, refatoradas, corrigidas e aprimoradas, através da criação de uma solução coesa que abrigasse todas as automatizações e utilizando as boas práticas de desenvolvimento mais comuns na indústria, como o paradigma de orientação a objetos, arquitetura da aplicação com camadas de responsabilidade únicas e a utilização dos design patterns mais adequados às diferentes automatizações. Foi definida uma nova metodologia para a liberação de atualizações e de novas automatizações, bem como para a aplicação automática das atualizações para todos os usuários. Foi realizado também o levantamento das demandas de usuários e de seus respectivos requisitos, com o posterior planejamento, execução e liberação. Após a finalização do projeto foi possível verificar uma facilitação no desenvolvimento de novas automatizações, maior velocidade na correção de erros e de implementação de melhorias, menor tempo e menor quantidade de suportes prestados aos usuários, menos erros em tempo de execução e maior autonomia dos projetistas na correção de erros de projeto

Palavras-chave: automatização de processos; design patterns; projetos de circuitos de comando e controle.

ABSTRACT

This work presents the planning, development and results of the process automation project in command and control circuit designs for transformers, carried out at “WEG Transmission and Distribution” in Blumenau and applied to other unities in Brazil. The objective of the project was to identify processes that could be automated and that would increase the productivity of the project sector, in order to keep up with the rapid growth of the company in recent years. Old automations that were already used in production were studied, refactored, corrected and improved, through the creation of a cohesive solution that housed all the automations and using the most common good development practices in the industry, such as the object-oriented paradigm, architecture of the application with unique layers of responsibility and the use of the most suitable design patterns for the different automations. A new methodology was defined for releasing updates and new automations, as well as for automatically applying updates to all users. A survey of user demands and their respective requirements was also carried out, with subsequent planning, execution and release. After the completion of the project, it was possible to verify a facilitation in the development of new automations, greater speed in correcting errors and implementing improvements, less time and less amount of support provided to users, fewer errors in runtime and greater autonomy for designers in the correction of project errors.

Keywords: process automation; design patterns; command and control circuit designs.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A automação de processos é tema inseparável da indústria no contexto atual, tornando muito difícil imaginar qualquer atividade que não possua pelo menos algum grau de automação e que seja realizada inteiramente de forma manual. Há décadas, softwares como o Excel permitem a utilização de funções para a manipulação de dados de maneira automática e até a criação de macros que permitem o desenvolvimento de automações ainda mais complexas. Olhando para o presente, exemplos como esse podem inclusive ser vistos como datados, considerando a grande variedade de softwares, robôs e inteligências artificiais que não só fazem parte do cotidiano das diversas indústrias adequadas aos princípios da Indústria 4.0, mas que têm seus limites expandidos a cada nova pesquisa.

As vantagens proporcionadas por essas tecnologias são tão expressivas que tomam o espaço de empresas que insistem em não se atualizar e que mantém os mesmos processos de anos atrás. Alguns dos benefícios são o aumento da eficiência garantido pela realização automática de tarefas, permitindo que os recursos humanos sejam melhor aproveitados em outras tarefas; a redução de erros decorrente de um processo completamente padronizado e, portanto, muito menos suscetível a erros; redução de custos ou aumento dos ganhos causados pelo maior tempo aproveitado em tarefas que agregam mais valor aos produtos e serviços finais; melhor capacidade de avaliação da qualidade dos processos, sendo possível medir o tempo gasto e a qualidade dos resultados entregues por operações automatizadas, investir em melhorias incrementais que, ao serem realizadas de dezenas a milhares de vezes, acarretam em ganhos expressivos, entre outros.

A tendência de utilizar sistemas cada vez mais inteligentes aumenta, e a expansão de sua utilização ainda pode trazer ganhos que nem se apresentam de forma tão clara atualmente.

A WEG, uma das maiores empresas do Brasil, faz questão de investir em pesquisa e desenvolvimento de forma a acompanhar essa tendência, e muitas vezes até sair à frente. Decorrente do grande crescimento que a empresa vem apresentando nos últimos anos, era essencial que seus processos começassem a ser atualizados para acompanhar o ritmo acelerado da companhia, em todas as áreas.

Mais especificamente, esse trabalho apresenta os esforços realizados para a automatização dos procedimentos de projetos de comando e controle de transformadores, ligados diretamente ao produto final da empresa. Os painéis de comando e controle são projetados para garantir que os transformadores operem de maneira segura e eficiente, obtendo os dados através de sensores instalados na estrutura do transformador para controlar variáveis como temperatura, tensão e corrente, além de contar com equipamentos de segurança como disjuntores e chaves seccionadoras. O projeto desse tipo de circuito depende do conhecimento técnico do projetista dadas as especificações exigidas pelas normas vigentes e pelas empresas contratantes, mas também de procedimentos administrativos, que envol-

vem documentações, levantamento dos materiais utilizados, cadastro das informações do projeto em sistemas de gerenciamento, entre outros, que são importantes para uma boa administração de projetos mas que exigem tempo dos projetistas sem necessariamente agregar valor ao produto final. Com a automatização desses processos é possível diminuir o tempo de projeto, principalmente das atividades ligadas à parte administrativa, disponibilizar mais tempo para o desenvolvimento da parte técnica do projeto, o que ajuda a produzir melhores projetos ou a realização de projetos mais complexos, evitar erros e garantir uma maior padronização das entregas. Nesse sentido, não só a automatização de processos é importante mas também as boas práticas que permitem o sucesso de sua implementação.

Dessa forma, os capítulos a seguir apresentam o planejamento do projeto, as metodologias e ferramentas utilizadas, a implementação do projeto com mais ênfase em seus detalhes técnicos, os procedimentos criados para a implantação das melhorias e para a obtenção de dados para a análise dos benefícios obtidos e os resultados finais alcançados, enquanto nas seções seguintes são trazidos os objetivos do trabalho e um breve resumo da história da WEG.

1.1 WEG

A WEG é uma empresa brasileira fundada em 1961 pelo eletricista Werner Ricardo Voigt, pelo administrador Eggon João da Silva e pelo mecânico Geraldo Werninghaus, sendo inicialmente chamada de Eletromotores Jaraguá e posteriormente trocando de nome para refletir a inicial de seus fundadores. A empresa atua na fabricação de motores elétricos, geradores, transformadores e equipamentos de automação industrial. Desde sua fundação, a WEG tem se destacado como uma empresa inovadora e líder no mercado brasileiro e internacional.

Ao longo de sua história, a WEG expandiu sua presença global, possuindo fábricas em 14 países, filiais em 38 países e mais de 38 mil colaboradores. A empresa também ampliou sua gama de produtos e serviços, incluindo soluções de energia renovável e de eficiência energética. A WEG tem sido reconhecida por sua excelência em inovação e sustentabilidade, sendo premiada com o selo de Empresa Cidadã pela BMF Bovespa e fazendo parte do índice de Sustentabilidade Empresarial (ISE) da B3.

Atualmente, a WEG é uma das maiores empresas fabricantes de motores elétricos do mundo e uma das principais empresas de equipamentos de geração de energia no Brasil. A empresa tem uma forte presença em mercados globais, incluindo América Latina, Europa, Ásia e África. A WEG continua a investir em inovação e sustentabilidade, buscando expandir sua presença global e fornecer soluções cada vez mais avançadas para seus clientes.

A WEG Transmissão e Distribuição de Blumenau é uma divisão da empresa WEG focada em soluções para o setor de transmissão e distribuição de energia elétrica, as

principais sendo:

a) Transformadores: em todas as faixas de potência, desde transformadores de distribuição até transformadores de potência, transformadores de instrumentação e transformadores a seco.

b) Dispositivos de Proteção e Controle: A WEG TD oferece uma linha completa de dispositivos de proteção e controle, incluindo disjuntores, chaves seccionadoras, chaves de isolamento e dispositivos de medição. Esses dispositivos são projetados para garantir a segurança e a confiabilidade do sistema elétrico.

c) Sistemas de Automação: incluem softwares e equipamentos para automação de subestações, automação de linhas de transmissão e automação de distribuição.

d) Sistemas de Proteção e Controle: como sistemas de proteção de subestações, sistemas de proteção de linhas de transmissão e sistemas de proteção de distribuição.

1.2 OBJETIVO GERAL

Implementar uma solução composta por diversas automações capazes de aumentar a produtividade em projetos de comando e controle de transformadores, que utilize boas práticas de desenvolvimento e que seja escalável, permitindo expandir os casos de uso das automações existentes e desenvolver novas automações.

1.3 OBJETIVOS ESPECÍFICOS

a) Reduzir a necessidade de executar tarefas repetitivas durante o desenvolvimento de projetos de circuitos de comando e controle de transformadores;

b) Melhorar a padronização dos processos e do projeto final;

c) Aumentar a produtividade dos projetistas, permitindo que seu tempo seja utilizado principalmente em atividades que agreguem valor ao projeto entregue ao cliente.

d) Estudar e aplicar as metodologias de padrões de projeto em programação para o desenvolvimento dos sistemas propostos nesse trabalho.

2 REVISÃO

Os conceitos de metodologias ágeis foram bastante utilizados tanto para o planejamento quanto para o desenvolvimento do projeto, permitindo um melhor planejamento das atividades, escolha das tarefas mais relevantes e acompanhamento da evolução do projeto.

Com relação à parte prática do desenvolvimento, um dos pontos mais importantes, além do uso do paradigma de orientação a objetos, foi a utilização de padrões de projeto. Através de estratégias bem estabelecidas para a resolução de problemas comuns dentro do desenvolvimento de aplicações foi possível entregar um resultado final de maior qualidade tanto para os usuários quanto para os desenvolvedores.

Ambos os conceitos são discutidos em mais detalhes nas seções seguintes.

2.1 METODOLOGIA ÁGIL

A metodologia ágil se refere a um conjunto de práticas que visam aumentar a eficiência dos processos dentro de um time, proporcionando uma entrega de valor contínua, de qualidade e alinhada com as necessidades do cliente. A metodologia foi pensada por um conjunto de desenvolvedores de software durante a década de 90 em resposta aos processos vigentes na época, muito engessados e lentos para atender às constantes mudanças existentes dentro da área de desenvolvimento (MARTIN, 2002).

Alguns valores fundamentais foram levantados como sendo a base para a metodologia ágil:

- Software em funcionamento mais que documentação abrangente;
- Indivíduos e interação mais que processos e ferramentas;
- Colaboração com o cliente mais que negociação de contratos;
- Responder a mudanças mais que seguir um plano.

Percebe-se que a metodologia se preocupa muito mais com o produto final do que com algum planejamento definido ao início do projeto.

De fato, o conceito de metodologias ágeis vem ganhando bastante espaço dentro de qualquer área, porém, mais ainda em projetos que estão constantemente correndo contra o tempo, tanto pelas expectativas dos clientes quanto pelas próprias tecnologias, que avançam quase mais rápido do que é possível se adaptar. Inclusive, o termo ágil não diz respeito necessariamente a desenvolvimentos cada vez mais rápidos, mas sim à capacidade de mudar de direção rapidamente. Para que isso seja garantido, é essencial que haja uma comunicação constante entre a equipe de desenvolvimento e o cliente, possibilitando que cada nova implementação seja validada e que esteja o mais alinhada possível às suas necessidades.

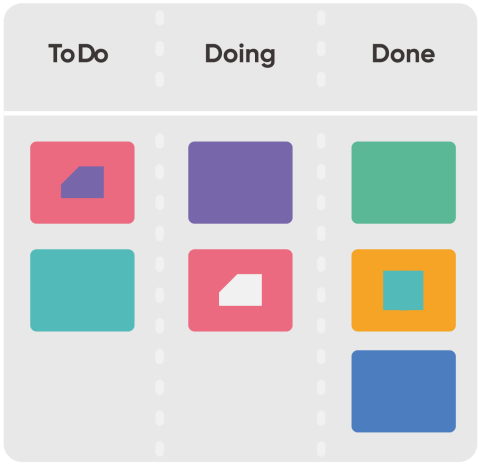
Outro ponto essencial da metodologia são as entregas constantes e incrementais, que também permitem a identificação de erros e sua correção o mais breve possível, garantindo que pequenos problemas durante o início do desenvolvimento sejam rapidamente tratados e evitando que se tornem grandes problemas em fases mais tardias de implementação.

Algumas das ferramentas utilizadas para a aplicação de metodologias ágeis são os métodos Kanban e Scrum, ambos melhor explicados na seção seguinte.

2.1.1 Kanban

O Kanban é um método que foi originalmente desenvolvido para o gerenciamento de tarefas em fábricas, mas que foi muito bem adaptado a projetos de desenvolvimento de sistemas computacionais. O método é baseado em três princípios, sendo eles: a visualização do fluxo de trabalho, limitação do trabalho em progresso e melhoria contínua. A visualização do fluxo de trabalho é feita através de um quadro onde são dispostas as tarefas planejadas, geralmente no formato de pequenos cartões com a descrição da tarefa, que são movidos entre os diferentes estágios de desenvolvimento. Um quadro Kanban bastante simples poderia ser composto, por exemplo, por três estágios: aguardando, em desenvolvimento e finalizado.

Figura 1 – Ilustração de um quadro Kanban.



Fonte: Nulab (2023)

A representação visual das tarefas e de seus status facilita muito o entendimento do andamento do projeto e do volume de trabalho já desenvolvido, em desenvolvimento e dos desenvolvimentos restantes.

A limitação do trabalho se dá pela definição de um limite de tarefas em desenvolvimento que precisam ser encerradas antes do time dar início a novas atividades. Essa

restrição garante que a equipe nunca estará sobrecarregada e que poderá focar no trabalho atual a ser desempenhado.

Já a melhoria contínua é alcançada com a definição de indicadores de desempenho e revisão constante da qualidade das entregas.

2.1.2 Scrum

Possuindo algumas similaridades com o Kanban, o método Scrum também permite a criação de cartões de tarefas que devem ser implementadas pelos desenvolvedores, porém, as tarefas são deixadas em "backlog", campo que armazena todas as tarefas levantadas ao início do planejamento e que visam abranger todo o produto.

As tarefas que devem ser desenvolvidas e que levam até o produto final são quebradas em grupos menores, visando entregas parciais, geralmente de funcionalidades que se agregam para formar o produto como um todo. Essas tarefas ficam agrupadas em "sprint backlogs", ou seja, o conjunto de tarefas de uma sprint.

As sprints representam os períodos de tempo disponíveis para a implementação do projeto pelos desenvolvedores, onde o objetivo é tentar entregar o máximo de valor para o cliente ao final da sprint, que geralmente dura de uma a quatro semanas. Nesse contexto, o máximo valor não se refere necessariamente a maior quantidade de funcionalidades entregues, mas à maior qualidade das entregas feitas. Durante o período de uma sprint, o time deve não só implementar as tarefas, mas buscar e discutir as melhores formas de alcançar os objetivos definidos.

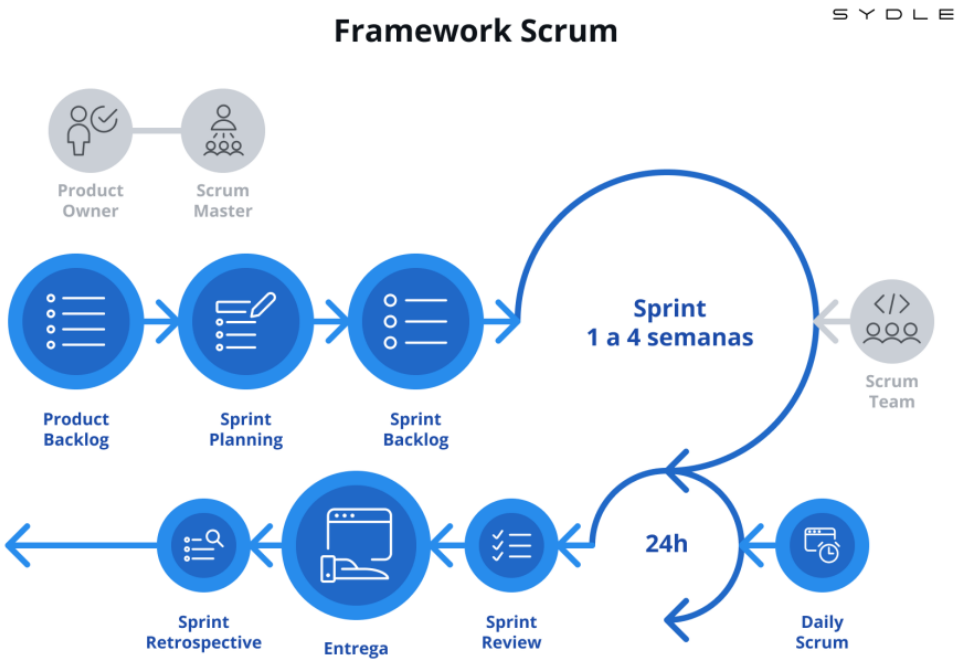
Semelhante à ferramenta Kanban, assim que as tarefas de uma sprint começam a ser desenvolvidas elas passam para o campo "em desenvolvimento", e em seguida para o campo "feito", permitindo também a inclusão de mais etapas dependendo das necessidades do projeto.

Outra vantagem da ferramenta é que as tarefas definidas durante o início do planejamento ou da própria sprint não são imutáveis, de forma que o time pode propor mudanças durante o desenvolvimento em caso de situação não previstas, como problemas durante o desenvolvimento, surgimento de novas ideias, ou qualquer outro motivo que o time identifique como uma oportunidade de agregar ainda mais valor ao produto.

O Scrum definine ainda outros conceitos, como os daily meetings (pequenas reuniões diárias para o alinhamento do andamento do projeto), sprint reviews (reuniões de feedback sobre a sprint recém encerrada), scrum master (colaborador responsável por garantir o seguimento da metodologia pelo time), product owner (responsável pelo produto final e que tem a função de auxiliar o time a se manter alinhado com as expectativas do cliente), entre outros (RODRIGUES, 2020).

A Figura 2 ilustra os personagens participantes e as etapas da metodologia Scrum.

Figura 2 – Ilustração das etapas da ferramenta Scrum.



Fonte: Sydle (2023)

2.2 PADRÕES DE PROJETO

Conforme a orientação a objetos se tornava o paradigma mais utilizado dentro da indústria para o desenvolvimento de aplicações, diversos problemas comuns entre diferentes projetos foram sendo notados. Não apenas isso, mas as soluções mais adequadas para esses problemas também pareciam seguir uma lógica bastante parecida.

Notando essas similaridades e se baseando no conceito de padrões de projetos para arquitetura escrito por Christopher Alexander (ALEXANDER *et al.*, 1977), onde são discutidos temas como a altura ideal para janelas, quantidade de andares em um edifício, tamanho das áreas verdes presentes em uma área residencial, entre outros, os quatro autores Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm escreveram, em 1994, o livro "Design Patterns: Elements of Reusable Object-Oriented Software", ou "Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos" (GAMMA *et al.*, 2009). No livro são discutidos 23 padrões de projeto que visam resolver de maneira sistemática diversos problemas encontrados frequentemente em projetos de desenvolvimentos que utilizam o paradigma de orientação a objetos.

No livro, em geral um padrão possui quatro elementos: seu nome, o problema do qual ele trata, a solução e seus prós e contras. As soluções não descrevem uma implementação

concreta que deve ser aplicada, mas através do contexto do problema apresenta uma solução abstrata, um modelo de elementos, relacionamentos e responsabilidades que se utilizados em conjunto funcionam para resolver o problema em questão.

Por serem soluções já verificadas e aprovadas, a utilização de padrões de projeto também permite um foco maior nas especificidades da aplicação sendo desenvolvida, aumentando a eficiência do projeto. Geralmente os padrões já levam em conta quesitos como segurança, confiabilidade e escalabilidade, garantindo um resultado final de maior qualidade, e inclusive facilitam a comunicação entre os membros do projeto, já que cada padrão possui um nome e uma lógica bem definida.

Apesar de todas as vantagens, é importante atentar-se ao uso excessivo de padrões de projeto, o que por vezes pode vir a aumentar a complexidade da aplicação sem necessidade.

Os padrões são divididos em três tipos: criacionais, estruturais e comportamentais, discutidos, respectivamente, nas seções 2.2.1, 2.2.2 e 2.2.3.

2.2.1 Padrões criacionais

Os padrões de projeto criacionais oferecem soluções para problemas comuns relacionados à criação de objetos de forma flexível, escalável e desacoplada das regras de negócio da aplicação, ajudando a manter o código mais limpo e organizado.

Esses padrões podem ainda ser divididos em três categorias: fábrica, construtor e protótipo. Os padrões de fábrica realizam a instanciação de objetos de acordo com as demandas, semelhante ao processo de uma fábrica, sem que saiba exatamente os detalhes da criação desses objetos.

O padrão construtor permite instanciar objetos de maneira incremental, lembrando uma linha de montagem, mas onde as etapas de construção são independentes e podem complementar umas às outras

Por fim, o padrão protótipo se baseia em algum outro objeto preexistente que serve de base para a instância de um novo objeto, assim como um protótipo permite a construção de elementos semelhantes a ele.

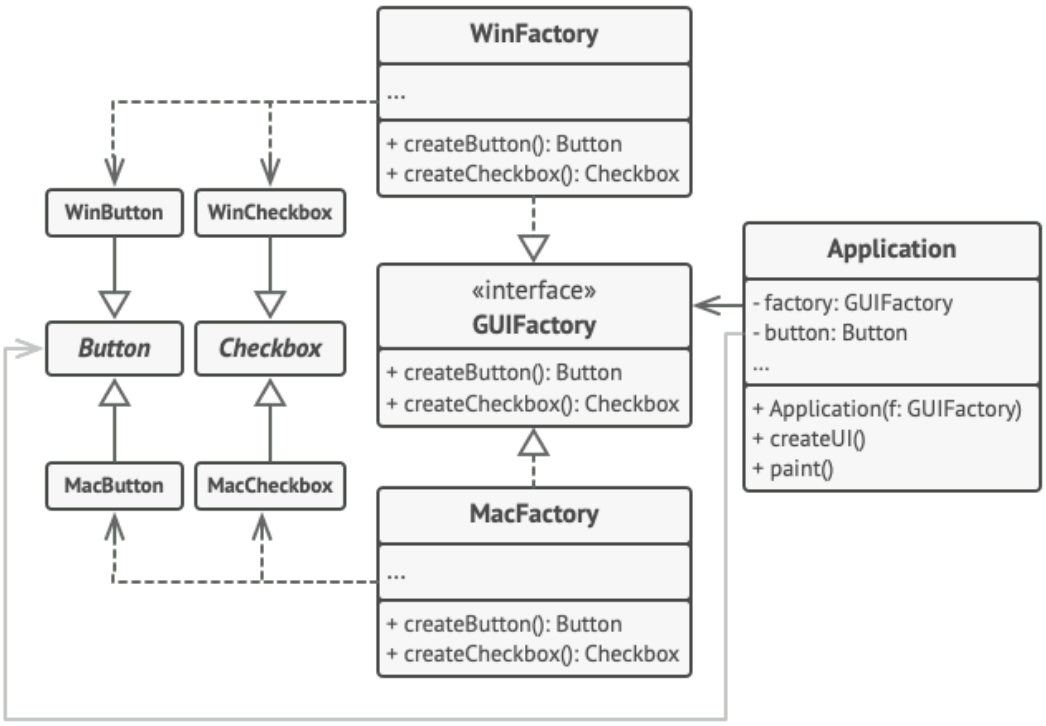
Os padrões criacionais são:

1. Abstract Factory

   O padrão Abstract Factory permite a criação de objetos relacionados sem a necessidade de especificar suas classes concretas. É uma solução indicada quando se deseja criar objetos que possuem características específicas apesar de pertencerem a uma mesma categoria. Um exemplo é a criação de elementos de interface gráfica, como botões e caixas de texto. Para sistemas Windows e Mac, os elementos possuem especificidades que não são importantes para a classe cliente. Dessa maneira, a interface Abstract Factory pode declarar os métodos "CriaBotao" e "CriaCaixaDeTexto", que são implementados nas classes concre-

tas WindowsFactory e MacFactory, ou seja, classes responsáveis por instanciar os elementos gráficos específicos para cada sistema operacional. Ainda, os elementos “Botao” e “CaixaDeTexto” devem implementar os atributos e métodos declarados nas interfaces que especificam as características desses elementos, independente do sistema para qual serão criados. A Figura 3 demonstra a estrutura das interfaces e classes criadas para o padrão em questão.

Figura 3 – Ilustração do padrão abstract factory.



Fonte: Guru (2023a)

Dessa maneira, o objeto retornado é do tipo da interface, de forma que a classe cliente não conhece os detalhes da implementação do objeto e só faz uso dos atributos e métodos declarados na interface “Botao” e “CaixaDeTexto”.

2. Builder

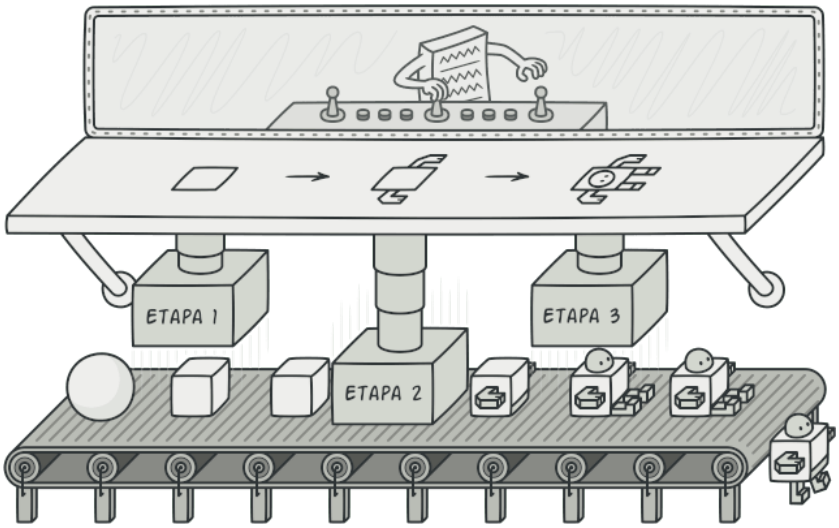
   Para objetos muito complexos, a sua criação pode levar a construtores com muitos parâmetros, de forma que a maioria desses parâmetros nem sequer é utilizada, resultando em um código bastante poluído.

   O padrão Builder permite a instanciação de objetos complexos passo a passo, utilizando uma classe construtora diferente da classe do produto que será construído.

Pode-se utilizar como exemplo a criação de um objeto carro que possui diversos opcionais, como ar-condicionado, direção elétrica, sensor de estacionamento, controle de estabilidade, etc. Por mais que se quisesse construir um carro sem nenhum opcional, ainda assim seria necessário passar por parâmetro para o construtor da classe carro cada um dos seus itens opcionais, mesmo que fosse para indicar a ausência desse item.

O padrão Builder transforma a agregação desses adicionais em métodos de uma classe construtora, permitindo que sejam invocados apenas os métodos que adicionam ao carro os itens desejados.

Figura 4 – Representação do padrão Builder.

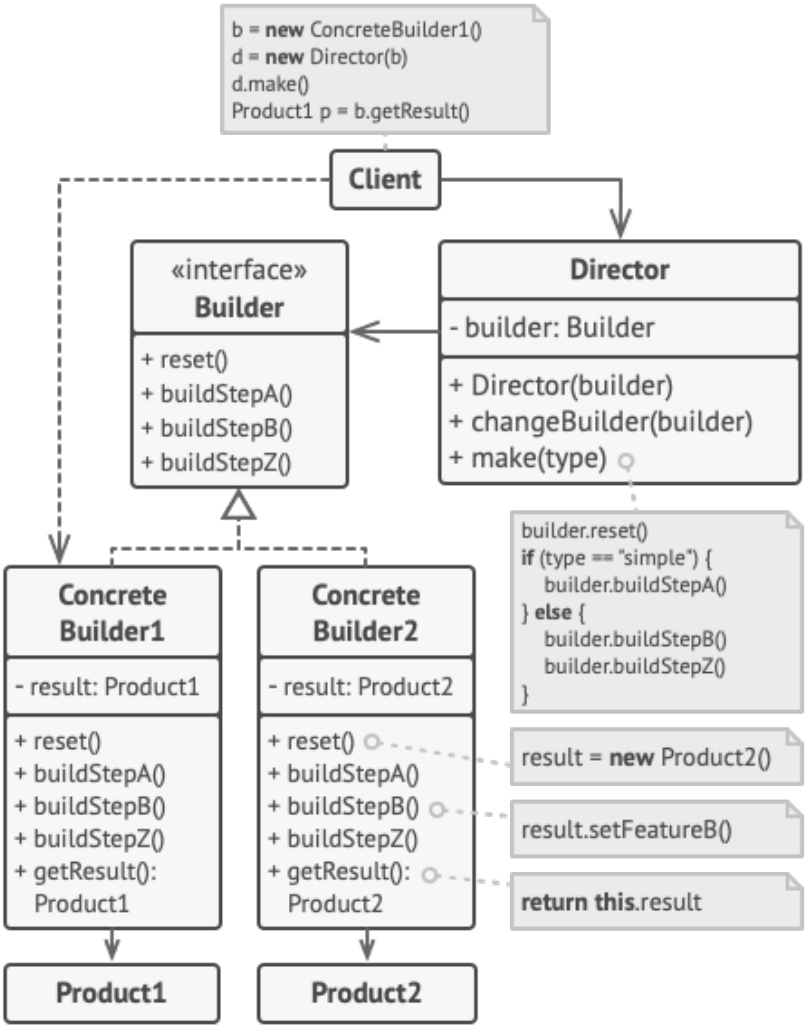


Fonte: Guru (2023d)

A Figura 4 ilustra o funcionamento do padrão Builder, onde cada etapa pode ser executada de maneira sequencial, mas ainda permitindo que apenas as etapas desejadas sejam aplicadas.

Adicionalmente, pode-se ainda utilizar uma classe diretora, que instancia um objeto construtor e possui certas etapas predefinidas. Na classe diretora seria possível, por exemplo, implementar os métodos “ConstruirCarroSimples” e “ConstruirCarroCompleto”, em que o primeiro retornaria um objeto carro sem nenhum opcional, enquanto o segundo retornaria um objeto carro com todos os opcionais disponíveis.

Figura 5 – Ilustração do padrão Builder.

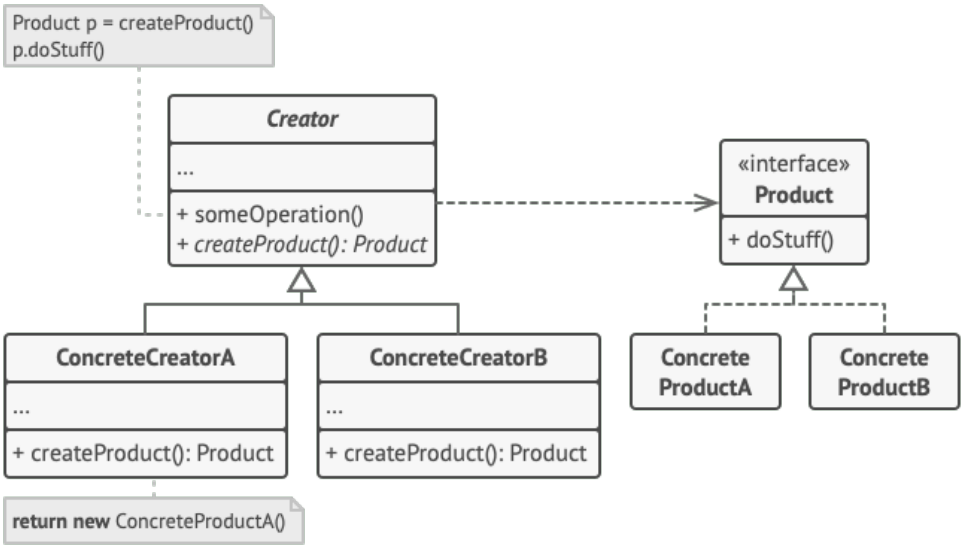


Fonte: Guru (2023d)

3. Factory Method

   O padrão Factory Method permite a instanciação de objetos em uma superclasse, mas delega para as subclasses a responsabilidade por escolher qual tipo de objeto será criado. Para a implementação desse padrão, são criadas classes concretas que implementam o método de criação de objetos definido na interface Creator, que retorna um objeto do tipo Product, sendo essa a interface implementada pelos produtos concretos.

Figura 6 – Ilustração do padrão Factory Method.



Fonte: Guru (2023j)

A Figura 6 apresenta as interfaces Creator e Product, que são implementadas pelas classes concretas de criação de objetos e dos objetos em si, respectivamente. Esse padrão é indicado quando não se conhece de antemão os tipos de objetos que podem vir a ser utilizados no código, mantendo uma solução com capacidade para a expansão dos tipos de produtos utilizados.

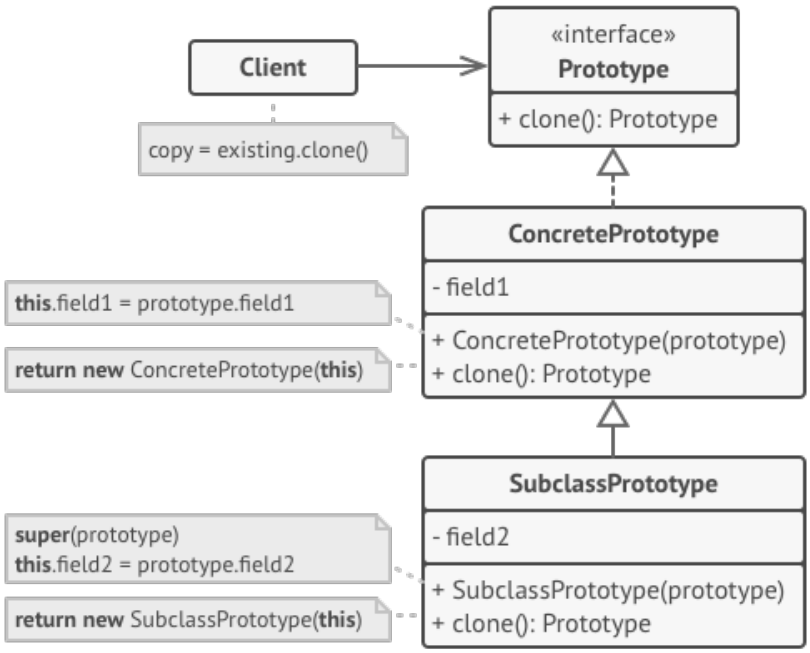
4. Prototype

   O padrão Prototype é utilizado quando se deseja criar cópias de objetos já existentes. Geralmente uma situação como essa é necessária quando o processo de criação do objeto é muito custoso, dependendo de acessos a recursos externos, por exemplo.

   Se no código da aplicação fosse criada uma nova instância de um objeto e todos os atributos do objeto original fossem copiados, o objeto cópia ainda poderia não ser fiel ao objeto original, já que é comum a existência de atributos privados que não podem ser acessados externamente à sua classe.

   O padrão Prototype permite resolver esse problema passando a responsabilidade pela cópia dos objetos para a própria classe do objeto. Geralmente classes do tipo protótipo implementam uma interface Prototype que possui apenas o método de clonagem. Na classe, a implementação do método é feita construindo um novo objeto e alimentando todos os seus atributos com os valores do objeto original, permitindo inclusive a cópia ou exclusão recursiva de atributos que são instâncias de outras classes.

Figura 7 – Ilustração do padrão Prototype.



Fonte: Guru (2023p)

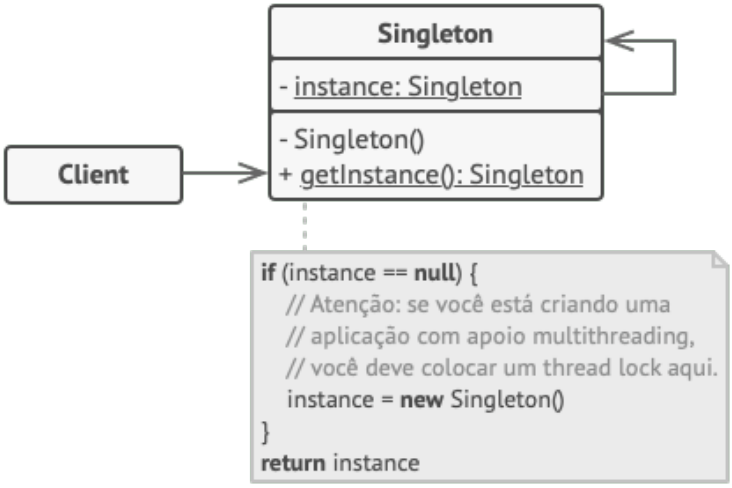
5. Singleton

   O padrão Singleton garante que uma classe possua apenas uma instância, provendo um ponto de acesso global a essa instância e a protegendo de modificações por outras classes.

   Esse tipo de situação é importante quando a aplicação faz uso de recursos compartilhados, como um banco de dados ou um arquivo. O objeto da classe que implementa o padrão Singleton pode, por exemplo, abrigar a conexão com o banco de dados, evitando que a cada novo acesso ao banco uma nova conexão ao banco seja criada.

   Isso é possível tornando o construtor da classe um método privado e a instância um atributo privado da própria classe, e deixando disponível um método público para obter o objeto. O método fica responsável por verificar se o objeto da classe já foi instanciado, ou instanciá-lo caso contrário, e retorná-lo ao contexto desejado.

Figura 8 – Ilustração do padrão Singleton.



Fonte: Guru (2023r)

2.2.2 Padrões estruturais

Os padrões de projeto estruturais fornecem soluções para problemas comuns relacionados à estruturação de objetos, como a criação de estruturas hierárquicas, adaptação de objetos para que sejam compatíveis em contextos diferentes e alteração das responsabilidades de um objeto de maneira dinâmica, ainda mantendo a organização do código.

Os padrões estruturais são:

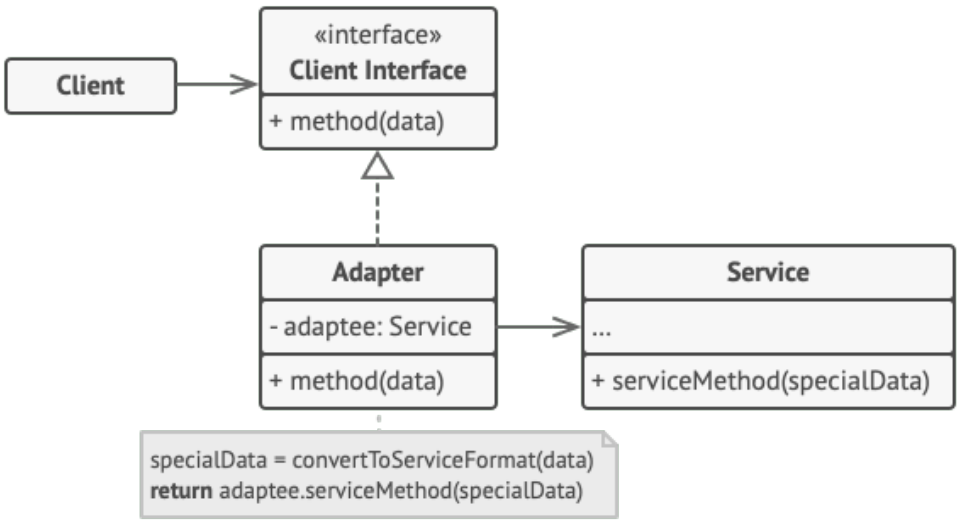
1. Adapter

   O padrão Adapter permite que objetos ou classes originalmente incompatíveis com a interface implementada por um serviço possam trabalhar de maneira conjunta.

   Isso é feito criando uma classe que possui métodos capazes de transformar as estruturas de dados originais no formato exigido pela interface do serviço.

   Pode-se considerar o exemplo de um serviço de meteorologia que faz a previsão do tempo com base em dados nas unidades do sistema imperial, enquanto os sensores fornecem os dados no sistema métrico. Nesse caso é possível criar uma classe que recebe os dados nas unidades métricas, faz a conversão para as unidades imperiais e fornece os dados transformados para o serviço de previsão do tempo.

Figura 9 – Ilustração do padrão Adapter.



Fonte: Guru (2023b)

2. Bridge

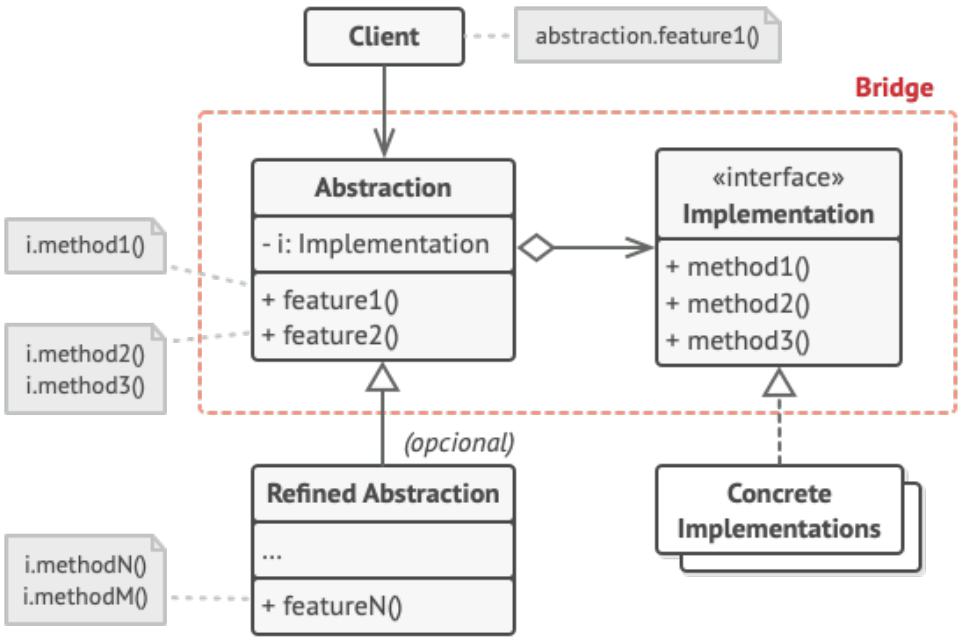
   O padrão Bridge utiliza o conceito de composição em vez de herança de classes, permitindo que tanto a classe de abstração quanto a de implementação possam ser alteradas individualmente, de maneira que o número de classes criadas para representar todas as possíveis combinações de objetos não cresça exponencialmente.

   O padrão Bridge possui dois componentes principais: a classe de abstração e a classe de implementação. Ambas são classes abstratas que funcionam como interfaces para as classes de abstração e as classes de implementação concretas.

   Pode-se utilizar como exemplo as classes de abstração Carro e Moto, que implementam a classe abstrata Veículo, e as classes de implementação Produção e Montagem, que implementam a classe Oficina. Ao adicionar um atributo do tipo Oficina aos objetos que implementam Veículo, torna-se possível alterar as características dos veículos sem que os métodos da Oficina sejam alterados e vice-versa (link: https://www.geeksforgeeks.org/bridge-design-pattern/).

   Se não fosse pela flexibilidade fornecida pelo padrão Bridge, caso alguma característica da classe Carro fosse atualizada, os seus métodos de fabricação e montagem também precisariam ser alterados, situação que se torna insustentável conforme o número de classes de abstração e implementação começa a crescer ou conforme sua complexidade aumenta.

Figura 10 – Ilustração do padrão Bridge.



Fonte: Guru (2023c)

3. Composite

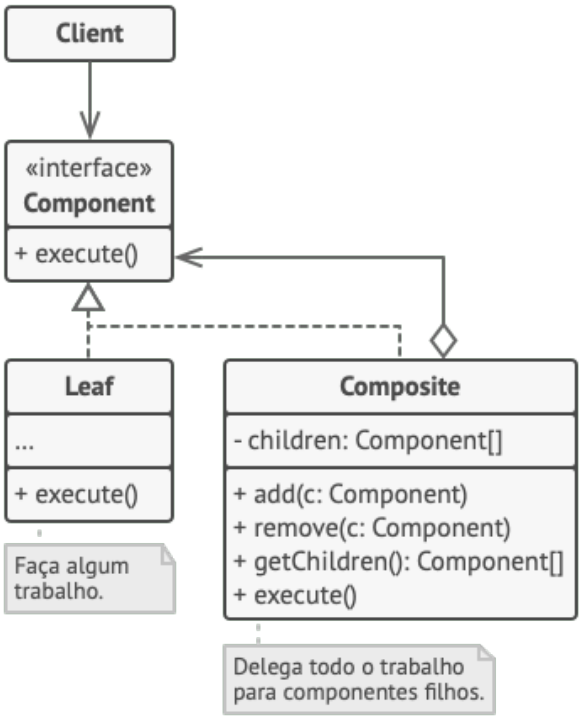
   O padrão Composite é bastante útil quando se deseja trabalhar com objetos em estruturas hierárquicas, como dados em árvore, grafos, estruturas de arquivos, entre outros, onde coleções de objetos possuem comportamentos semelhantes aos seus objetos individuais, permitindo que os objetos de superclasses deleguem suas funções para os elementos das subclasses até que se chegue nas folhas da estrutura, ou seja, nos últimos nós da estrutura que não possuem elementos filhos.

   Situações onde essa lógica se aplica podem ser encontrados nos diferentes níveis de estruturas políticas, como o poder executivo a nível federal, estadual e municipal, nas organizações militares, e até em estruturas de arquivos, onde pastas podem conter outras pastas e arquivos, que por outro lado podem possuir ainda mais pastas e arquivos e assim por diante.

   O padrão Composite poderia ser usado, por exemplo, para contar o número total de elementos dentro de uma determinada pasta. Para isso, todos os elementos da estrutura deveriam implementar uma interface com um método para percorrer todos os seus elementos filho e executar novamente o método da interface. Esse comportamento seria executado em cascata, do nível mais alto até o nível mais baixo, quando já não houvessem mais elementos filho dentro do nó sendo analisado. Nesse momento o número de elementos dentro dessa pasta

seria contabilizado e começaria a subir na cascata, sendo somado ao número de elementos de todas as outras pastas até alcançar novamente a pasta mais superior, trazendo consigo o número total de elementos.

Figura 11 – Ilustração do padrão Composite.



Fonte: Guru (2023g)

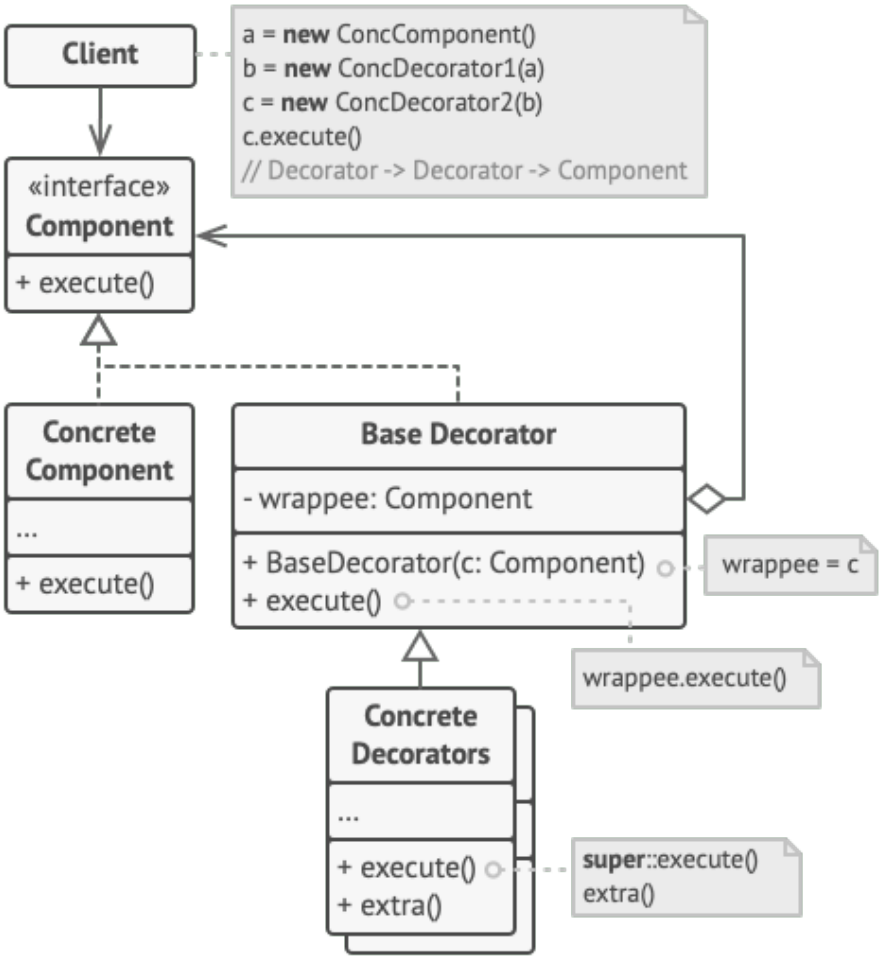
4. Decorator

   O padrão Decorator permite que os comportamentos de objetos sejam alterados em tempo de execução através do uso de decoradores, agregando comportamentos ao objeto conforme a necessidade. Ao envolver um objeto com decoradores que implementam os métodos de uma interface Decorator, o objeto agrega todas as diferentes implementações, conforme os decoradores com os quais foi envolvido.

   Esse padrão de projeto é caracterizado ao definir uma interface Componente que é implementada tanto pelos envoltórios quanto pelos objetos envolvidos. Classes concretas de componentes podem ser definidas para implementar os comportamentos mais básicos do objeto. Também é definida uma interface Decorator, que ao mesmo tempo implementa a interface e possui uma referência privada para um objeto Componente. A referência privada guarda o componente passado por parâmetro durante a inicialização das classes concretas que implementam Decorator e que sobrescrevem seus métodos. Dessa maneira, os

objetos que implementam Decorator podem tanto incrementar comportamentos ao seu objeto quanto alterar seu comportamento original, resultando no comportamento dinâmico desejado.

Figura 12 – Ilustração do padrão Decorator.



Fonte: Guru (2023h)

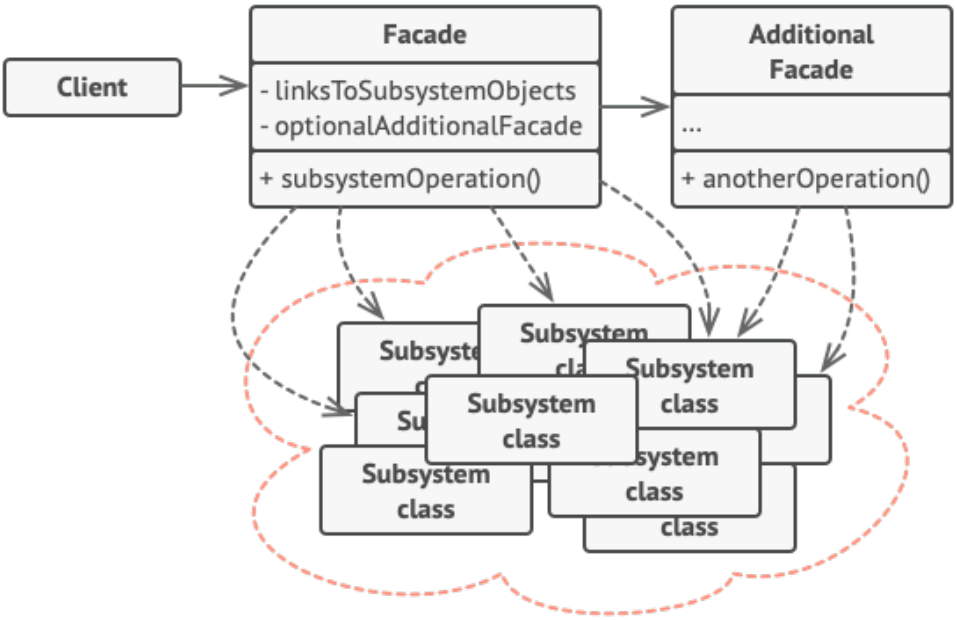
5. Façade

   O padrão Façade é um padrão estrutural que permite esconder toda a complexidade por trás de um conjunto de interfaces fornecendo uma interface simplificada para a utilização pelo usuário.

   Pode-se considerar o exemplo de uma loja online, que possui uma estrutura gigantesca para manter o negócio funcionando, desde a parte técnica, como o aplicativo da loja, a infraestrutura para manter o aplicativo disponível, estoque, banco de dados de clientes, processamento dos pedidos, passando pela cadeia de produção, com a compra de matérias-primas, processamento dos materiais, montagem ou produção do produto final, até os setores de marketing, gerência,

diretoria, enfim, uma cadeia produtiva tão complexa, mas que tem toda essa complexidade escondida do cliente, que está preocupado e vê apenas o pedido sendo confirmado na tela do aplicativo e a compra sendo entregue dentro de alguns dias.

Figura 13 – Ilustração do padrão Façade.



Fonte: Guru (2023i)

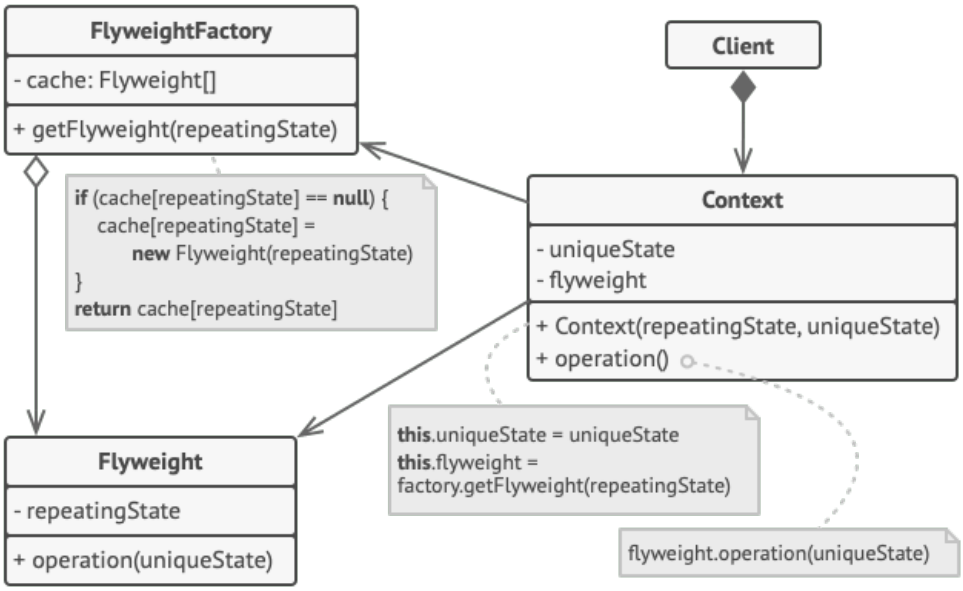
6. Flyweight

   O padrão Flyweight é um padrão de projeto que visa a economia de recursos permitindo o compartilhamento de objetos que possuem estados constantes ao longo de diversos elementos que possuem esses estados em comum.

   Geralmente utilizado em aplicações onde a economia de memória é importante, como sistemas de larga escala ou jogos.

   Utilizando os jogos como exemplo, é comum a existência de uma quantidade muito grande de objetos gráficos para compor uma cena, mas onde muitos deles possuem características bastante similares, como é o caso da vegetação do cenário. Todas as características em comum entre os elementos de vegetação são mantidos em um objeto compartilhado entre todas as instâncias da vegetação, mantendo nos objetos individuais apenas as características únicas entre si. Essa estratégia ajuda tanto a economizar recursos quanto melhorar a performance da aplicação.

Figura 14 – Ilustração do padrão Flyweight.



Fonte: Guru (2023k)

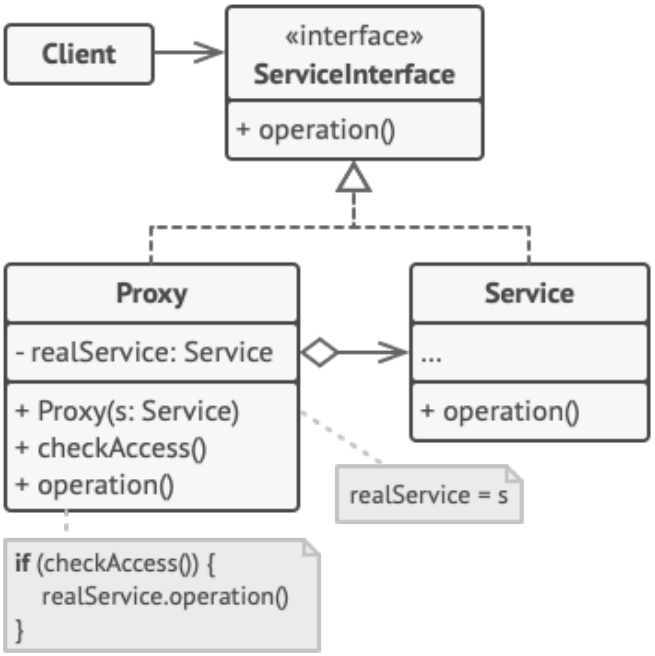
7. Proxy

   O padrão Proxy serve de interface entre o cliente e uma classe que presta algum tipo de serviço, fazendo o tratamento das requisições antes que elas sejam de fato processadas pelo serviço real. Através desse padrão é possível implementar controle de acesso ao serviço, tratamento dos dados da requisição, gerenciamento da sequência de acesso, entre outras funcionalidades necessárias.

   Para isso a classe Proxy implementa a mesma interface do serviço real, além de implementar funcionalidades auxiliares para garantir que todas as especificações exigidas e que a segurança do serviço real sejam atingidas.

   Pode-se utilizar como exemplo um serviço de streaming de filmes e séries que permite que todo o seu catálogo seja visualizado abertamente, mas que exige que seja feito o login para poder reproduzir qualquer mídia. Nesse caso pode ser implementada uma classe Proxy que faz a interface entre o catálogo aberto e a tela de reprodução, exigindo o login e verificando a conta do usuário antes de invocar o serviço real de reprodução.

Figura 15 – Ilustração do padrão Proxy.



Fonte: Guru (2023q)

2.2.3 Padrões comportamentais

Os padrões de projeto comportamentais apresentam soluções para problemas comuns relacionados à comunicação entre objetos e definição de suas responsabilidades, visando ainda manter a organização, flexibilidade e escalabilidade do código.

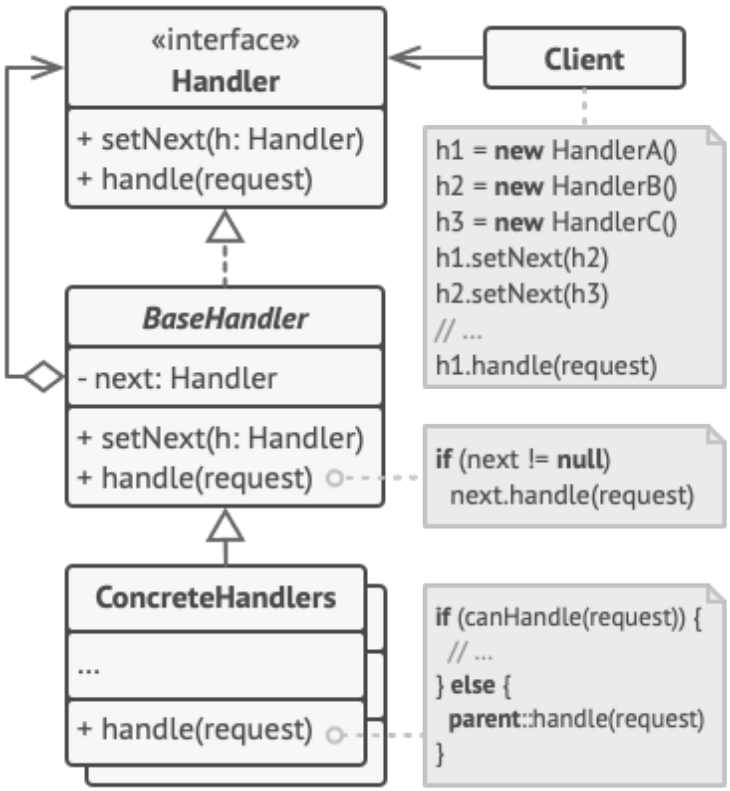
Alguns padrões comportamentais são:

1. Chain of responsibility

   O padrão Chain of Responsibility permite que diversos tipos de requisições sejam processados de diferentes maneiras por uma série de handlers encadeados, ou seja, objetos responsáveis por processar requisições, mas que podem analisá-las e decidir se processam os pedidos ou se os passam para o próximo objeto da lista. As requisições são propagadas através da cadeia de handlers até que um deles verifique que é o responsável por tratar aquele pedido, ou até que a requisição alcance o último objeto da lista sem ser processado.

   Esse padrão é útil quando não se sabe de antemão quais são exatamente os pedidos que deverão ser processados e também quando se deseja delegar o tratamento das requisições de forma dinâmica.

Figura 16 – Ilustração do padrão Chain of Responsibility.



Fonte: Guru (2023e)

2. Command

   O padrão Command permite transformar a invocação de métodos em objetos que contém toda a informação sobre a requisição.

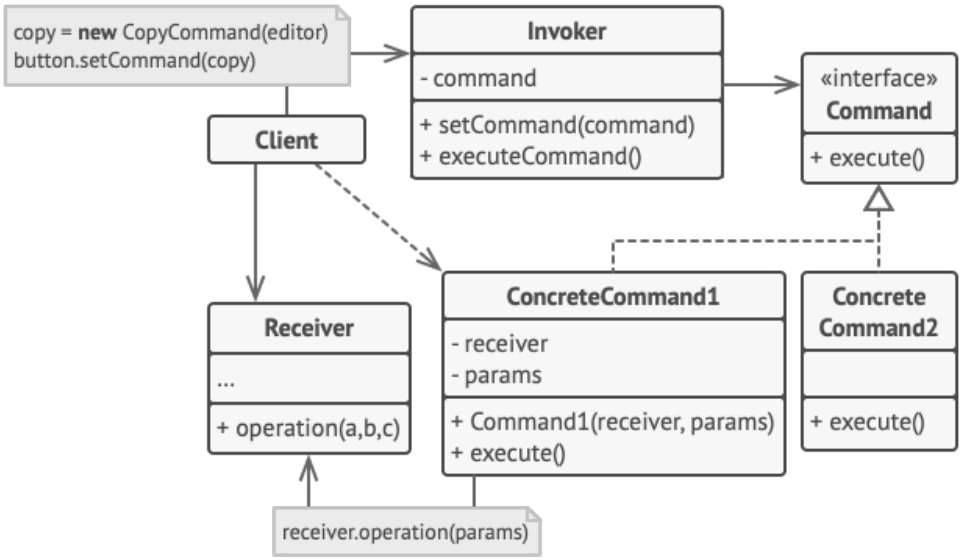
   Os principais elementos desse padrão são os objetos remetentes, que possuem uma referência para um comando, e os objetos Command, que executam a ação.

   Para utilizar esse padrão, primeiro deve ser definida uma interface Command que declara um método de execução. Depois, todas as classes Command concretas devem implementar essa interface adicionando o seu comportamento ao método de execução.

   Por possuir uma referência para um objeto do tipo Command, os objetos invocadores podem solicitar a execução da ação associada ao seu comando em vez de chamar o método diretamente.

   As principais vantagens desse padrão são a capacidade de definir dinamicamente os comandos associados aos invocadores e permitir desfazer operações, já que é possível salvar em uma pilha cada um dos comandos executados.

Figura 17 – Ilustração do padrão Command.



Fonte: Guru (2023f)

3. Iterator

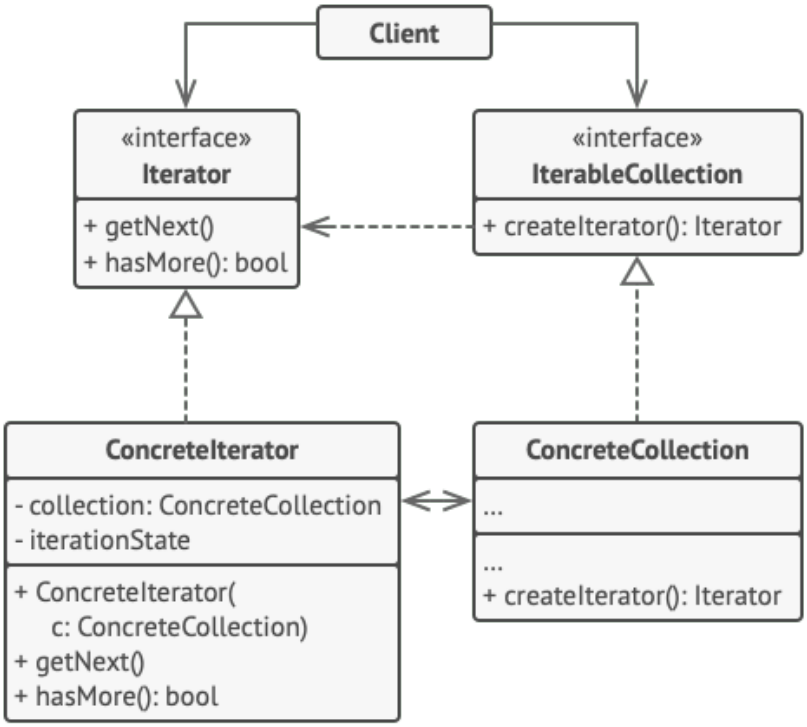
   O padrão Iterator é um padrão comportamental que permite extrair a lógica de acesso aos elementos de uma coleção sem que o cliente precise conhecer a estrutura da coleção ou lidar com a sua complexidade. Isso pode ser desejado tanto para simplificar o processo de iteração quanto por motivos de segurança.

   Os principais componentes do padrão Iterator são: a interface do iterador e suas classes concretas, e a interface das coleções e suas classes concretas.

   A interface do iterador precisa declarar no mínimo um método para obter o próximo elemento da coleção, podendo também declarar outros métodos para recuperar o elemento anterior, calcular quantos elementos faltam para alcançar o fim da coleção, entre outros, enquanto a interface da coleção deve declarar um método para obter uma instância de um iterador compatível com a sua estrutura.

   São nas classes iteradoras concretas que os algoritmos para percorrer as coleções são de fato implementados, classes essas que devem guardar uma referência para a coleção que irão percorrer. Essa associação é realizada quando a coleção passa uma referência de si própria para o construtor do iterador.

Figura 18 – Ilustração do padrão Iterator.



Fonte: Guru (2023l)

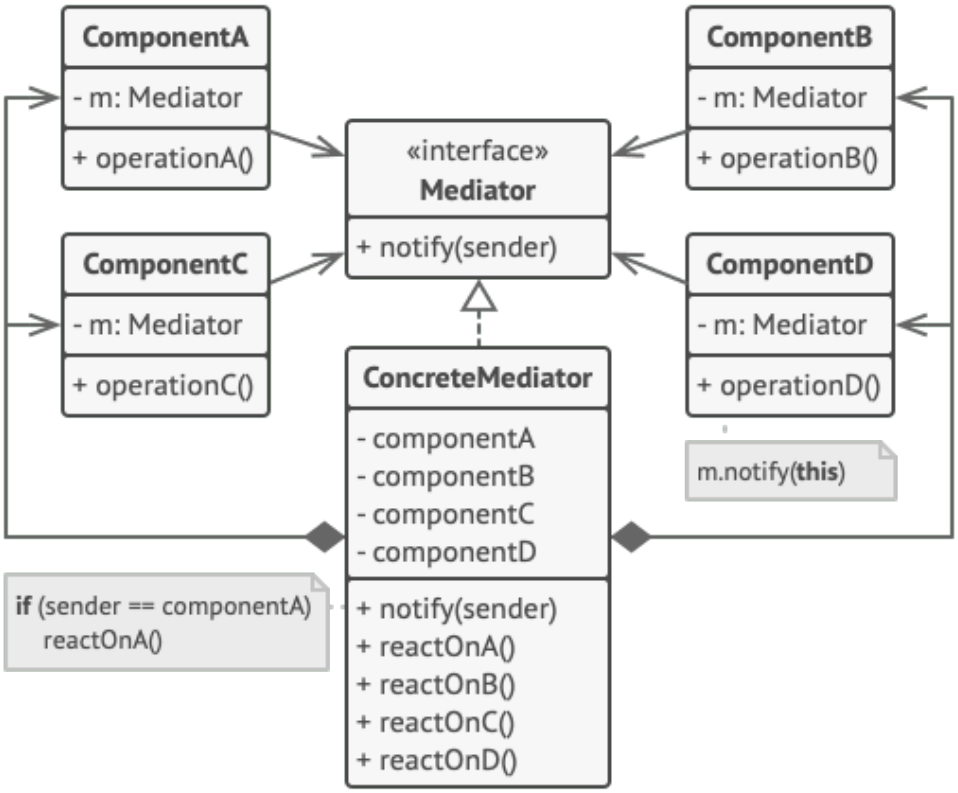
4. Mediator

   O padrão de projeto Mediator tem como objetivo definir um objeto intermediário que controla a comunicação entre objetos relacionados. Em vez dos objetos se comunicarem diretamente, o que implicaria que todos os objetos mantivessem referências para os seus objetos relacionados e conhecessem as particularidades de suas implementações, eles se comunicam através do mediador, que tem a responsabilidade de gerenciar todas as interações entre eles, ajudando a manter uma estrutura mais clara e de baixo acoplamento.

   Para implementar o padrão Mediator deve-se declarar uma interface Mediator que declara um método de comunicação com os objetos, geralmente apenas um método para notificá-los sobre as atualizações relevantes.

   As classes Mediator concretas implementam essa interface, além de guardar uma referência para cada um dos objetos relacionados e implementar métodos específicos para lidar com os diferentes objetos sob sua responsabilidade.

Figura 19 – Ilustração do padrão Mediator.



Fonte: Guru (2023m)

5. Memento

   O padrão Memento é um padrão de projeto que permite salvar os estados de um objeto e restaurá-los quando necessário.

   Esse padrão é comumente encontrado em aplicações que permitem desfazer ações, como softwares de desenho, por exemplo, e em aplicações que podem ser prejudicadas caso a manipulação de seus estados sofra algum problema, permitindo restaurar o estado dos dados antes que qualquer manipulação fosse executada.

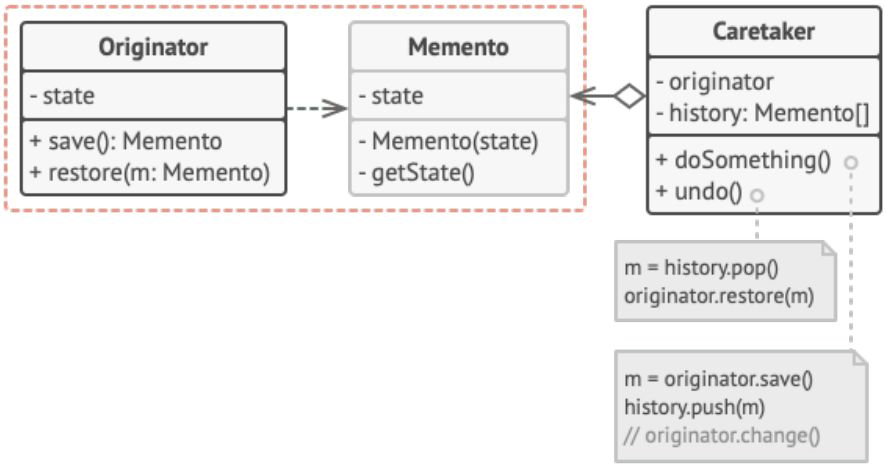
   O padrão memento possui três participantes principais: a classe Originadora, a classe Memento e a classe Cuidadora. A classe Originadora possui os estados que se deseja salvar, a classe memento é responsável por guardar os estados da originadora, e a classe Cuidadora gerencia o salvamento e restauração dos estados.

   É possível implementar o padrão Memento de diferentes maneiras:

   a) Classes aninhadas:

A classe Memento é declarada dentro da classe Originadora, de forma que possui acesso a todos os seus campos, mesmo os que forem declarados privados. Por outro lado, a classe Cuidadora nesse caso não possui acesso restrito aos atributos da classe Memento, tornando inviável a manipulação de seus campos.

Figura 20 – Ilustração do padrão Memento com classes aninhadas.



Fonte: Guru (2023n)

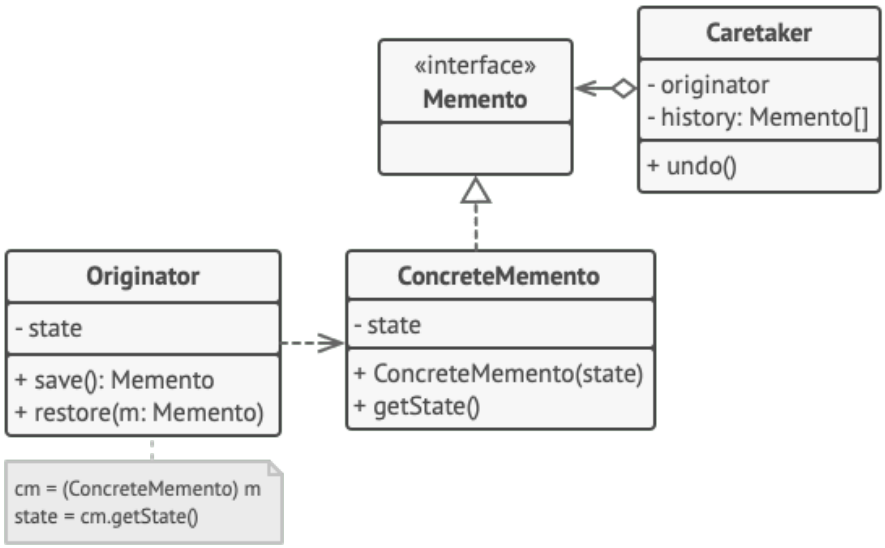
b) Interface intermediária

É implementada uma interface Memento que declara apenas os métodos relacionados à persistência dos estados da classe Originadora, de forma que todos os seus atributos continuam inacessíveis à classe Cuidadora, enquanto a classe Originadora se comunica diretamente com objetos de uma classe Memento concreta, que implementa a interface Memento.

c) Encapsulamento restrito

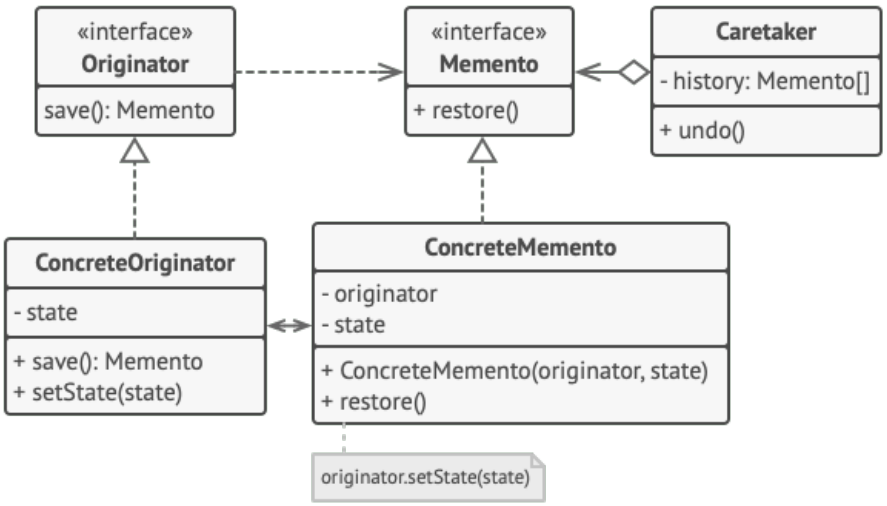
São criadas interfaces tanto para as classes Originadoras quanto para as classes Mementos, de forma que cada classe Originadora tenha sua correspondente classe Memento. A classe cuidadora agora se torna independente da classe Originadora, pois a responsabilidade de restaurar seus estados é transferida para a classe Memento.

Figura 21 – Ilustração do padrão Memento com interface intermediária.



Fonte: Guru (2023n)

Figura 22 – Ilustração do padrão Memento com encapsulamento restrito.



Fonte: Guru (2023n)

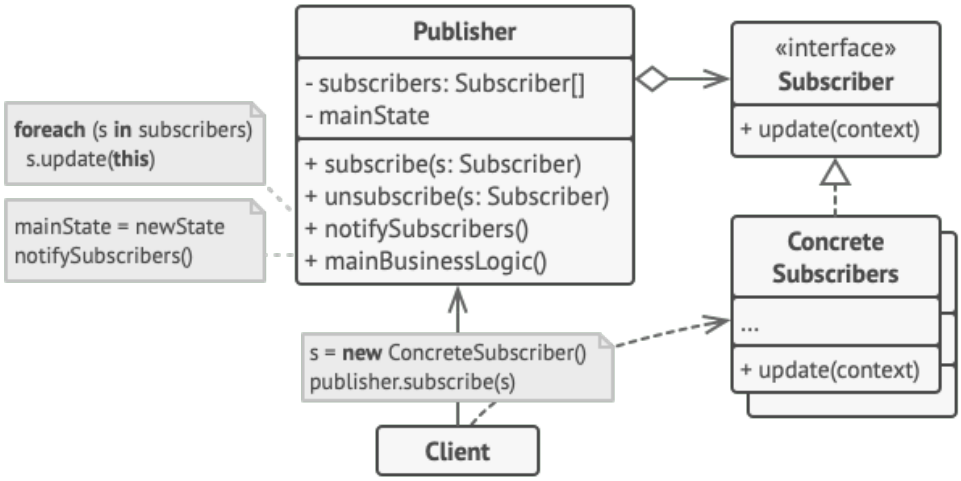
6. Observer

   O padrão Observer permite que um objeto, geralmente chamado de sujeito ou publicador, informe sobre mudanças de seus estados internos para uma coleção de objetos interessados nessas mudanças. Os objetos notificados são comumente chamados assinantes. Através da implementação desse padrão, os objetos assinantes não precisam ficar constantemente acessando o publicador para verificar se houve alguma mudança em seu estado, e se evita que o objeto

publicador notifique também objetos que não estão interessados nos seus valores internos.

Para implementar esse padrão o publicador deve possuir uma coleção de objetos assinantes, métodos para adicionar e excluir assinantes, e um método para percorrer a coleção e notificar cada um dos assinantes. Já os assinantes devem implementar um método de atualização que implementa a lógica esperada sempre que forem notificados de uma mudança no estado do objeto publicador.

Figura 23 – Ilustração do padrão Observer.



Fonte: Guru (2023o)

7. State

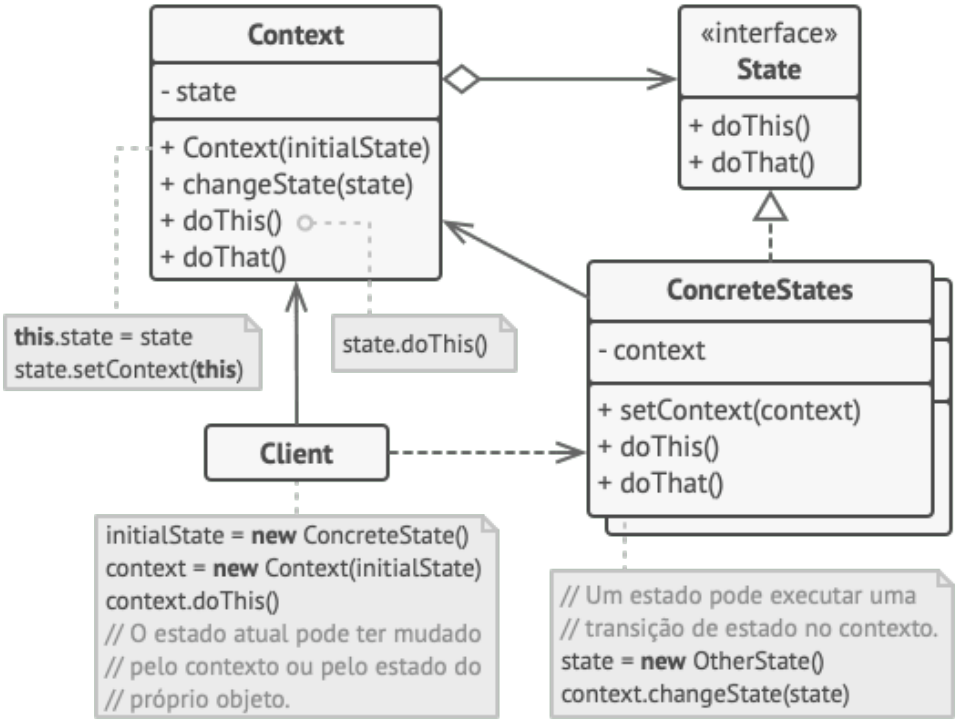
   O padrão State permite alterar o comportamento de um objeto de acordo com o seu estado interno, quase como se o objeto mudasse de classe.

   Para implementar o padrão State devem ser criadas classes para representar todos os estados possíveis do objeto, onde cada classe é responsável por implementar os comportamentos específicos daquele estado. O objeto original, por sua vez, armazena uma referência para o seu estado atual, o que altera a classe para qual o estado aponta e, por consequência, o seu comportamento.

   Esse padrão é bastante comum em aplicações que possuem estados bem definidos que representam uma lógica de negócio, como em aplicativos de entrega de comida, onde o pedido pode estar no carrinho, aguardando pagamento, em produção, em entrega e entregue.

   O padrão State proporciona as vantagens de tornar o código mais legível, já que cada comportamento está separado em classes que representam os diferentes estados da aplicação, além de evitar cadeias de ifs ou switches muito extensas.

Figura 24 – Ilustração do padrão State.



Fonte: Guru (2023s)

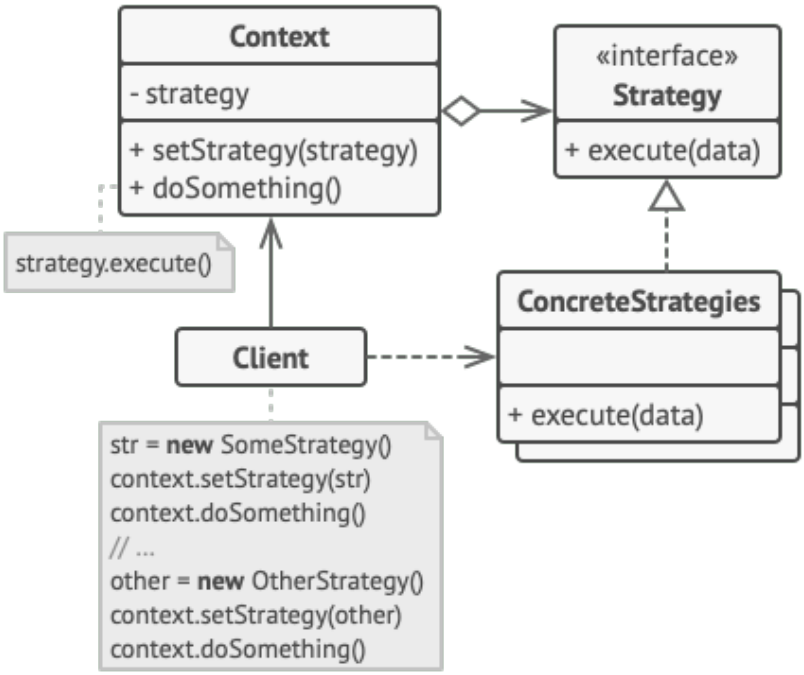
8. Strategy

   O padrão Strategy permite o desenvolvimento de diferentes algoritmos para um mesmo fim de maneira que alterações em uma das estratégias não afetem as outras.

   A implementação desse padrão se dá através da criação de diferentes classes concretas para o desenvolvimento de cada um dos algoritmos, onde todas implementam a mesma interface Strategy. O acesso aos diferentes algoritmos é feito através de uma classe Context que guarda uma referência para um objeto de uma das classes Strategy concretas. O cliente informa para o Context qual estratégia deseja utilizar, e então executa o algoritmo. Esse padrão também permite que a estratégia desejada seja alterada em tempo de execução.

   Um exemplo de utilização desse padrão é o cálculo de rota realizado por aplicativos de mapa, onde é possível escolher caminhos a pé, de carro ou ônibus, por exemplo.

Figura 25 – Ilustração do padrão Strategy.



Fonte: Guru (2023t)

9. Template Method

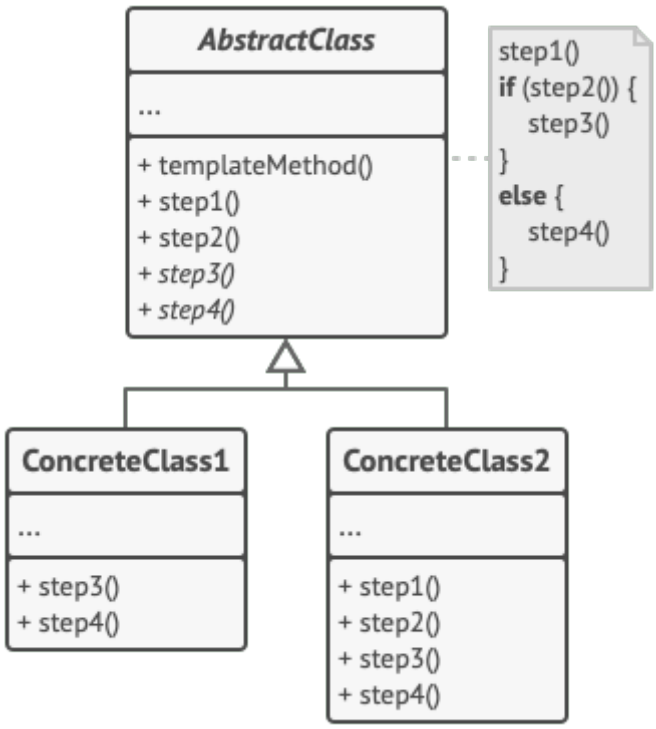
   O padrão Template Method tem o objetivo de definir a estrutura de um algoritmo em uma operação, permitindo que subclasses sobrescrevam o comportamento de seus métodos adaptando os detalhes da implementação para situações específicas, enquanto mantém a estrutura principal intacta.

   Esse padrão requer a criação de uma classe abstrata que declara os métodos que representam cada uma das etapas de um algoritmo, além de um método principal que ordena a chamada desses métodos. Devem também ser criadas classes concretas que sobrescrevem os métodos da classe abstrata de acordo com a necessidade. O único método que não pode ser sobrescrito é o método padrão, garantindo que a sequência original continue a mesma.

   Uma situação prática que utiliza esse conceito é lavagem de roupas. O processo sempre segue uma sequência onde primeiro as roupas são colocadas na máquina de lavar, a máquina entra em operação, e ao final da lavagem as roupas são secas, dobradas e guardadas. Porém, dependendo de algumas características como o tipo de tecido e cor das roupas, o processo escolhido pode ser diferente, deixando as roupas em molho por mais ou menos tempo, com uma agitação mais ou menos agressiva, o processo de secagem pode ser realizado pela própria

máquina ou pendurando as roupas no varal, e assim por diante. A sequência sempre segue o mesmo padrão, mesmo que os detalhes variem.

Figura 26 – Ilustração do padrão Template Method.



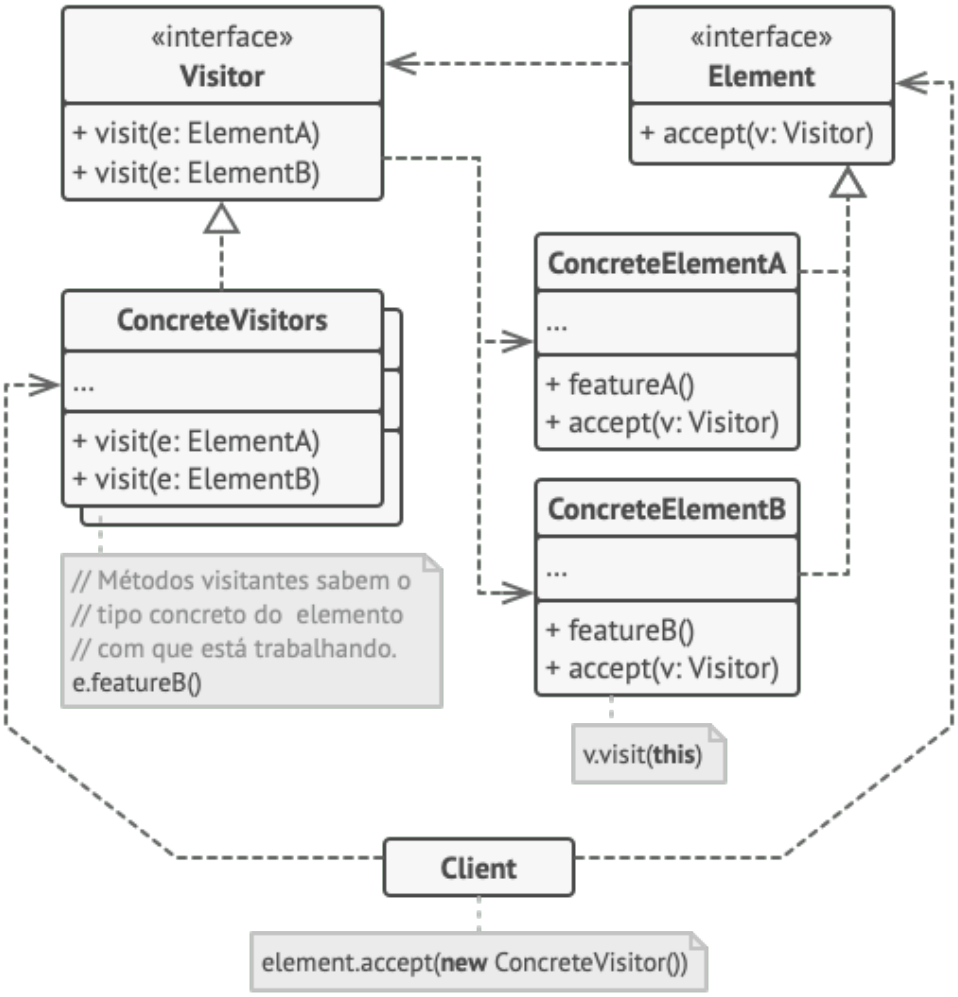
Fonte: Guru (2023u)

10. Visitor

    O padrão Visitor permite separar algoritmos dos objetos sob os quais eles operam, adicionando novos comportamentos às classes sem alterar seu código. Além disso, os objetos podem pertencer a classes diferentes e exigir comportamentos particulares.

    A implementação desse padrão se dá através da criação de uma interface Visitor que declara os diferentes métodos para cada tipo de objeto, as classes Visitor concretas que implementam esses métodos com as suas particularidades, uma interface de elemento que declara apenas um método de aceitação de acesso, e as classes concretas de elementos que implementam essa interface. Nos elementos concretos, o método de aceitação é implementado ao invocar da classe Visitor o algoritmo apropriado à sua classe, basicamente informando a classe Visitor qual algoritmo deve ser aplicado sobre seus elementos.

Figura 27 – Ilustração do padrão Visitor.



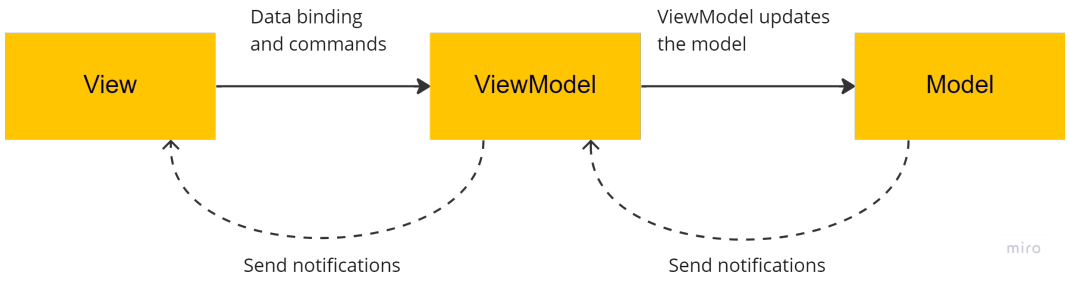
Fonte: Guru (2023v)

2.3 ARQUITETURA MVVM

O Model-View-ViewModel é um padrão de projeto arquitetural utilizado para desacoplar a camada de visualização da aplicação (View) da camada de modelo (Model) (MICROSOFT, 2022b), a qual inclui regras de negócio, acessos a bancos de dados e até serviços externos (CADU, 2022). O padrão MVVM é muito semelhante ao padrão Model-View-Controller (MVC), sendo uma espécie de especialização desse padrão para desenvolvimentos utilizando a tecnologia Windows Presentation Foundation (WPF) da Microsoft, uma linguagem de marcação utilizada para a criação de interfaces gráficas (MICROSOFT, 2022e). A camada ViewModel é uma camada intermediária responsável por associar os dados da camada Model com os elementos gráficos da camada View. Essa associação é realizada através de um mecanismo chamado de *binding*, que abstrai o processo de comunicação entre os objetos de dados e os elementos da interface, permitindo

que alterações nos dados sejam refletidas automaticamente na interface e que edições realizadas pelo usuário através da interface sejam refletidas nos objetos de dados.

Figura 28 – Diagrama da estrutura MVVM.



Fonte: O Autor (2022).

Uma das principais vantagens dessa arquitetura é a clara separação entre a camada de dados e a camada de apresentação, permitindo que as regras de negócio da aplicação sejam alteradas ou que recebam atualizações constantes sem interferir na interface gráfica do programa, facilitando a criação de testes unitários, e até mesmo alterando toda a interface gráfica, desde a disposição dos elementos na tela até a troca de ambiente de desktop para mobile, por exemplo, sem que nenhuma linha de código do backend seja alterada.

3 DESENVOLVIMENTO

Apesar do princípio de funcionamento dos transformadores ser relativamente simples - a transformação de tensão e corrente entre duas bobinas através da indução eletromagnética - os transformadores atuais estão longe de serem simples. Além de toda a evolução no que diz respeito à parte mecânica dos transformadores, como seu formato, dissipadores de calor, dispositivos de segurança, entre outros, a parte elétrica não está restrita apenas à parte ativa, sendo composta também por circuitos de ventilação, controle, sinalização e mais. Por muitos anos, as ferramentas do tipo CAD - Computer-Aided Design, ou design auxiliado por computador - foram as principais ferramentas utilizadas na WEG para o projeto desses circuitos auxiliares. A desvantagem desse tipo de ferramenta é que elas não permitem o cadastro de informações relativas aos dispositivos utilizados, dados do fornecedor ou características físicas, tornando necessário a adição dessas informações manualmente pelos projetistas.

O primeiro passo para a melhoria desse processo foi a substituição de ferramentas do tipo CAD por ferramentas do tipo CAE - Computer-Aided Engineering, ou Engenharia auxiliada por computador - que auxilia desde o desenvolvimento do projeto até o planejamento da manufatura (HOERLLE, 2022). No contexto desse trabalho, a ferramenta CAE que substituiu os antigos processos foi o software E3.Series. O programa permite que os componentes e todos os seus dados construtivos ou de fornecedor sejam cadastrados de antemão, trazendo todas as informações necessárias assim que os componentes são utilizados no projeto, evitando que esse processo fique sob responsabilidade do projetista.

Após essa mudança, uma nova camada de melhorias foi aplicada sobre a ferramenta: o desenvolvimento de automatizações de processos via scripts. Em um primeiro momento, as automatizações criadas para os projetos de circuitos foram implementadas utilizando a linguagem VBSript da Microsoft, usualmente utilizada em automatizações para ferramentas do pacote Office ou scripts para a WEB.

Como os scripts desenvolvidos trouxeram bons resultados e aumentaram a produtividade, a demanda por automatizações só cresceu, e os antigos métodos utilizados para a implementação dos scripts se tornaram insustentáveis. Por esse motivo, a responsabilidade pela manutenção, suporte, melhorias e novos desenvolvimentos dos scripts foi transferida do setor de projetos para o setor de Pesquisa, Desenvolvimento e Inovação e Produtos Digitais (PD&I) da WEG. Essa mudança permitiu um foco muito maior na velocidade e qualidade das entregas, pois não era mais necessário que os projetistas ficassem divididos entre o desenvolvimento de projetos de circuitos e o desenvolvimento de automatizações. Além disso, os conhecimentos presentes dentro do setor de PD&I estavam muito mais alinhados com as boas práticas de desenvolvimento de automatizações.

A partir desses pontos, foi planejado o desenvolvimento de um projeto para refatorar todas as automatizações já utilizadas e unificá-las em uma solução coesa, aproveitando as

ferramentas e conhecimentos do setor de PD&I.

Esses capítulo irá abordar cada um desses tópicos em mais profundidade e na sequência apresentar as etapas de implementação do projeto.

3.1 CONTEXTO

3.1.1 Software E3.Series

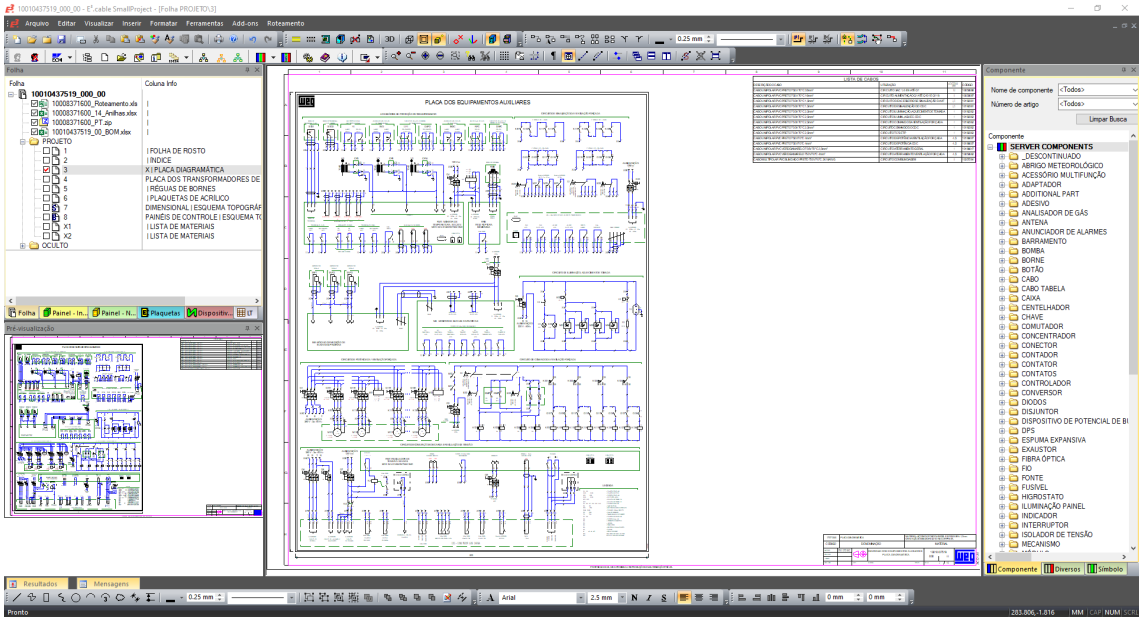
O E3.Series é um software de desenho focado principalmente no projeto de sistemas elétricos e eletrônicos (MARKET, 2022a), mas que também permite o projeto mecânico de painéis de controle, integração entre os sistemas elétrico e mecânico, e até sistemas de fluidos (MARKET, 2022b).

O software possui uma biblioteca de componentes e símbolos completamente customizável, permitindo que sejam utilizados desde dispositivos genéricos até dispositivos completamente configurados. Uma das vantagens do software é a sua natureza voltada a objetos, de forma que os elementos utilizados no projeto não são apenas desenhos, mas instâncias dos componentes cadastrados no banco de dados com todas as informações relevantes para a manufatura do produto final, desde características físicas, como dimensões espaciais, tensão, corrente e mais, até características do fornecedor, como modelo, marca, versões, entre outros.

Mas as suas capacidades de customização do programa vão ainda além, já que disponibiliza uma biblioteca que permite a manipulação dos elementos do software através de scripts que podem ser adicionados à sua interface. A biblioteca permite essa interação através do sistema Component Object Model (COM), o qual disponibiliza componentes binários de software que podem ser acessados e que servem de ponte entre os comandos do script e os elementos internos do programa (MICROSOFT, 2022a). O software possui dois tipos de scripts: os chamados add-ons, que podem ser disparados pelo usuário, e os triggered scripts, que são disparados automaticamente pelo próprio software dado certo gatilho, como abertura ou encerramento do software, inserção ou remoção de dispositivos no projeto, alteração das características de um dispositivo, entre outros.

Na Figura 29 pode-se observar a área de trabalho do software E3.Series contendo uma folha do tipo esquemático, ou seja, uma folha dedicada ao desenho dos dispositivos e conexões que compõe o circuito de comando e controle.

Figura 29 – Folha do tipo esquemático no software E3.Series.



Fonte: O Autor (2023).

Já a Figura 30 apresenta uma folha do tipo painel, que possui um terceiro eixo de referência, permitindo a adição de componentes tridimensionais. Por se tratar do projeto mecânico do painel de comando, faz sentido que essa folha possua mais uma dimensão quando comparada com o esquemático do projeto. No exemplo da imagem, os componentes estão dispostos em seus devidos lugares, mas ainda sem possuírem nenhuma conexão elétrica, ou seja, sem nenhum roteamento.

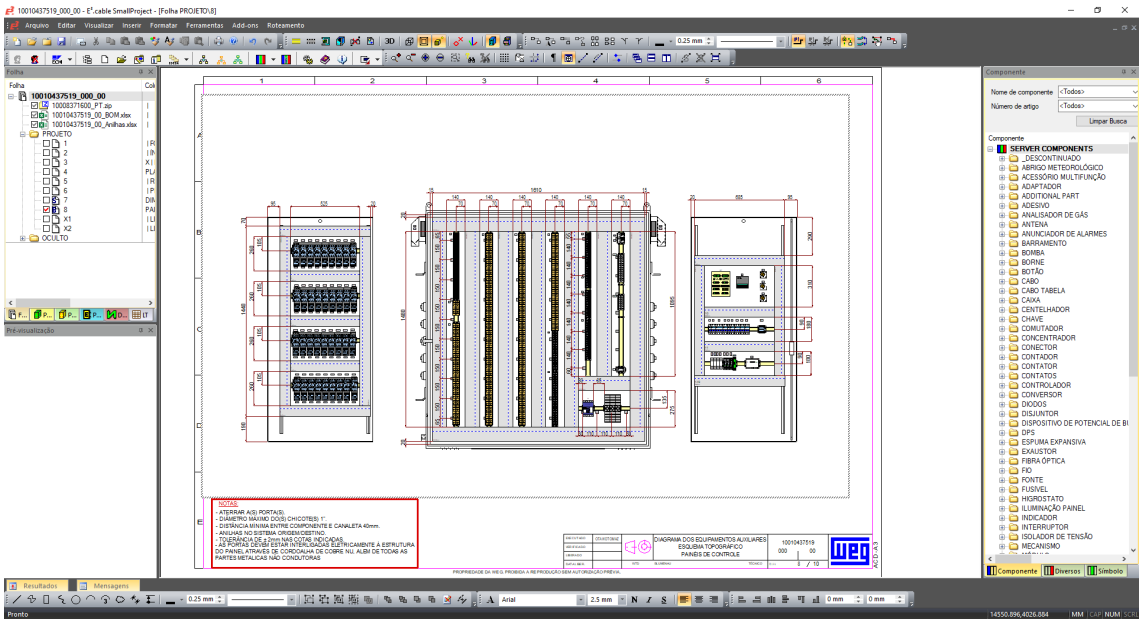
A Figura 31 exibe mais um exemplo de projeto mecânico do painel de controle, mas agora roteado, onde todos os dispositivos estão conectados de acordo com as conexões desenhadas no projeto esquemático.

3.1.2 Antigos Scripts e Linguagem VBScript

Percebendo a possibilidade de automatizar diversos processos dentro dos projetos de comando e controle, o próprio setor de projetos começou o desenvolvimento de scripts. A documentação do software inclusive possui uma seção dedicada às suas principais classes internas e os métodos disponibilizados por elas, onde todos os exemplos são dados utilizando a linguagem VBScript. A linguagem faz parte da família Visual Basic for Applications (VBA) da Microsoft, geralmente utilizadas na automação de processos em softwares do pacote Office (MICROSOFT, 2022f). Apesar da sintaxe bastante semelhante, existem algumas particularidades à linguagem VBScript, como:

a) Não tipada: a linguagem possui apenas um tipo de dado principal, chamado variante. Dessa forma, o desenvolvedor não pode escolher o tipo de dados

Figura 30 – Folha do tipo painel sem roteamento.



Fonte: O Autor (2023).

que a variável irá abrigar com antecedência, ficando a cargo do interpretador decidir qual o subtipo mais adequado, diferenciando entre cadeias de caracteres e diferentes tipos de valores numéricos (MICROSOFT, 2022d);

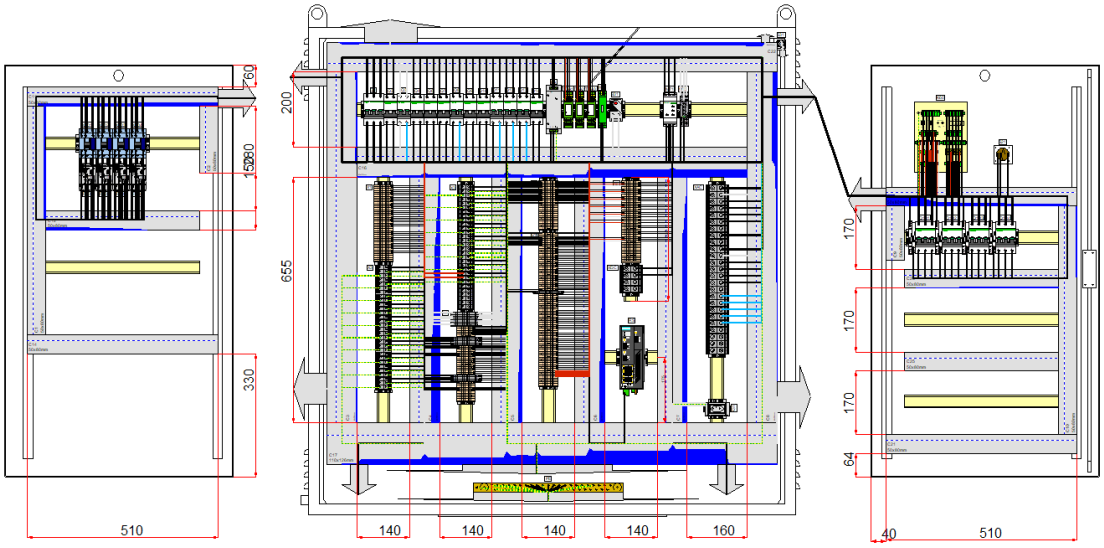
b) Não compilada: a linguagem é interpretada, significando que o código é convertido para linguagem de máquina linha a linha cada vez que o script é iniciado, ao contrário de linguagens compiladas que são previamente convertidas para linguagem de máquina, evitando a necessidade de conversão toda vez que o script é executado e geralmente ganhando no quesito performance;

c) Não possui suporte a early binding: em linguagens compiladas, por padrão o compilador conhece as classes e objetos com as quais está trabalhando, permitindo também o uso de tecnologias como o Intellisense que fornece sugestões para autocompletar o código com base nas propriedades e métodos dos objetos. Sendo uma linguagem interpretada, o VBS recorre à técnica de late binding, definindo o tipo dos objetos apenas em tempo de execução.

Por esses motivos a linguagem se torna uma boa escolha para pequenas automatizações, já que rapidamente permite o desenvolvimento de scripts sem um grande conhecimento prévio de programação e sem a complexidade adicional que as linguagens compiladas trazem. Porém, com a demanda crescente por automatizações cada vez mais complexas, o uso da linguagem começou a trazer grandes desvantagens.

Por exemplo, apesar de permitir a criação de classes com seus respectivos atributos e métodos, a linguagem não é amigável ao desenvolvimento de scripts orientados a objetos,

Figura 31 – Folha do tipo painel com roteamento.



Fonte: O Autor (2023).

não sendo a melhor escolha para soluções orientadas a objetos mais complexas. Além disso, de qualquer forma todas as definições de classes deveriam estar contidas dentro do mesmo arquivo que o resto do script, tornando-o quase tão extenso quanto um desenvolvimento estruturado. Uma segunda opção seria separar todas as definições de classes em arquivos separados que deveriam ser mantidos na mesma pasta que o arquivo do scritp, mas sem que houvesse nenhuma maneira de mostrar a dependência entre os arquivos.

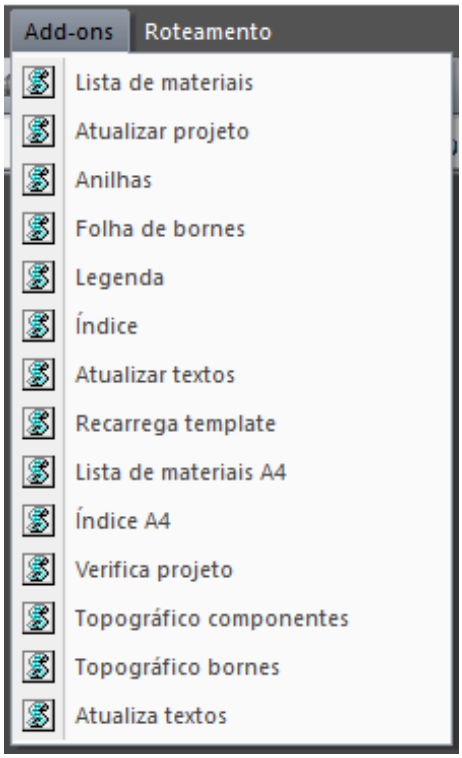
A maneira como as automatizações foram desenvolvidas originalmente se deu pela criação de diversos arquivos, um para cada automatização. Sem que tenham sido criados métodos para a realização de ações básicas, como a conexão do script com o projeto do E3.Series, obtenção de todos os componentes ou símbolos do projeto, entre outros, esses fragmentos de código eram constantemente reescritos em todos os scripts, e se eventualmente fosse encontrada uma maneira mais otimizada para realizar alguma ação, cada um dos scripts precisaria ter seu código reescrito para atender a nova metodologia.

Outras desvantagens incluem a inexistência de uma IDE oficial, com capacidade para depuração de código e sugestões para autocompletar o código. Todos esses pontos dificultavam o desenvolvimento e manutenção dos scripts, já que não era possível verificar quais métodos e atributos os objetos possuíam, constantemente exigindo a busca por algum tipo de documentação. (MICROSOFT, 2022c), as depurações precisavam ser feitas através de comandos como *print* para descobrir o valor das variáveis, enfim, uma série de dificuldades que tornavam essa metodologia de desenvolvimento insustentável a longo prazo.

Na interface do E3 a aba de automatizações era exibida conforme mostra a Figura

32, onde cada um dos itens correspondia a um arquivo VBS dedicado a uma automatização específica. Pode-se perceber a existência de scripts como “Lista de materiais” e “Lista de materiais A4”, e também de “Índice” e “Índice A4”. Isso acontecia pois na metodologia de desenvolvimento antiga era mais intuitivo copiar o código inteiro de um script, colar em outro arquivo e alterar um único parâmetro, como nesses casos o tamanho da folha, do que desenvolver métodos com funções específicas e apenas alterar o parâmetro de acordo com a escolha do usuário.

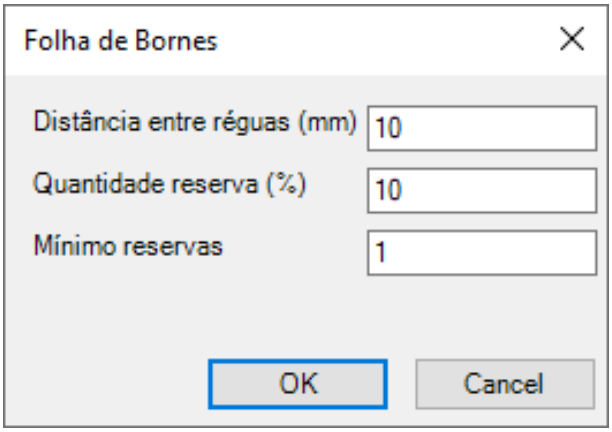
Figura 32 – Aba com os antigos scripts em VBS.



Fonte: O Autor (2023).

Os poucos scripts que possuíam uma interface para a definição de parâmetros não chegavam a apresentar telas muito mais complexas do que a apresentada na Figura 33, contendo apenas caixas de texto para inserção dos valores e botões para confirmar ou cancelar a execução do script.

Figura 33 – Antiga interface do script Folha de Bornes.



Fonte: O Autor (2023).

3.1.3 Antigo processo de liberação

O antigo processo de liberação dos scripts consistia em fazer o upload dos arquivos para uma pasta em nuvem compartilhada entre os projetistas e inserir manualmente a nova automatização na interface de cada usuário, criando um novo item de menu que apontava para o arquivo recém copiado. Mais uma vez foi possível identificar um processo não sustentável, já que o número de suportes para a inserção de novas automatizações aumentaria proporcionalmente ao número de projetistas, e delegar essa função para os usuários também não era uma opção, pois não seria possível garantir que todos os projetistas o fariam, possivelmente existindo dezenas de versões diferentes dos scripts sendo utilizadas simultaneamente, ou até scripts não sendo utilizados. Além do mais, dificuldades como essas em um ambiente com prazos apertados geralmente incentivam os usuários a desenvolverem suas tarefas da primeira maneira que encontrarem, sem se importarem em ir atrás de automatizações que exigem ainda mais do seu tempo.

3.2 PLANEJAMENTO

3.2.1 Linguagem

A nova linguagem escolhida para a implementação das automatizações foi C#. A escolha da linguagem se deu principalmente por possuir todos os recursos faltantes na linguagem VBScript, como uma boa documentação, ambiente próprio de desenvolvimento, possibilidade de depuração de código, diversas bibliotecas nativas para integração com aplicações Microsoft, além de já ser a linguagem utilizada por padrão nos desenvolvimentos do setor de PD&I.

3.2.2 Interpretação dos scripts em VBS

Com a intenção de construir um melhor planejamento para a execução do projeto, previamente foi acordado que seria realizada uma leitura de todos os scripts em VBS, com a intenção de avaliar o tamanho total do escopo do projeto, descobrir as principais tarefas executadas dentro de cada script e entender melhor como se dava a manipulação de elementos dentro do software E3 via código. Foram criadas cópias desses scripts, de maneira que as automatizações em produção não corressem o risco de serem afetadas, e adicionados diversos comentários ao longo dos códigos para elucidar o que e como cada processo era automatizado, além de já servir de base para o desenvolvimento de lógicas mais eficientes para realizar essas funções.

3.2.3 Definição das tarefas

Após finalizar a interpretação dos scripts, o projeto começou a ser planejado de fato. Através da plataforma Jira, muito utilizada para o planejamento, controle e análise de projetos de desenvolvimento, foram criadas tarefas para a conversão de cada um dos scripts, além de tarefas auxiliares como: planejamento da estrutura de projetos e classes, metodologia de liberação dos scripts convertidos, metodologia de testes, entre outras. Com base no tamanho e aparente complexidade dos scripts em VBS, foram adicionados pontos de complexidade para as tarefas de conversão de cada um dos scripts. Também foram agendadas reuniões com projetistas e usuários do E3 para entender quais eram as automatizações que mais causavam impacto na sua produtividade. Com base nos pontos de complexidade e impacto para os projetistas foi possível definir uma ordem de prioridade para as tarefas, com o objetivo de entregar a maior quantidade de valor o mais breve possível para os usuários. Por fim, considerando as ordens de prioridades definidas, todas as tarefas foram distribuídas em sprints de forma que cada sprint abrigasse aproximadamente a mesma quantidade de trabalho. Nesse momento ainda não era possível calcular uma data para a entrega final do projeto, mas assim que o desenvolvimento começasse seria possível cruzar os pontos de complexidade da sprint e o tempo total despendido, para assim estimar um tempo total de execução.

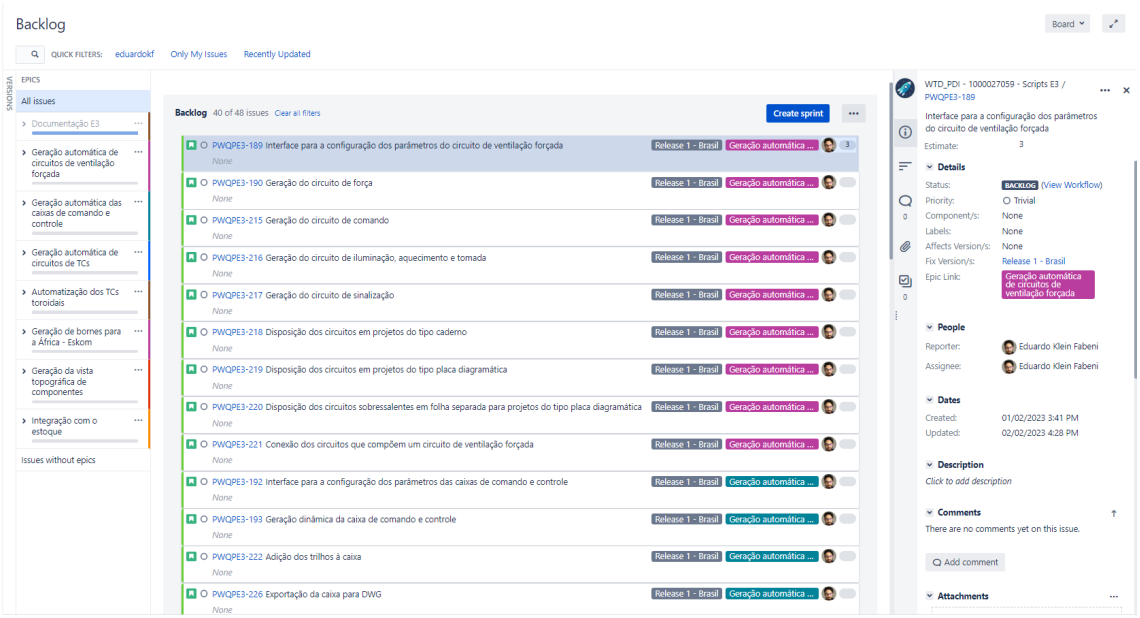
As tarefas no Jira são geralmente compostas por épicos, histórias e tarefas. Os épicos estão relacionados a entrega final para o usuário, como a geração automática de listas de materiais. As histórias representam funcionalidades específicas que entregam valor para o usuário, como a possibilidade de escolher o tamanho da folha da lista de materiais, definir que tipo de ordenação será aplicada à lista, entre outros, enquanto as tarefas são especificações relevantes aos desenvolvedores, descrevendo as tarefas que devem ser executadas para implementar uma história.

A Figura 34 demonstra um exemplo de planejamento de atividades no Jira, contendo seus respectivos épicos e histórias. O modo de trabalho pode ser definido entre Scrum e

Kanban, dependendo das necessidades do projeto e da metodologia definida pelo time.

Além de permitir o planejamento e acompanhamento das entregas, a plataforma também fornece uma série de relatórios sobre o andamento do projeto que permitem extrair informações como quantidade de entregas, velocidade, complexidade do trabalho sendo executado, entre muitos outros, que podem ser utilizados pelo Scrum Master para definir estratégias ainda melhores para as etapas posteriores de desenvolvimento.

Figura 34 – Exemplo de tarefas criadas no Jira.



Fonte: O Autor (2023).

3.3 EXECUÇÃO

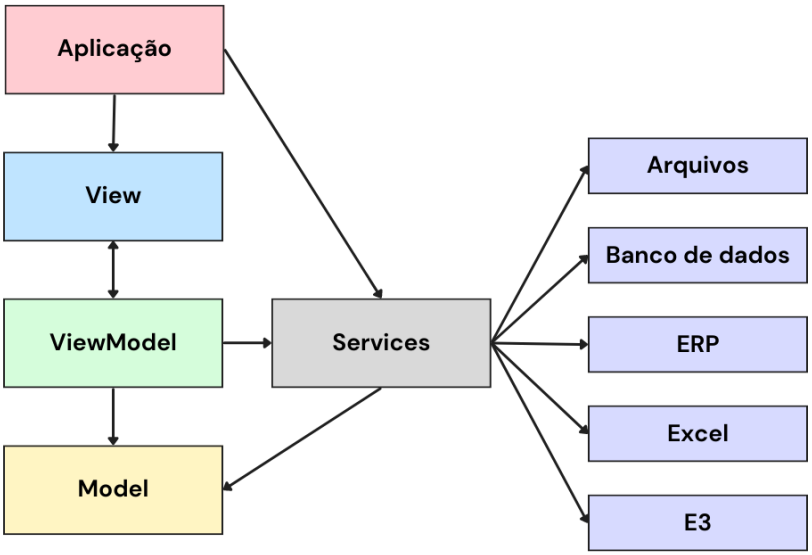
3.3.1 Primeiros Passos

O primeiro passo da execução do projeto foi a criação da estrutura sobre a qual toda a solução seria construída. Foram criados diversos projetos para as diferentes responsabilidades da aplicação. O projeto E3.View ficou responsável por conter apenas as interfaces gráficas que eventualmente seriam utilizadas para permitir a manipulação dos parâmetros das automatizações. O projeto E3.Model foi criado para armazenar todas as classes que representam elementos do E3, como componentes, dispositivos, folhas, símbolos, além de elementos menos diretos, como os atributos de cada um dos modelos citados anteriormente, propriedades do projeto, e até mesmo os menus do programa. O projeto E3.ViewModel contém todas as classes responsáveis por interligar os elementos de modelo às interfaces, implementando boa parte da lógica dos scripts e acessando os serviços disponibilizados pelo projeto E3.Services. O projeto E3.Services é um dos mais complexos da solução, pois fornece acesso à diversas outras classes, como a DAOController, criada para a interação

com bancos de dados, SAPController, para integração com o SAP ERP, e também com uma das classes mais importantes da solução, E3Controller, classe que implementa todos os métodos utilizados para a manipulação dos elementos internos do E3.

Fora os projetos já citados, é relevante citar também que boa parte dos scripts não necessita de uma interface gráfica. Para esses casos também foi criado o projeto E3.Scripts, que implementa a lógica de scripts que não precisam da configuração de nenhum parâmetro e podem simplesmente ser executados por trás da aplicação, manipulando elementos do projeto ou gerando algum documento de saída.

Figura 35 – Diagrama das camadas de responsabilidade do projeto.



Fonte: O Autor (2023).

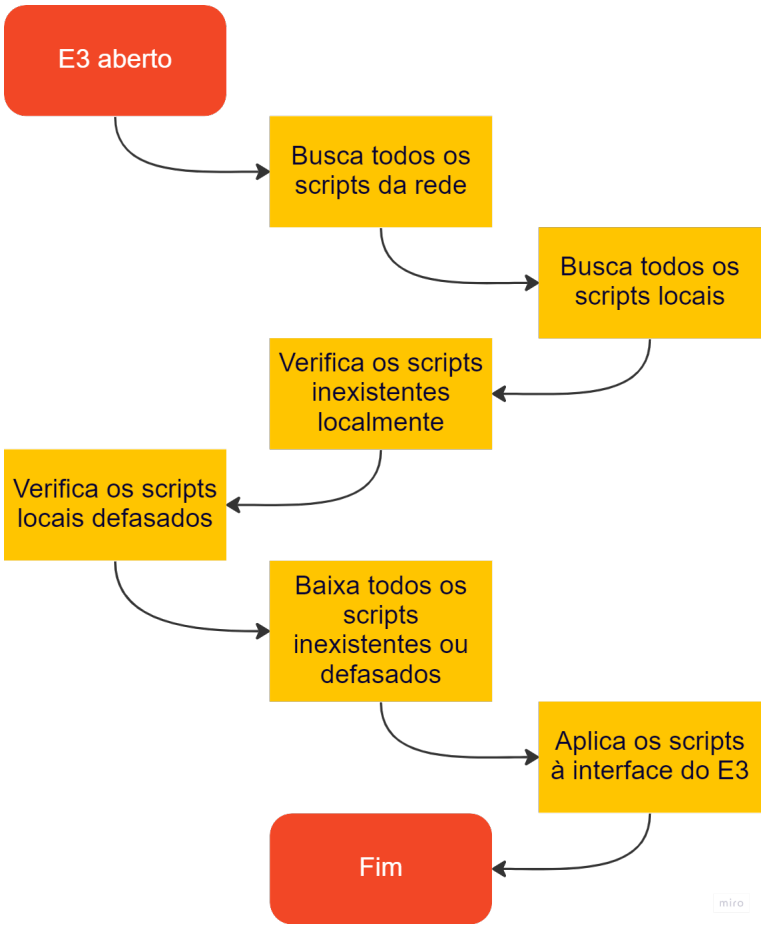
3.3.2 Planejamento e implementação de um método de liberação dos scripts

Desde o começo do planejamento, um dos principais objetivos era entregar valor aos projetistas o mais breve possível, como sugere a metodologia ágil Scrum. Assim, não fazia sentido esperar até o final do projeto para pensar em uma maneira de entregar os scripts convertidos aos projetistas, de forma que logo após finalizar a conversão do primeiro script deu-se início ao planejamento de uma metodologia de liberação. Ao estudar a documentação do E3, verificou-se que o software possui gatilhos que são disparados por eventos específicos, como durante a abertura ou o fechamento do programa. Ao constatar esse fato, foi natural pensar em um atualizador que é executado sempre que o programa é iniciado.

A ideia era aproveitar a rede interna da empresa para armazenar os executáveis dos scripts refatorados, que seriam então baixados para o computador dos projetistas e executados localmente.

A execução dessa solução se deu através da implementação de um novo script, focado na verificação e aplicação de atualizações. O atualizador fazia o levantamento de todos os scripts refatorados na rede e verificava se o computador local possuía esses arquivos, além de checar a última data de modificação de cada arquivo. Em casos onde o projetista não possuía alguma das automatizações presentes na rede ou a data de modificação dos scripts locais estivessem defasadas, o atualizador realizava o download dos scripts necessários e fazia a inserção dessas automatizações na interface do E3 do usuário. O script atualizador foi cadastrado como um triggered script disparado durante o início da aplicação, de forma que sempre que uma nova automatização ou atualização fosse liberada, todos os usuários receberiam as mudanças automaticamente.

Figura 36 – Diagrama da lógica do script atualizador.



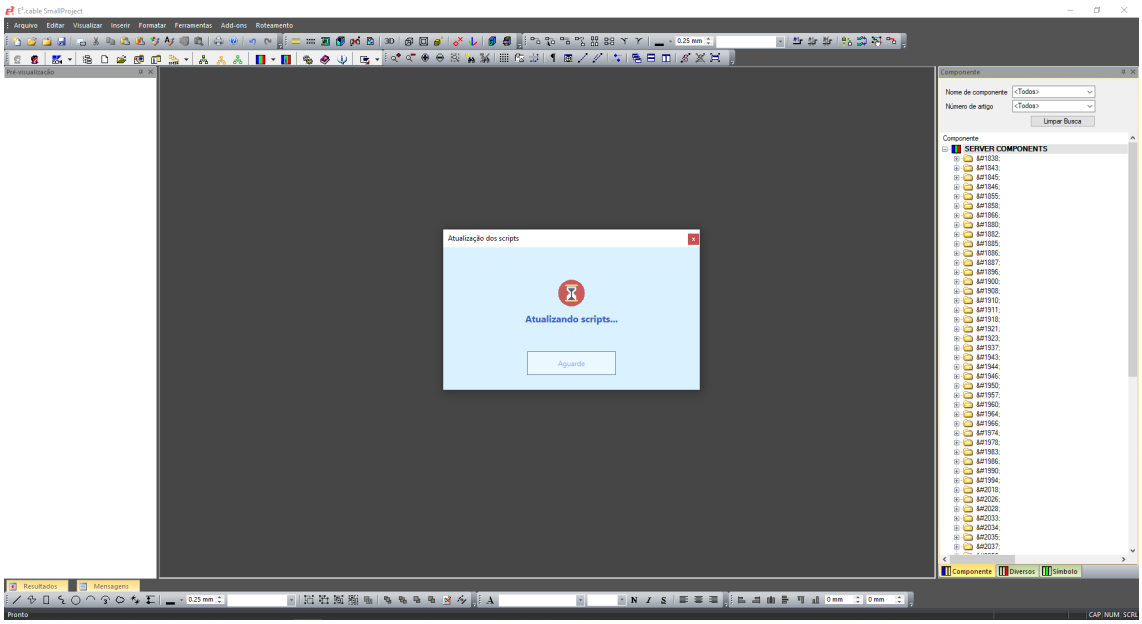
Fonte: O Autor (2023).

É importante citar que durante esse período nenhum script antigo era retirado da interface, permitindo que os projetistas ainda pudessem utilizar as antigas automatizações,

bem como testar e aproveitar as melhorias dos scripts refatorados.

A Figura 36 exibe a janela de atualização que é disparada sempre que o E3 é inicializado. Quando não existem atualizações esse processo não costuma demorar mais do que poucos segundos, podendo se estender mais no caso de liberação de versões ou de novos scripts, quando é necessário fazer o download dos arquivos e aplicar à interface.

Figura 37 – Atualizador sendo executado durante o início do software.



Fonte: O Autor (2023).

3.3.3 Conversão dos scripts

Ao continuar o desenvolvimento do projeto, cada sprint demorava em torno de duas semanas, com alguns scripts especialmente complexos podendo chegar a até quatro semanas, como foi o caso da geração automática da lista de materiais do projeto. Esse script, diferente de todos os outros, havia sido desenvolvido pela empresa representante do E3 no Brasil, e apontava para métodos presentes em outros dois arquivos VBS. Por não possuir uma IDE amigável para navegar entre os diferentes arquivos e nem uma maneira de depurar o código, sempre que um método era invocado era necessário buscar por ele nos outros arquivos de forma manual, onde cada arquivo possuía centenas ou até milhares de linhas. Para esse caso, portanto, optou-se por não converter a lógica do antigo script, mas sim escrevê-la do zero. Esse fato explica o porquê desse desenvolvimento ter demorado o dobro do tempo normal, mas também foi um dos momentos de maior aprendizado com relação ao funcionamento interno do E3 e sua manipulação via código.

Ao evoluir no desenvolvimento do projeto, além de uma familiarização cada vez maior com as particularidades do desenvolvimento para o E3, a classe E3Controller ganhava

cada vez mais métodos que podiam ser reaproveitados em outras automatizações. Com isso, observou-se um aumento na velocidade de conversão dos scripts, que às vezes chegavam a uma semana ou até menos. Nessa nova etapa de desenvolvimento e com mais ferramentas disponíveis, começou-se a perceber diversos trechos de código dos scripts que poderiam ser melhorados. Entre finalizar a sprint com uma semana de antecedência ou aproveitar o tempo restante para atuar em melhorias, optou-se pela segunda opção. Foi principalmente nessa fase que as melhorias trazidas pelo planejamento e utilização de ferramentas mais adequadas ao desenvolvimento de automatizações passaram a ganhar destaque, com os scripts se tornando mais rápidos, mais robustos contra erros, ou até mesmo gerando resultados melhores.

3.3.4 Metodologia de trabalho

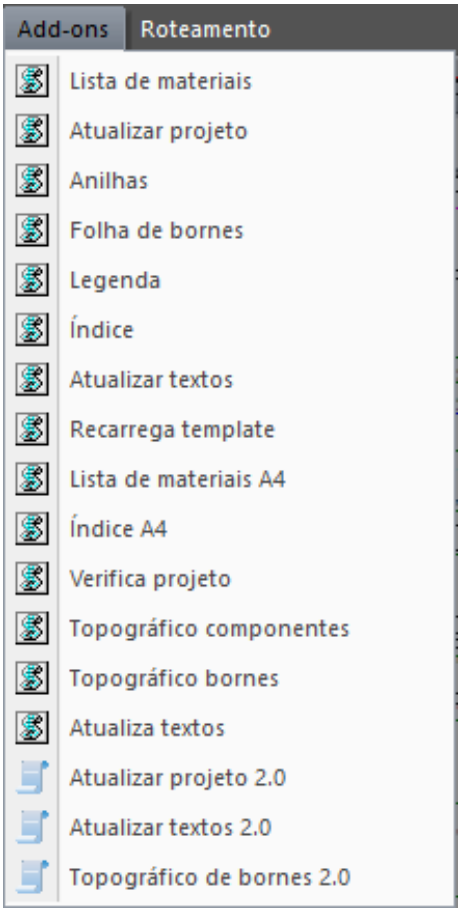
Uma semana normal de desenvolvimento consistia em criar um novo projeto em C# para o script definido na sprint, percorrer o antigo código em VBS e convertê-lo para a nova linguagem, aproveitando também para corrigir erros e implementar melhorias. Ao fim da conversão, os testes eram realizados ao comparar os resultados gerados pelo script refatorado com os resultados gerados pela automatização original. Após uma bateria de testes, que variavam de acordo com a finalidade de cada script, a nova versão era compilada e enviada para a pasta da rede que continha todas as refatorações. Por consequência, da próxima vez que os projetistas iniciassem o software E3, o novo script seria adicionado à interface ficando disponível para utilização.

3.3.5 Substituição dos scripts antigos

Com o fim da conversão do último script, foi definida uma data para a desativação dos scripts em VBS e sua substituição pelos scripts em C#. O mesmo atualizador utilizado para buscar novos scripts na rede foi modificado para identificar os scripts em VBS ainda presentes na interface do software e, a partir da data definida, removê-los do programa, mantendo assim apenas os scripts atualizados.

Na Figura 38 pode-se ver a aba de scripts contendo todas as automatizações em VBS seguidas pelos primeiros scripts convertidos para C#. Dessa forma era possível que os projetistas testassem e já tirassem proveito das novas versões liberadas, mas mantendo a liberdade de escolha caso preferissem usar as ferramentas mais familiares.

Figura 38 – Aba de scripts com as primeiras automatizações convertidas para C#.



Fonte: O Autor (2023).

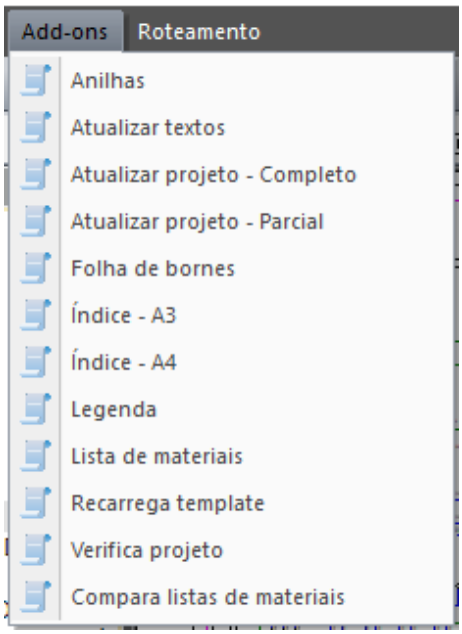
Após a data final de atualização, a aba de scripts passou a ficar como mostrado na Figura 39, sem mais nenhum dos scripts antigos.

3.3.6 Melhorias e novos desenvolvimentos

Durante a conversão dos scripts foram sendo observadas diversas oportunidades de melhoria não apenas na lógica das automatizações, mas também em novas funcionalidades que só poderiam ser proporcionadas após a refatoração dos scripts e com a utilização de ferramentas mais adequadas.

Um dos problemas observados foi a dificuldade em identificar a causa de erros quando alguma exceção era disparada. Na maioria das vezes o script era encerrado sem maiores explicações, ficando sob responsabilidade do projetista descobrir o problema e corrigi-lo. Assim, foi implementada uma tela de avisos com a função de auxiliar os projetistas a identificar e corrigir os problemas com o mínimo de necessidade de suporte por parte do PD&I, apresentando mensagens compostas por três campos: local do erro, motivo e possível solução.

Figura 39 – Aba de scripts apenas com os scripts atualizados.



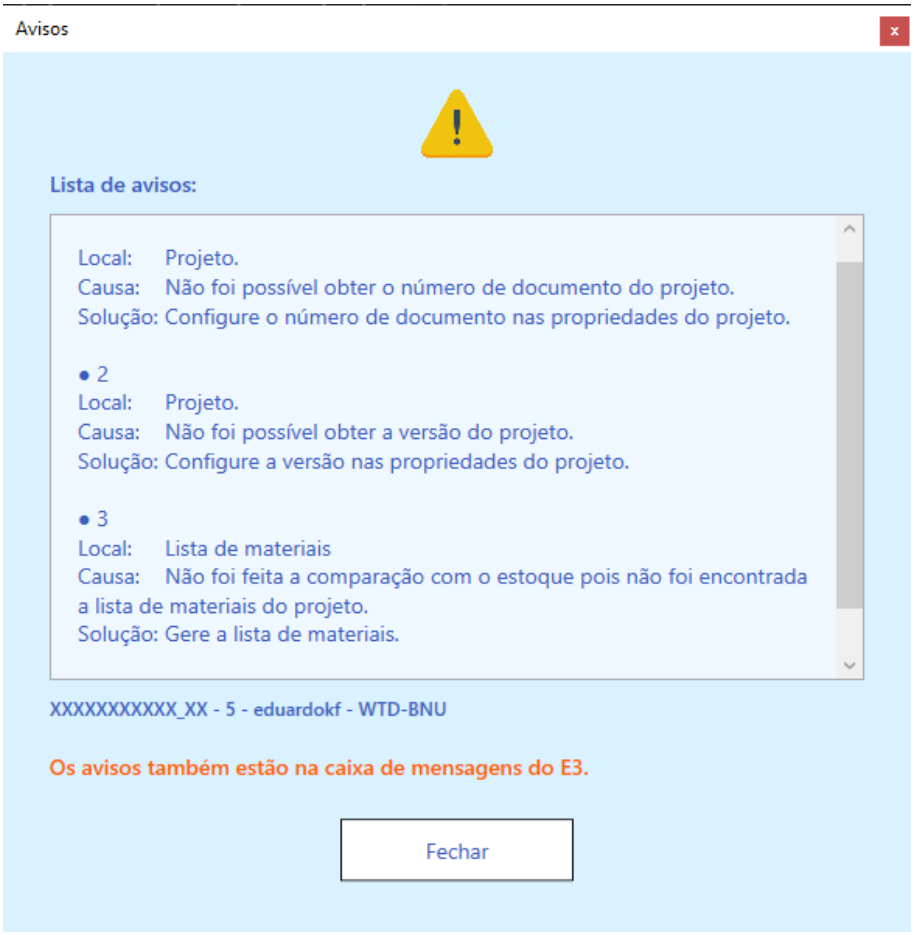
Fonte: O Autor (2023).

A Figura 40 exibe uma janela de avisos com três inconsistências verificadas ao longo da execução de um script. Cada um dos erros é salvo em uma lista de mensagens de erro, que são exibidas em ordem ao final da execução. Como todos se tratam de problemas conhecidos, foi possível cadastrar mensagens de solução específicas para os problemas previstos que são exibidas para os projetistas, ajudando-os a resolver os problemas sem a necessidade de suporte.

Quando uma exceção é causada por algum bug do script, ou seja, quando a responsabilidade do erro está no lado do desenvolvedor, geralmente não é possível informar uma solução para o usuário, justamente por se tratar de um erro inesperado. Nesses casos a origem do erro e o motivo da exceção ainda são informados, mas no lugar de uma mensagem de solução específica para o problema ocorrido, é exibida uma mensagem incentivando os projetistas a abrirem um chamado de suporte para o setor de PD&I requisitando a correção do bug. Já quando a responsabilidade do erro recai sobre o projetista, no caso de algum procedimento de projeto não ter sido executado corretamente ou quando as configurações do projeto não foram definidas, os erros que podem ser gerados são mais previsíveis, permitindo que soluções sejam cadastradas de antemão e exibidas para o usuário quando necessário.

Outra funcionalidade implementada foi a geração de logs de execução sempre que uma automatização é disparada. No arquivo de log são salvas informações como: nome do script executado, data, hora, tempo total de execução e, no caso de alguma exceção ter sido disparada, a mensagem de erro gerada, bem como o nome da classe, método e linha

Figura 40 – Tela de avisos com exemplos de mensagens de erro.



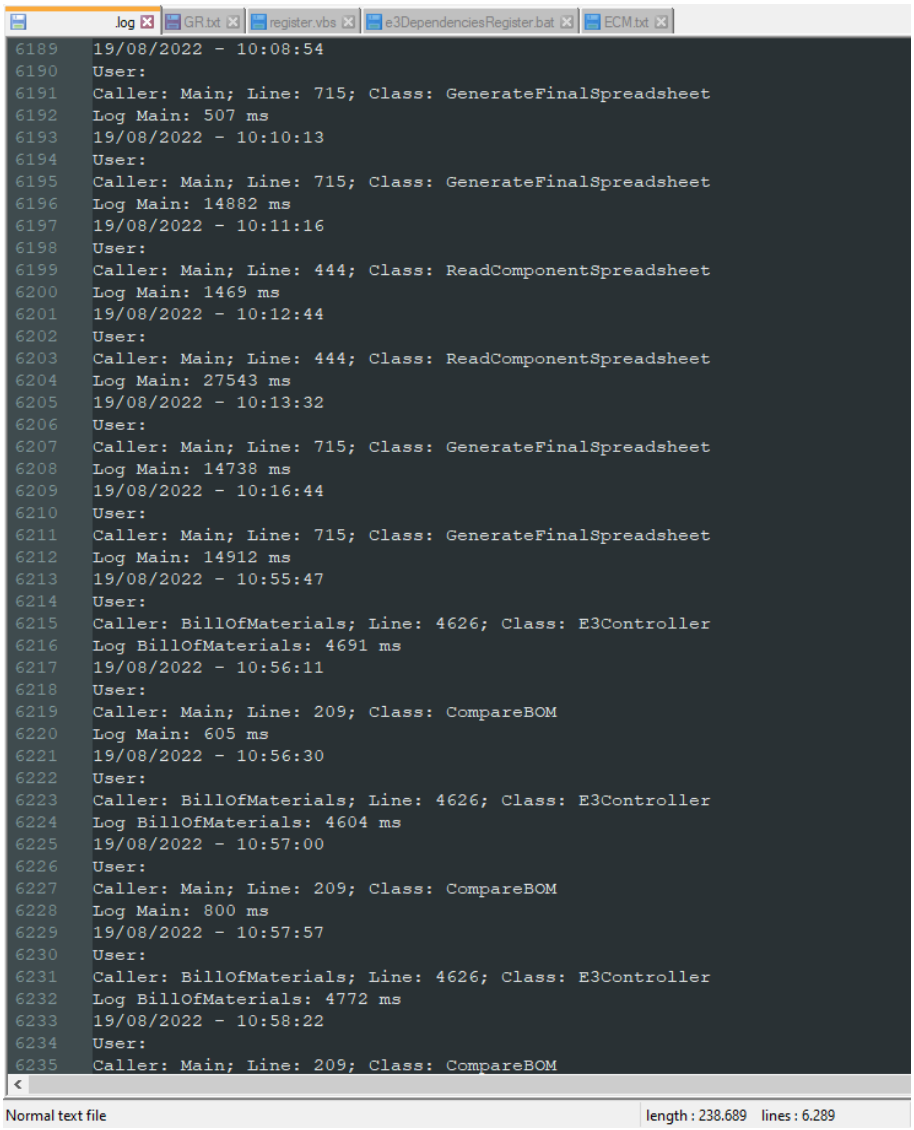
Fonte: O Autor (2023).

específica que disparou o erro. Essa funcionalidade permitiu uma identificação muito mais rápida da causa dos erros e, por consequência, uma prestação de suporte mais eficiente.

Um fragmento de um log de execuções pode ser visualizado na Figura 41, contendo a data, hora, usuário, classe (script) e tempo total de execução.

Apesar dessas melhorias, sempre havia um fator humano que contribuía para que erros se perpetuassem, que era a falta de comunicação sobre a ocorrência dos erros. Devido aos prazos de entrega ou a falta de conhecimento sobre a quem relatar os problemas, era comum que os projetistas encontrassem alguma forma de contornar os erros, mesmo que fosse executando o trabalho manualmente, no lugar de buscar uma solução definitiva. Aproveitando o log de execuções já bem estruturado, foi implementada mais uma função: a de envio automático de e-mails para o suporte do PD&I sempre que uma exceção fosse disparada, tendo como conteúdo do e-mail a mensagem de erro do log. Dessa forma, mesmo que nenhum projetista relatasse os problemas que estava enfrentando, era possível para o suporte tomar ciência do ocorrido, tentar reproduzir o erro, implementar uma solução, e liberar a versão corrigida do script assim que o problema fosse resolvido.

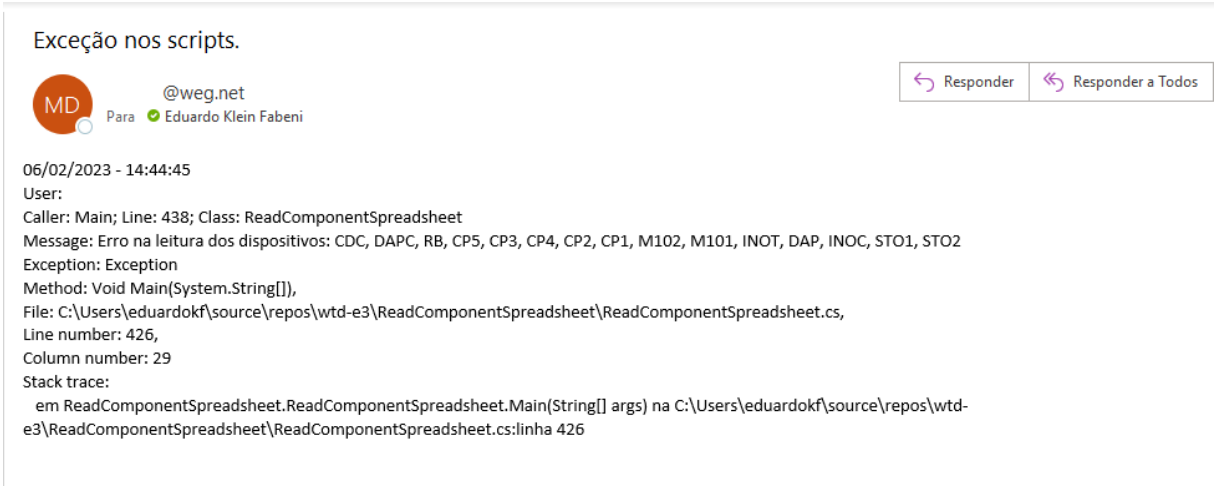
Figura 41 – Exemplo de arquivo de log.



Fonte: O Autor (2023).

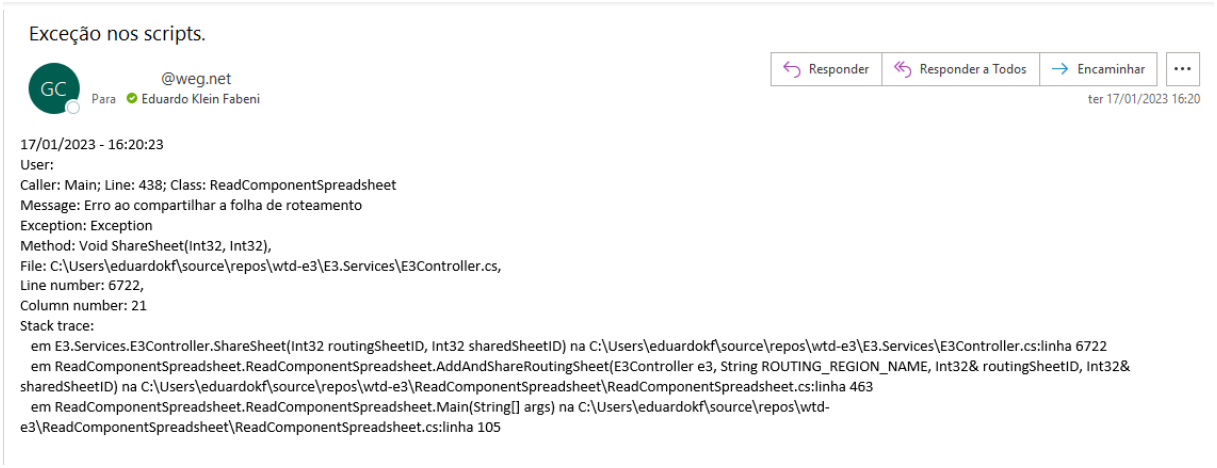
As figuras 42 e 43 exibem exemplos de e-mail recebidos pelo suporte. As mensagens são bastante parecidas com as salvas nos logs de execução, contendo data, hora, usuário, script executado e rastreamento de pilha do erro.

Figura 42 – Primeiro exemplo de e-mail de erro.



Fonte: O Autor (2023).

Figura 43 – Segundo exemplo de e-mail de erro.



Fonte: O Autor (2023).

Os arquivos de log ainda puderam ser utilizados para mais uma função importante: histórico e levantamento de dados sobre as automatizações executadas. Como as mensagens de log são concatenadas ao arquivo em vez de sobrescrevê-lo, é possível ter informações sobre todos os scripts executados por todos os projetistas desde que essa funcionalidade foi liberada. Ainda, como as mensagens de log seguem um padrão bem definido, foi possível escrever um script auxiliar que interpreta essas informações e alimenta uma tabela com a quantidade total de execuções de cada um dos scripts, o tempo total que cada um dos scripts passou em execução, o tempo médio de execução, e também o número de exceções geradas para cada script, tudo isso separado em abas individuais para cada

um dos projetistas. A partir desses dados foi possível verificar quais são os scripts mais utilizados pelos projetistas, os scripts mais lentos, comparar os resultados para um mesmo script entre diferentes versões liberadas e mais.

As figuras 44 e 45 mostram o histórico de execução de scripts de dois projetistas. Através desses dados é possível verificar uma adesão muito maior às automatizações por parte do primeiro projetista, o que pode indicar uma carga maior de projetos, utilização do software E3 para mais tipos de projetos, ou mesmo uma tendência maior para a utilização das ferramentas fornecidas. De qualquer forma, também é possível identificar quais as automatizações menos utilizadas, e que portanto seriam classificadas com uma prioridade mais baixa caso sejam reportados bugs para mais de um script simultaneamente, ou que não necessitam de uma otimização muito urgente caso possuam um tempo de execução mais longo, justamente por serem executados em ocasiões mais específicas e que talvez não justifiquem o trabalho.

Figura 44 – Primeiro exemplo de tabela de execução de scripts.

DADOS DOS SCRIPTS - 20/02/2023 10:41:49				
USUÁRIO 1				
Scripts	Execuções	Tempo total (s)	Tempo médio (s)	Exceções
Check-in before	1308	10497	7,9	28
Check-in after	1283	3888	3,1	0
Anilhas	376	5156	13,9	2
Verifica projeto	295	8849	29,6	5
Lista de materiais	256	2824	11,3	0
SAP BOM Upload	234	2041	8,8	3
5 - Cria canaleta	119	1470	12,7	0
Compara BOM	116	110	0,9	0
6 - Gera listagem final	96	3726	39,2	0
1 - Configura roteamento	92	1	0	0
3 - Gera planilha mecânico	77	912	12,2	0
4 - Lê planilha mecânico	73	3439	47,8	2
7 - Corrige cabos tabela	35	38	1,1	0
Atualiza Workspace	26	67	2,4	2
Índice	25	20	0,8	1
Folha de bornes	18	114	6,7	0
Recarrega template	11	29	2,7	0
Atualiza textos	11	33	3,1	0
Atualiza projeto completo	9	757	84,1	0
Legenda	4	9	2,3	0
Topográfico bornes	2	11	5,7	0
2 - Cria montagens	1	0	0,5	0

Fonte: O Autor (2023).

Figura 45 – Segundo exemplo de tabela de execução de scripts.

DADOS DOS SCRIPTS - 21/02/2023 09:36:53				
USUÁRIO 2				
Scripts	Execuções	Tempo total (s)	Tempo médio (s)	Exceções
Check-in before	320	1034	3,3	4
Check-in after	316	563	1,8	0
Verifica projeto	44	1283	30,6	0
SAP BOM Upload	31	206	6,9	0
Atualiza Workspace	22	60	2,6	1
Lista de materiais	19	229	12,1	0
Anilhas	18	194	10,8	0
3 - Gera planilha mecânico	12	46	4,2	0
Folha de bornes	3	23	7,7	0
Índice	3	1	0,5	0
1 - Configura roteamento	2	0	0	0
Recarrega template	1	0	0,5	0

Fonte: O Autor (2023).

Como os recursos da tecnologia WPF permitem o desenvolvimento de interfaces gráficas muito mais complexas do que é possível em VBS, foi possível unificar scripts que anteriormente se diferenciavam apenas por algumas configurações ou parâmetros e disponibilizar esses itens em uma janela, deixando o próprio projetista manipular esses valores da maneira que lhe parecesse mais adequada. Um exemplo é o do script "folha de bornes", que insere em uma nova folha a vista topográfica de todos os pares dispositivo-pino conectados aos bornes de todas as réguas de bornes. O projetista pode escolher entre os dois símbolos diferentes existentes para as vistas topográficas, além de definir a distância que deve haver entre as réguas quando os símbolos forem inseridos no projeto, além da porcentagem e a quantidade mínima de bornes reserva. Antes da refatoração das automatizações existiam dois scripts diferentes para compreender os dois diferentes símbolos, com todo o código subjacente sendo idêntico nos dois arquivos, a não ser pelo nome dos símbolos e distância padrão entre eles. Dessa forma, unificar essas soluções em uma única automatização ajudou tanto a simplificar a interface dos projetistas quanto o código da solução para os desenvolvedores.

A Figura 46 exibe a interface do script de integração com o estoque, onde são apresentadas informações como o status do material, código, descrição, preço unitário, quantidade reutilizada, quantidade presente no projeto, quantidade presente no estoque e a planta a qual o material pertence.

Figura 46 – Interface de integração com o estoque.

Status	Código	Descrição	Preço	Qtd. Reutilizada	Qtd. Projeto	Qtd. Estoque	Planta
	16086922	BORNE OLHAL RBO 8-HC (3247973)	R$ 59,68	0	6	4	1504
	11620376	BORNE OLHAL ST25 - 100A	R$ 19,70	0	14	16	1505
	10050701	CABO UNIPOLAR PVC AMARELO 750V 70°C 2,5mm²	R$ 1,48	0	60	194,8	1504
	10012735	CABO UNIPOLAR PVC AZUL 750V 70°C 2,5mm²	R$ 1,81	0	240	600	1504
	10012553	CABO UNIPOLAR PVC BRANCO 750V 70°C 2,5mm²	R$ 1,84	0	220	279,8	1504
	10510716	CABO UNIPOLAR PVC VERDE 750V 70°C 2,5mm²	R$ 1,41	0	60	498	1500
	10510716	CABO UNIPOLAR PVC VERDE 750V 70°C 2,5mm²	R$ 0,76	0	60	492	1505
	10731888	CABO UP FLEX 7,94 PVC 1X16mm² 70°C AZ CL	R$ 9,49	0	1	51	1500
	10731888	CABO UP FLEX 7,94 PVC 1X16mm² 70°C AZ CL	R$ 9,09	0	1	95	1202
	10731828	CABO UP FLEX PVC 1X16mm² 70°C BR 0,75kV	R$ 9,45	0	1	800	1504
	10731828	CABO UP FLEX PVC 1X16mm² 70°C BR 0,75kV	R$ 8,76	0	1	679	1500
	10731889	CABO UP FLEX PVC 1X16mm² 70°C VM 0,75kV	R$ 9,45	0	1	82	1504
	16082261	CANALETA PLASTICA 50X126mm CINZA	R$ 19,69	0	0,18	8,94	1504
	10153839	CHAVE SELETORA ST45/10E 600V 10A	R$ 153,37	0	1	5	1202

Fonte: O Autor (2023).

Ao clicar sobre algum dos materiais, a janela é atualizada para exibir todas as suas informações relevantes, bem como permitir a entrada de dados para a quantidade que se deseja reutilizar do estoque e o setor para o qual o material será reservado.

A Figura 48 apresenta a validação da quantidade informada pelo usuário, se tratando de um valor inválido por ser superior a quantidade presente em estoque.

Outra funcionalidade desenvolvida foi a busca de materiais alternativos para suprir a falta de um material em estoque. A Figura 49 exibe a lista de materiais sugeridos pelo script para substituir o borne original utilizado no projeto.

Por fim, a Figura 50 mostra a mudança de status dos materiais selecionados para reutilização. Todos ficam em situação "Aguardando" até que o projetista confirme a reutilização e os valores sejam atualizados no banco de dados de estoque.

Esse tipo de desenvolvimento seria impossível com as antigas tecnologias utilizadas, e por si só já justificariam a refatoração dos scripts para uma linguagem mais moderna, além de permitir o desenvolvimento de automatizações cada vez mais complexas.

3.3.7 Design Patterns aplicados às automatizações

3.3.7.1 Classes Controller

A classe E3Services, que serve de ponto de acesso para todos os serviços que podem ser utilizados pelos scripts, bem como todas as classes Controller que implementam esses serviços foram construídos utilizando o padrão Singleton.

Figura 47 – Interface para um material selecionado.

Fonte: O Autor (2023).

A classe E3Services possui uma instância privada do seu próprio tipo, e um construtor também privado. A única forma de obter um objeto da classe para ter acesso aos seus métodos é através do método público GetInstance. Esse método verifica se a propriedade privada instance é nula, e nesse caso chama o construtor da classe que atribui a ela um novo objeto do tipo E3Services, senão simplesmente retorna o valor da instância. Dessa forma pode-se garantir que sempre que o método GetInstance é invocado o mesmo objeto é retornado. Além de atribuir uma instância à propriedade privada, o construtor da classe E3Services também inicializa todas as suas propriedades Controller, ou seja, E3Controller, que possui os métodos de comunicação e manipulação do software E3.Series, DAOController, que faz a comunicação com os bancos de dados relevantes à aplicação, SharepointController, que implementa os métodos de acesso e manipulação de páginas do Sharepoint, garantindo que todas essas propriedades também sejam inicializadas apenas uma vez. A partir daí, sempre que um método de alguma das classes Controller é chamado, pode-se ter certeza que o objeto que fornece o acesso a esses métodos é o mesmo do início

Figura 48 – Interface para uma quantidade inválida.

Fonte: O Autor (2023).

da aplicação.

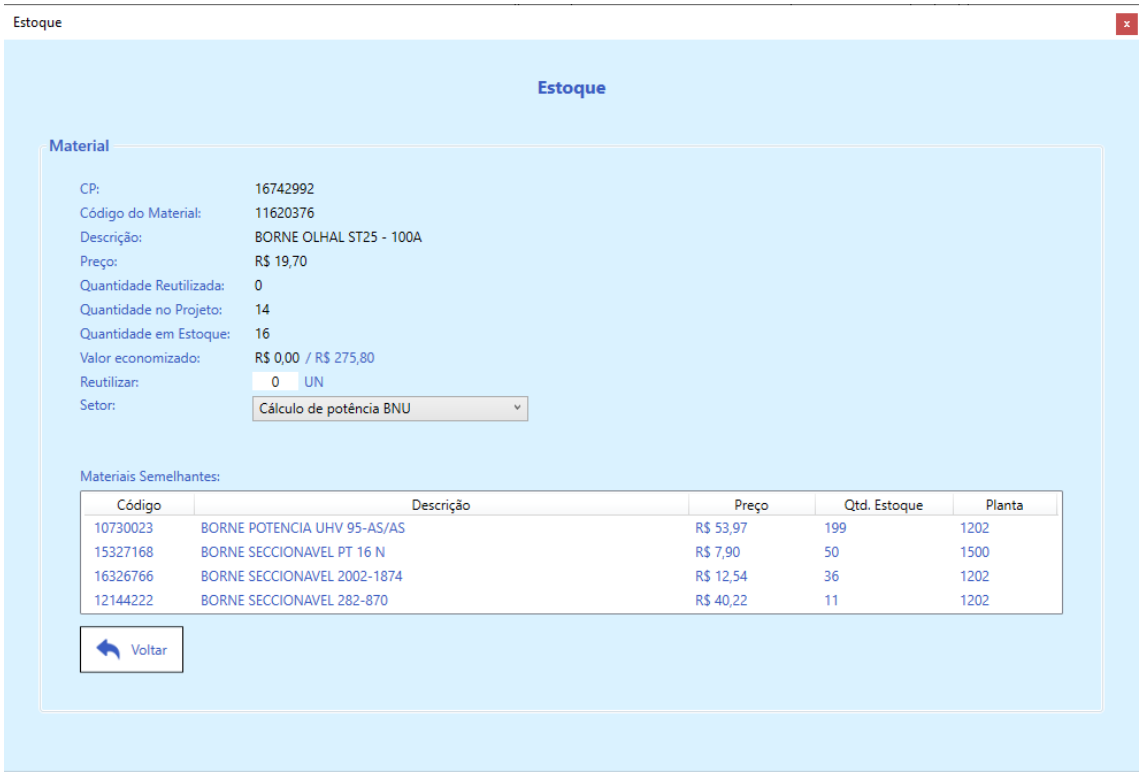
A importância de se utilizar o padrão Singleton nos casos citados vem principalmente da necessidade de garantir a persistência de informações importantes ao longo da execução do script, evitar a criação de múltiplas conexões diferentes com o mesmo serviço e garantir que não irão sobrar conexões abertas quando a automatização for finalizada.

Um dos problemas causados pelos scripts em VBS quando aconteciam exceções durante sua execução era o bloqueio do E3. Como a automatização era encerrada abruptamente, as instruções finais responsáveis por liberar os objetos de conexão com o software nunca eram alcançadas. Dessa maneira a conexão continuava aberta e impedia a correta utilização do E3, já que o programa entendia que ainda havia um script sendo executado.

Da mesma forma, manter conexões abertas com bancos de dados e com páginas do Sharepoint podem causar problemas semelhantes, afetando a performance das aplicações e algumas vezes até impedindo que outros usuários conseguissem se conectar a esses serviços.

Ao utilizar o padrão Singleton e tratamento de exceções nos scripts é possível garan-

Figura 49 – Interface para a seleção de material alternativo.



Fonte: O Autor (2023).

tir que ao final da execução, independente da ocorrência de erros, os métodos responsáveis por fechar as conexões e limpar os objetos de acesso aos serviços serão executados.

3.3.7.2 Model Handlers

Apesar de ser um software focado em objetos, a biblioteca do E3 fornece métodos para a manipulação de seus elementos que funcionam de maneira um pouco diferente do que seria esperado. Para obter todos os dispositivos contidos em um projeto, por exemplo, é possível chamar o método GetDeviceIds do objeto COM que implementa a interface IJobInterface, ou seja, do objeto para manipulação de projetos no E3. Como o nome indica, esse método não retorna objetos do tipo dispositivo, mas sim uma lista de IDs. Então, para acessar os atributos que cada dispositivo é preciso definir o ID de um objeto COM que implementa a interface IDeviceInterface e utilizar seus métodos para a obtenção dos atributos.

O código abaixo exibe um exemplo comum para obter o nome de dois objetos do projeto.

```
1 class Example
2 {
3     static void Main(string[] args)
```

Figura 50 – Interface para a confirmação de reutilização.

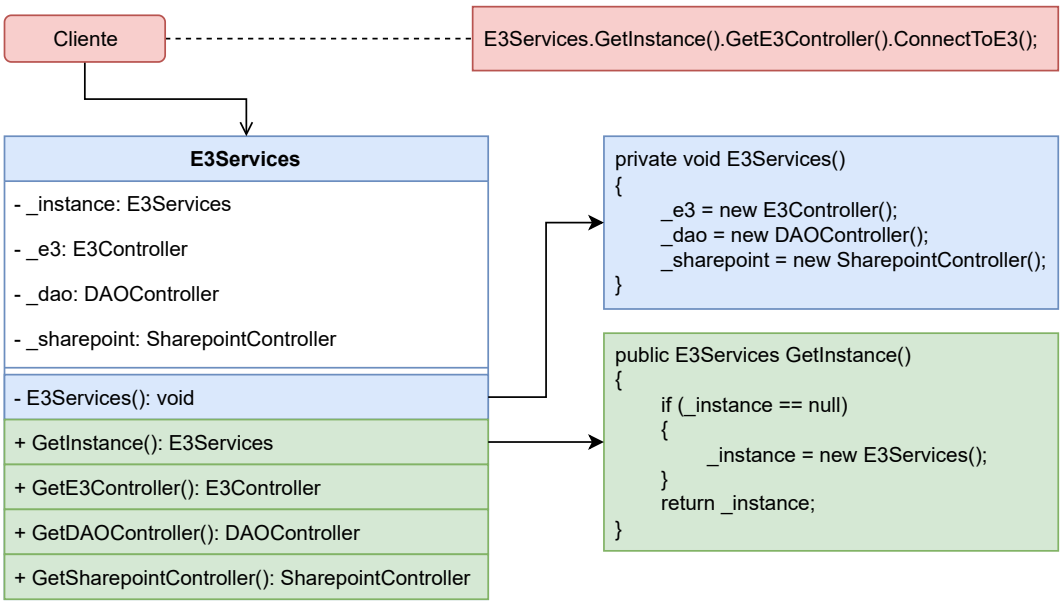
Status	Código	Descrição	Preço	Qtd. Reutilizada	Qtd. Projeto	Qtd. Estoque	Planta
	16086922	BORNE OLHAL RBO 8-HC (3247973)	R$ 59,68	4	6	4	1504
	11620376	BORNE OLHAL ST25 - 100A	R$ 19,70	14	14	16	1505
	10050701	CABO UNIPOLAR PVC AMARELO 750V 70°C 2,5mm²	R$ 1,48	0	60	194,8	1504
	10012735	CABO UNIPOLAR PVC AZUL 750V 70°C 2,5mm²	R$ 1,81	0	240	600	1504
	10012553	CABO UNIPOLAR PVC BRANCO 750V 70°C 2,5mm²	R$ 1,84	0	220	279,8	1504
	10510716	CABO UNIPOLAR PVC VERDE 750V 70°C 2,5mm²	R$ 1,41	30	60	498	1500
	10510716	CABO UNIPOLAR PVC VERDE 750V 70°C 2,5mm²	R$ 0,76	30	60	492	1505
	10731888	CABO UP FLEX 7,94 PVC 1X16mm² 70°C AZ CL	R$ 9,49	0	1	51	1500
	10731888	CABO UP FLEX 7,94 PVC 1X16mm² 70°C AZ CL	R$ 9,09	0	1	95	1202
	10731828	CABO UP FLEX PVC 1X16mm² 70°C BR 0,75kV	R$ 9,45	0	1	800	1504
	10731828	CABO UP FLEX PVC 1X16mm² 70°C BR 0,75kV	R$ 8,76	0	1	679	1500
	10731889	CABO UP FLEX PVC 1X16mm² 70°C VM 0,75kV	R$ 9,45	0	1	82	1504
	16082261	CANALETA PLASTICA 50X126mm CINZA	R$ 19,69	0,1	0,18	8,94	1504
	10153839	CHAVE SELETORA ST45/10E 600V 10A	R$ 153,37	0	1	5	1202

Fonte: O Autor (2023).

```
4      {
5          e3Application app = new e3Application();
6          e3Job job = app.CreateJobObject();
7          e3Device dev = job.CreateDeviceObject();
8
9          object devIds = null;
10         job.GetDeviceIds(ref devIds);
11
12         object[] devIdsArray = devIds as object[];
13         // devIdsArray: {22151, 23512, ...}
14
15         dev.SetId(devIdsArray[0]);
16         string deviceName1 = dev.GetName(); // deviceName1: "M1"
17
18         dev.SetId(devIdsArray[1]);
19         string deviceName2 = dev.GetName(); // deviceName2: "DAP"
20     }
21 }
```

O objeto *app* representa a aplicação E3. O objeto *job* representa um projeto do E3, contendo métodos para a obtenção e manipulação de seus atributos. É através do objeto *job* que é invocado o método para a obtenção dos IDs de todos os dispositivos do projeto. Então, para acessar os atributos de cada um dos objetos deve-se definir o ID do objeto

Figura 51 – Utilização do padrão Singleton para as classes Controller e E3Services.



Fonte: O Autor (2023).

device igual aos IDs da lista, um por vez, e chamar os métodos para obter informações sobre aquele objeto específico.

Visando tornar a manipulação de elementos do E3 um pouco mais intuitiva e de acordo com uma aplicação orientada a objetos, foram criadas classes de modelo para representar esses elementos.

Para representar os dispositivos do projeto foi criada a classe E3Device. Essa classe possui diversos atributos que guardam os dados retornados pelos métodos do objeto COM do tipo e3Device, como ID, nome, modelo, descrição, nome do componente, lista de atributos, entre outros.

Abaixo é exibido um código de exemplo para a obtenção dos dispositivos de um projeto utilizando uma classe E3Device que possui apenas os atributos ID e Name.

```
1 class E3Device
2 {
3     public int ID { get; set; }
4     public string Name { get; set; }
5 }
6
7 class E3Controller
8 {
9     private static e3Application _app;
10    private static e3Job _job;
11    private static e3Device _dev;
12
13    public void ConnectToE3()
```

```
14      {
15          _app = new e3Application();
16          if (_app == null) { throw new E3ApplicationException(); }
17
18          _job = _app.CreateJobObject();
19          if (_job == null || _job.GetId() <= 0) { throw new
                E3ProjectException(); }
20      }
21
22      public int[] GetDeviceIds()
23      {
24          object devIds = null;
25          _job.GetDeviceIds(ref devIds);
26          object[] devIdsArray = devIds as object[];
27          int[] ids = new int[devIdsArray.Length];
28          for (int i = 0; i < devIdsArray.Length; i++)
29          {
30              ids[i] = (int) devIdsArray[i];
31          }
32          return ids;
33      }
34
35      public void InitializeDevice()
36      {
37          if (_dev == null)
38          {
39              _dev = _job.CreateDeviceObject();
40          }
41      }
42
43      public E3Device[] GetDevices()
44      {
45          InitializeDevice();
46          int[] devIds = GetDeviceIds();
47          E3Device[] devices = new E3Devices[devIds.Length];
48          for (int i = 0; i < devIds.Length; i++)
49          {
50              _dev.SetId(devIds[i]);
51              devices[i] = new E3Device()
52              {
53                  ID = _dev.GetId(),
54                  Name = _dev.GetName()
55              };
56          }
57          return devices;
58      }
59  }
```

```
60
61 class Client
62 {
63     public static void Main(string[] args)
64     {
65         E3Controller e3 = E3Services.GetInstance().GetE3Controller();
66         e3.ConnectToE3();
67         E3Devices[] devices = e3.GetDevices();
68         // devices[0]: { ID: 22151, Name: "M1" }
69         // devices[1]: { ID: 23512, Name: "DAP" }
70     }
71 }
```

Após essa reestruturação foi possível verificar melhorias significativas, não sendo mais necessário iterar sobre cada um dos IDs para verificar qual o objeto representado por ele.

Porém, classes como essa ainda apresentam uma dificuldade, que é a ausência de métodos. Da forma como foram construídas funcionam apenas como objetos para o armazenamento de dados, sem definir nenhum comportamento. Todas as manipulações foram definidas na classe E3Controller, como mostra o exemplo a seguir:

```
1 class E3Controller
2 {
3     public void SetName(E3Device device, string newName)
4     {
5         InitializeDevice();
6         _dev.SetId(device.ID);
7         _dev.SetName(newName);
8         device.Name = _dev.GetName();
9     }
10 }
11
12 class Client
13 {
14     public static void Main(string[] args)
15     {
16         E3Controller e3 = E3Services.GetInstance().GetE3Controller();
17         e3.ConnectToE3();
18         E3Devices[] devices = e3.GetDevices();
19         E3Device device = devices[0];
20         // device: { ID: 22151, Name: "M1" }
21         e3.SetName(device, "M2");
22         // device: { ID: 22151, Name: "M2" }
23     }
24 }
```

Logo que a classe E3Controller passou a abrigar muitos métodos devido à evolução do projeto, notou-se uma dificuldade em encontrar quais funções já haviam sido desenvolvidas e para quais modelos elas poderiam ser aplicadas. Como resultado pensou-se em trazer o comportamento relativo a cada modelo para dentro da classe, de forma que pudesse ser acessado pelo próprio objeto e se evitasse a necessidade de buscar entre todos os métodos da classe E3Controller.

Uma das alternativas que parecia melhor se adaptar à necessidade era através da implementação do padrão de projeto estrutural Decorator. Através desse padrão foi possível agregar comportamentos aos modelos existentes através da estratégia de composição, ou seja, guardando uma referência do objeto modelo e delegando seu comportamento para métodos de outra classe.

Nesse caso, o comportamento dos objetos modelo foram delegados à classe E3Controller, de maneira que a chamada de um método do modelo tinha seu resultado refletido automaticamente no software E3.

A implementação dessa estratégia se deu da seguinte forma: foram criadas classes "Handler" que possuem uma referência para um objeto do tipo Model e que possuem acesso aos métodos fornecidos pela classe E3Controller.

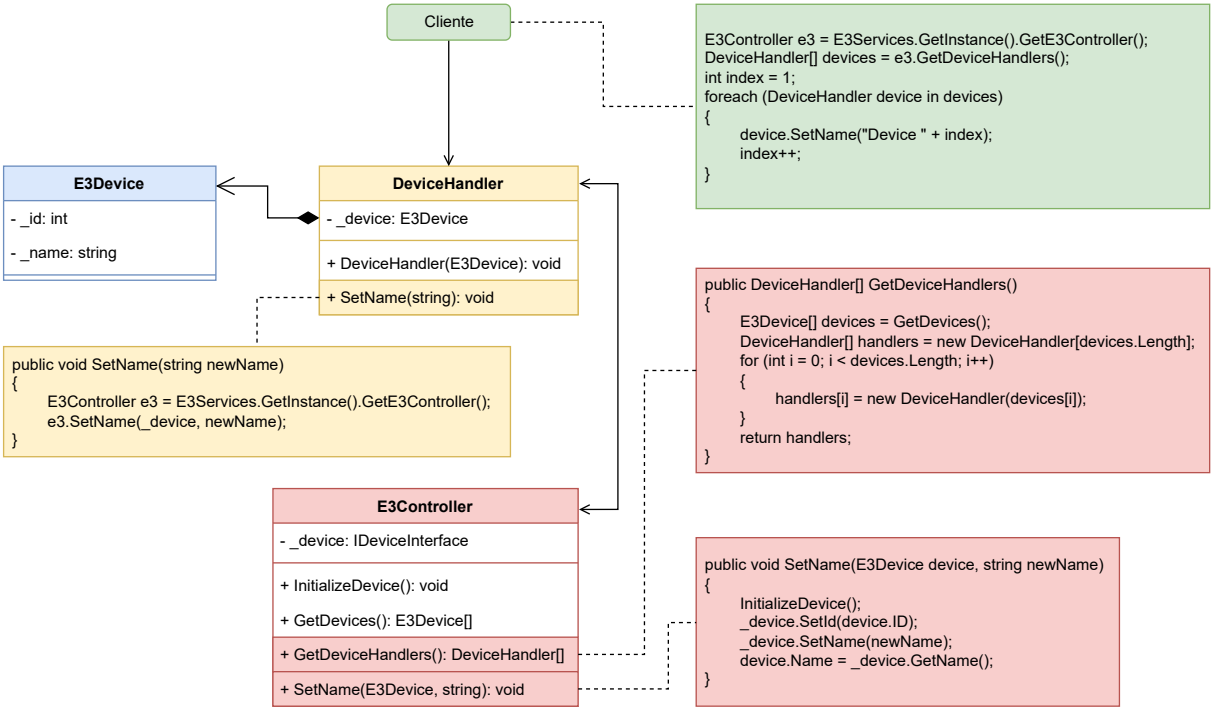
Um código de exemplo é exibido abaixo:

```
1  class E3Device
2  {
3      public int ID { get; set; }
4      public string Name { get; set; }
5  }
6
7  class DeviceHandler()
8  {
9      private E3Device _device;
10
11     public DeviceHandler(E3Device device)
12     {
13         _device = device;
14     }
15
16     public void SetName(string newName)
17     {
18         E3Controller e3 = E3Services.GetInstance().GetE3Controller();
19         e3.SetName(_device, newName);
20     }
21 }
22
23 class E3Controller
24 {
25     public DeviceHandler[] GetDeviceHandlers()
```

```
26      {
27          E3Device[] devices = GetDevices();
28          DeviceHandler[] handlers = new DeviceHandler[devices.Length];
29          for (int i = 0; i < devices.Length; i++)
30          {
31              handlers[i] = new DeviceHandler(devices[i]);
32          }
33          return handlers;
34      }
35  }
36
37  class Client
38  {
39      public static void Main(string[] args)
40      {
41          E3Controller e3 = E3Services.GetInstance().GetE3Controller();
42          DeviceHandler[] devices = e3.GetDeviceHandlers();
43          DeviceHandler device = devices[0];
44          // device: { ID: 22151, Name: "M1" }
45          device.SetName("M2");
46          // device: { ID: 22151, Name: "M2" }
47      }
48  }
```

Ao utilizar objetos do tipo “Handler” no lugar das classes de dados, é possível utilizar os métodos de manipulação de elementos do E3 diretamente dentro da classe com a que se está trabalhando, com os resultados dessas manipulações sendo refletidos automaticamente no software.

Figura 52 – Diagrama de utilização do padrão Decorator para adição de comportamentos aos modelos.



Fonte: O Autor (2023).

3.3.7.3 Geração automática de circuitos de ventilação forçada

O projeto de circuitos de ventilação forçada depende de diversos parâmetros e pode ser construído de diferentes formas de acordo com o tipo do projeto para o qual está sendo desenvolvido, mas ainda assim segue uma sequência lógica de construção que nunca muda. Por esse motivo o padrão Template Method foi utilizado para implementar essa automatização.

A classe abstrata ForcedVentilationTemplate define os métodos utilizados para a construção de cada parte do circuito de ventilação forçada, e um método template principal que invoca cada um dos métodos de construção na sequência correta. É o método template principal que define a sequência de passos que formam o esqueleto da lógica de construção do circuito.

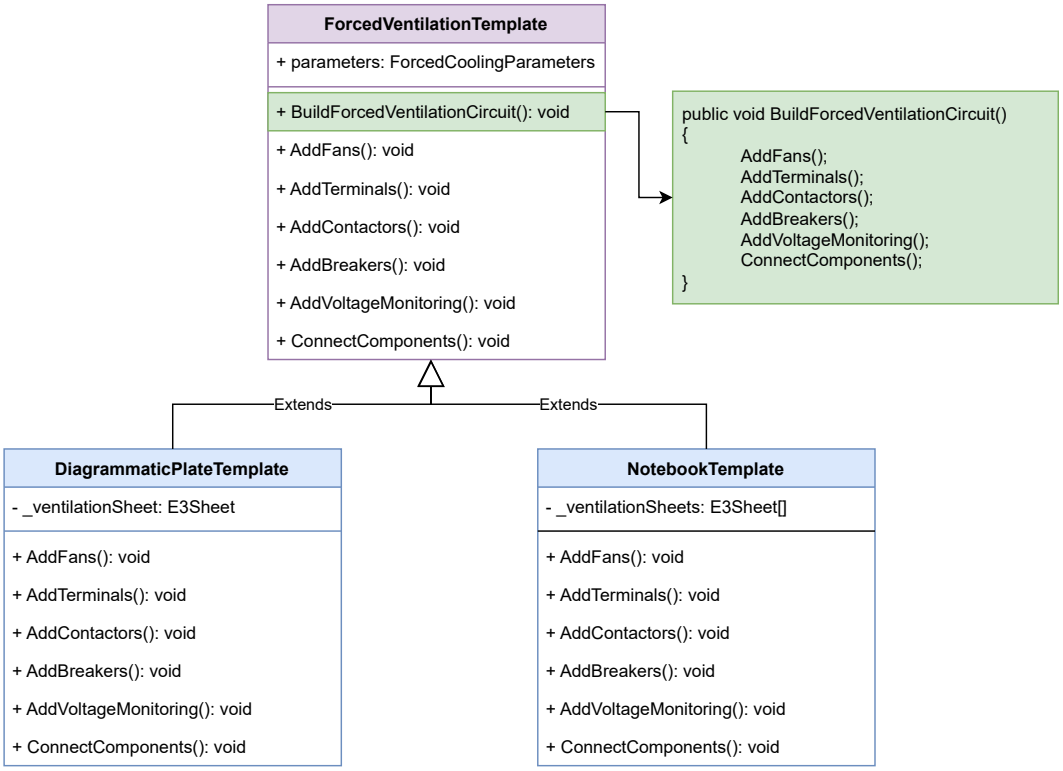
A classe define os métodos AddFans, AddTerminals, AddContactors, AddBreakers, AddVoltageMonitoring e ConnectComponents, além do método BuildForcedVentilationCircuit, que chama cada um dos métodos anteriores em sequência, sendo exatamente essa a lógica de construção do circuito, ou seja, adição dos ventiladores, bornes, contatores, disjuntores, monitor de tensão e conexão dos componentes.

Porém, na classe abstrata os métodos são apenas definidos. São nas classes concretas que estendem a classe abstrata que os métodos são realmente implementados de acordo

com suas especificidades.

Os projetos no E3 costumam seguir dois tipos principais: placa diagramática e caderno. Nos projetos do tipo placa diagramática, todo o projeto dos circuitos de comando e controle são inseridos em uma única placa, contendo os circuitos de força, ventilação, iluminação, controle, entre os outros circuitos que compõem um projeto completo, enquanto os projetos do tipo caderno possuem cada um desses circuitos dispostos em folhas separadas, específicas para aquele circuito. Por essa razão, durante a construção de um circuito de ventilação forçada é preciso se atentar para o tipo do projeto e construir cada uma das partes de acordo com sua especificação. Por essa razão foram criadas duas classes concretas para a construção dos circuitos, DiagrammaticPlateTemplate e NotebookTemplate, em que cada uma implementa os métodos da classe abstrata ForcedVentilationTemplate com algumas diferenças entre si para reproduzir o comportamento esperado para cada tipo de projeto.

Figura 53 – Utilização do padrão Template Method para a construção de circuitos de ventilação forçada.



Fonte: O Autor (2023).

3.3.7.4 Filtro de componentes para a geração automática de circuitos

Um ponto bastante importante na geração automática de circuitos é a correta seleção dos componentes que irão compor o circuito. Após a definição dos parâmetros

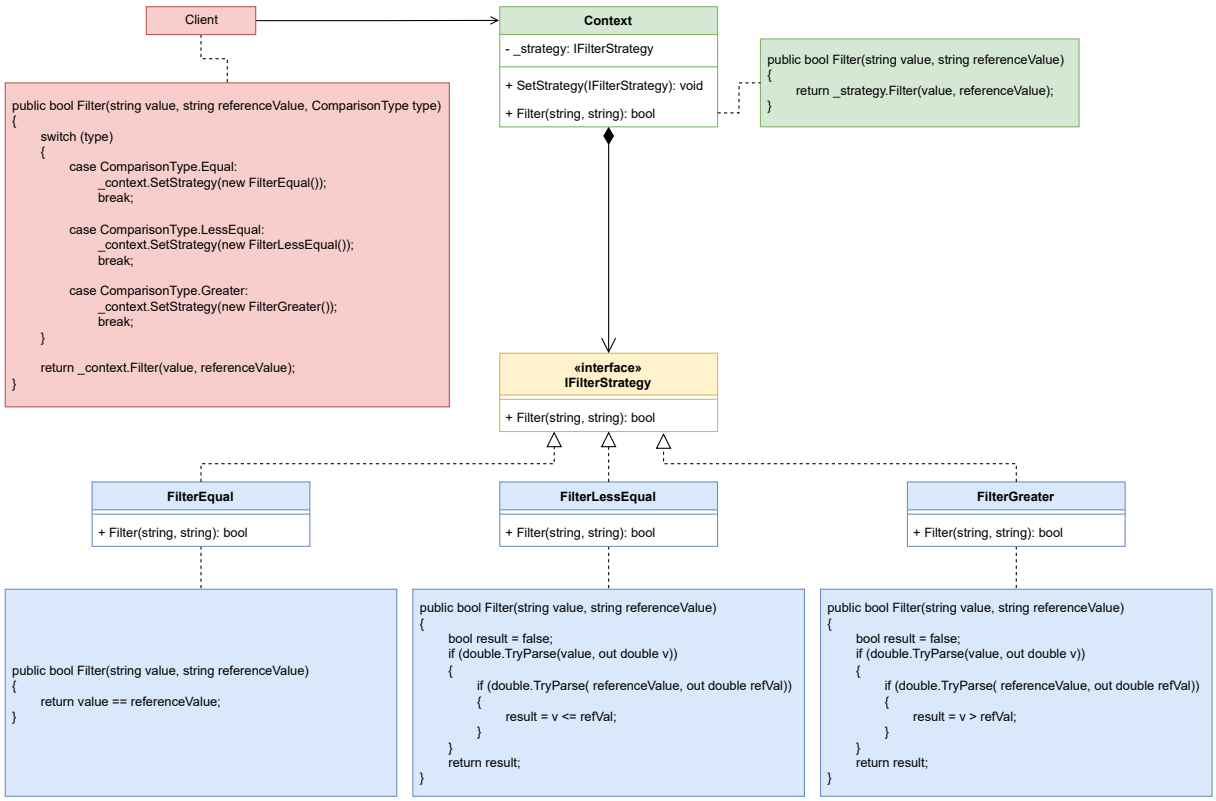
por parte do projetista, o script deve ser capaz de encontrar os componentes adequados, utilizando para isso diferentes tipos de comparação de dados.

Foi implementada uma classe para estender o comportamento das listas de componentes contendo um método "Filter", que recebe por parâmetro o nome da característica que deve ser utilizada para filtrar os componentes, o valor de referência e o tipo de comparação, que pode ser: igual, menor, menor-igual, maior, maior-igual, ou contém.

Como o software permite a criação de novas características que são salvas no banco de dados junto ao seu componente, todos os valores são armazenados no formato de texto. Dessa forma, comparações que só fazem sentido entre valores numéricos, como menor, maior, etc, devem primeiro converter os valores de texto para valores numéricos para depois realizar a comparação. Já operações como igual ou contém não precisam se preocupar com conversões, podendo ser aplicadas diretamente.

Visando tornar o código da função "Filter"menos poluído, foi utilizado o padrão de projeto comportamental Strategy. Esse padrão define uma interface que declara um método que deve ser implementado por suas classes concretas, onde cada classe implementa o algoritmo de maneira diferente.

Figura 54 – Utilização do padrão Strategy para o filtro de componentes.



Fonte: O Autor (2023).

3.3.7.5 Desenvolvimento de interfaces com WPF

A utilização da arquitetura MVVM para o desenvolvimento de interfaces usando WPF praticamente exige a utilização do padrão de projeto comportamental Command, já que a própria estrutura MVVM funciona através da ligação de elementos da interface com propriedades das classes de controle, cuja ligação é feita via objetos Command que carregam todos os atributos da requisição e disparam a execução das funções necessárias.

Elementos de interface em WPF como os botões inclusive possuem um atributo chamando Command, que deve ser inicializado com uma propriedade da interface ICommand declarada na ViewModel. Essa propriedade recebe como parâmetro uma função que deve ser executada sempre que o botão for clicado.

A conexão entre a propriedade da ViewModel é feita através do mecanismo de binding fornecido pela ferramenta WPF.

```
1 // Botao WPF
2 <Window>
3     <StackPannel>
4         <!-- Botao com o parametro Command em binding com a
              propriedade ExecuteFunctionCommand da ViewModel -->
5         <Button Command={Binding ExecuteFunctionCommand}/
6     </StackPannel>
7 </Window>
```

```
1 // Classe concreta que implementa Icommand
2 public class DelegateCommand : ICommand
3 {
4     // Atributo do tipo Action que recebe a funcao que deve ser
          executada
5     private Action<object> _execute;
6
7     // Construtor da classe recebendo a funcao por parametro
8     public DelegateCommand(Action<object> execute)
9     {
10        _execute = execute;
11    }
12
13    // Implementacao do metodo Execute da interface ICommand que
          invoca a execucao da funcao
14    public void Execute(object parameter)
15    {
16        _execute(parameter);
17    }
18 }
```

```
1 // Classe que representa a camada ViewModel
2 public class ViewModel
```

```
3 {
4     // Propriedade em binding com o atributo Command do botao da
          interface
5     public ICommand ExecuteFunctionCommand { get; set; }
6
7     // Construtor da ViewModel
8     public ViewModel()
9     {
10        // Propriedade sendo inicializada com um novo comando que
              invoca a funcao MyFunction
11        ExecuteFunctionCommand = new DelegateCommand(MyFunction);
12    }
13
14    // Funcao executada quando o botao e clicado
15    public void MyFunction(object obj)
16    {
17        // logica de negocio
18    }
19 }
```

Ao observar os fragmentos de código em WPF, a classe DelegateCommand e a classe ViewModel, pode-se perceber que o clique do botão não dispara diretamente a função MyFunction, mas sim que a requisição da execução é feita através de um objeto command que interliga o elemento gráfico às regras de negócio da aplicação.

4 RESULTADOS

Após oito meses da liberação do último script refatorado, as automatizações já foram executadas mais de 50 mil vezes.

Figura 55 – Dados de execução dos scripts.

DADOS DOS SCRIPTS - 07/03/2023 09:40:44				
TOTAL				
Scripts	Execuções	Tempo total (s)	Tempo médio (s)	Exceções
Check-in before	13107	62796	4,7	403
Check-in after	12770	30154	2,4	8
Anilhas	3837	38511	10,2	13
Verifica projeto	3718	92670	24,4	94
SAP BOM Upload	3555	23176	6,6	18
Lista de materiais	3393	35855	10,5	32
6 - Gera listagem final	1298	42393	32,8	20
Atualiza Workspace	1274	7568	5,9	20
3 - Gera planilha mecânico	1016	12363	11,9	34
5 - Cria canaleta	965	9700	10,2	3
4 - Lê planilha mecânico	844	44178	51,6	14
1 - Configura roteamento	716	11	0	2
Folha de bornes	650	6032	9,1	29
Recarrega template	467	1407	3,1	4
Compara BOM	370	339	0,9	0
Índice	356	211	0,6	1
7 - Corrige cabos tabela	254	293	1,2	0
	253	1806	7,3	0
2 - Cria montagens	242	126	0,5	2
Atualiza textos	90	171	1,9	0
Atualiza projeto parcial	73	572	8,2	0
Atualiza projeto completo	42	3780	90	1
Legenda	42	38	0,9	0
Atualiza projeto parcial	33	614	19,2	0
Atualiza projeto completo	13	1361	104,7	0
Índice A4	13	5	0,4	0
Topográfico bornes	7	14	2,1	0
Documentos do projeto	3	0	0	0

Fonte: O Autor (2023).

Ao final das refatorações, alguns scripts tiveram seu tempo de execução reduzido, enquanto outros scripts ficaram mais lentos. No segundo caso, o maior tempo de execução decorreu principalmente de dois fatores: correção de bugs que faziam os scripts antigos finalizarem mais rápido, mas com resultados incorretos, e adição de novas funcionalidades às automatizações.

O aumento de tempo de alguns scripts não se tornou um problema pois as novas

funcionalidades agregaram valor qualitativo às automatizações que compensam alguns segundos a mais de execução.

Boa parte das novas funcionalidades dizem respeito às melhores mensagens de erro, que permitem que os projetistas identifiquem os problemas rapidamente e sem a necessidade de abertura de chamados de suporte, e a novas funcionalidades que antes precisariam ser executadas manualmente.

Mesmo que algumas automatizações tenham se tornado mais lentas, ao analisar todo o conjunto dos scripts utilizados e quantidade de vezes que eles são executados, o tempo total de execução ainda foi reduzido. Para chegar nessa conclusão foram utilizados os arquivos de log gerados ao longo de dois meses, permitindo reunir dados como quantidade de execuções e tempo total de execução de cada script. Foram também calculados os tempos de execução dos antigos scripts, de forma que se chegou em uma razão entre o tempo de execução que seria demandado pelos scripts antigos e pelos scripts refatorados. Estendendo o cálculo para compreender um período de 12 meses, chegou-se em 20,6 horas de redução em tempo de execução.

Para o cálculo de economia de tempo, os scripts que possuíam um tempo de execução não desprezível foram comparados entre suas versões em VBS e em C# permitindo encontrar a razão de tempo entre ambos, como demonstra a tabela 1.

Tabela 1 – Tabela de tempo de execução dos scripts em VBS e C#.

Scripts	VBS (s)	C# (s)	Razão
Anilhas	5,5	3	0,5455
Atualiza texto	7	0,5	0,0714
Atualiza projeto - Completo	62	69	1,1129
Atualiza projeto - Parcial	3	3	1
Folha de bornes	1	4	4
Lista de materiais	6	2	0,3333
Verifica projeto	8	5	0,625
Gera planilha mecânico	5,5	8	1,4545
Lê planilha mecânico	2,5	9	3,6
Gera listagem final	4,5	9	2

Metade dos scripts analisados tiveram seu tempo de execução aumentado, mas para calcular o impacto líquido dessas mudanças os tempos foram multiplicados proporcionalmente à sua quantidade de execuções. Dessa forma, um script um pouco mais rápido utilizado com muita frequência ainda gera uma economia de tempo total quando comparado a um script mais lento executado poucas vezes, e foi exatamente esse o resultado obtido.

Figura 56 – Economia de tempo dos scripts em C#.

DADOS DOS SCRIPTS - 11/08/2022 14:50:33				
TOTAL				
Scripts	Execuções	Tempo total (s)	Razão	Economia (s)
Anilhas	855	3559	0,5455	1483
Verifica projeto	761	13653	0,6250	4096
Lista de materiais	705	4631	0,3333	4632
Lê planilha mecânico	223	7905	3,6000	-2855
Gera listagem final	227	3008	2,0000	-752
Gera planilha mecânico	163	1395	1,4545	-218
Folha de bornes	147	776	4,0000	-291
Atualiza projeto completo	29	2406	1,1129	-122
Atualiza projeto parcial	27	197	1,0000	0
Atualiza textos	14	31	0,0714	202

Fonte: O Autor (2023).

Os tempos totais exibidos na Figura 56 são dos scripts já em C# executados ao longo de dois meses. Para calcular o tempo economizado foi utilizado o seguinte equacionamento:

$$r = \frac{t}{T} \tag{1}$$

$$\Delta T = T - t \tag{2}$$

$$\Delta T = \frac{t}{r} - t = \frac{t - rt}{r} = \frac{t}{r}(1 - r) \tag{3}$$

onde ΔT é a economia de tempo, t o tempo total em C#, T o tempo total em VBS e r a razão do tempo em C# pelo tempo em VBS.

A economia de tempo exibida na última coluna da tabela da Figura 56 é resultado da aplicação da equação (3) aos valores da tabela.

A soma de dos valores de economia de tempo resultam em aproximadamente 103 minutos de economia por mês, ou quase 21 horas por ano.

Esse valor corresponde aos ganhos que eram possíveis de serem calculados quantitativamente, mas se fosse possível analisar também os ganhos qualitativos com redução de tempo de suporte, maior autonomia dos projetistas para correção de erros de projeto e a utilização de novos desenvolvimentos que até então eram realizados manualmente, esses ganhos seriam ainda maiores.

Com o novo processo de liberação implementado, também foi possível inserir novas automatizações à interface do E3 para todos os projetistas sem que fosse necessária nenhuma intervenção por parte do suporte, bem como a atualização automática de arquivos de configuração.

5 CONCLUSÃO

Ao final do desenvolvimento desse trabalho de conclusão de curso foi possível entregar uma solução coesa para a automatização dos processos de projetos de circuitos de comando e controle.

A solução foi implementada utilizando boas práticas de desenvolvimento, como a estruturação do projeto em diferentes camadas com responsabilidades únicas, orientação a objetos e suas principais características, e documentação do código e suas funcionalidades.

Além disso, foi possível estudar os padrões de projeto criacionais, comportamentais e estruturais, entender seus casos de uso, buscar oportunidades de utilização e aplicá-los de acordo com as necessidades dos scripts e da solução como um todo, contribuindo para a construção de uma solução mais organizada e flexível.

A refatoração dos antigos scripts de VBS para C# proporcionou ainda outras melhorias tanto para os usuários quanto para o time de suporte e desenvolvimento.

Do ponto de vista dos usuários houve melhorias nas interfaces gráficas para a definição de parâmetros dos scripts, tempos de execução mais rápidos, melhor tratamento de exceções, implementação de mensagens de alerta mais claras que ajudam os projetistas a identificar os erros mais rapidamente e corrigi-los sem a necessidade de auxílio e expansão das automatizações para abranger casos até então não contemplados.

Para o time de suporte, as melhorias tornaram todo o conjunto de automatizações mais coeso, permitiram o rastreamento de dados de utilização dos scripts, como quantidade de execuções, tempo total despendido, quantidade de exceções e suas mensagens de erro, recebimento automático de e-mails de alerta de exceção, documentação tanto do código quanto da funcionalidade das automatizações e a utilização de uma linguagem de programação com ferramentas mais adequadas para o desenvolvimento e manutenção dos scripts.

Essa reestruturação também abriu caminho para trabalhos futuros, com o planejamento de um novo projeto, visando o atendimento de demandas antigas e de novas necessidades que surgiram ao longo dos últimos meses, que passarão por um processo semelhante de planejamento, desenvolvimento, testes e liberação, mas agora com uma base muito mais bem estruturada para permitir o desenvolvimento de automações mais complexas e completas.

REFERÊNCIAS

ALEXANDER, Christopher *et al.* **A Pattern Language**: Towns, Buildings, Construction. New York: Oxford University Press, 1977. ISBN 9780195019193.

CADU. **Entendendo o Pattern Model View ViewModel MVVM**. [*S.l.*], 2022. Disponível em: https://www.devmedia.com.br/entendendo-o-pattern-model-view-viewmodel-mvvm/18411. Acesso em: 12 nov. 2022.

GAMMA, Erich *et al.* **Design Patterns**: Elements of Reusable Object-Oriented Software. [*S.l.*: *s.n.*], 2009.

GURU, Refactoring. **Abstract Factory**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/abstract-factory. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Adapter**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/adapter. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Bridge**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/bridge. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Builder**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/builder. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Chain of Responsibility**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/chain-of-responsibility. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Command**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/command. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Composite**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/composite. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Decorator**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/decorator. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Façade**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/facade. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Factory Method**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/factory-method. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Flyweight**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/flyweight. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Iterator**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/iterator. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Mediator**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/mediator. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Memento**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/memento. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Observer**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/observer. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Prototype**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/prototype. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Proxy**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/proxy. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Singleton**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/singleton. Acesso em: 19 jun. 2023.

GURU, Refactoring. **State**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/state. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Strategy**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/strategy. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Template Method**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/template-method. Acesso em: 19 jun. 2023.

GURU, Refactoring. **Visitor**. Refactoring Guru. Disponível em: https://refactoring.guru/design-patterns/visitor. Acesso em: 19 jun. 2023.

HOERLLE, Arthur. **Diferenças entre CAD, CAE e Vantagens do E3.series**. [*S.l.*], 2022. Disponível em: https://www.e3seriescenters.com/pt/blog-engenharia-eletrica-moderna/diferen%5C%C3%5C%A7as-entre-cad-e-cae. Acesso em: 13 nov. 2022.

MARKET, Cim-Team Latin. **Software para Projetos Elétricos**. [*S.l.*], 2022. Disponível em: https://www.e3seriescenters.com/pt/software-para-fazer-projetos-eletricos. Acesso em: 12 nov. 2022.

MARKET, Cim-Team Latin. **Visão Geral do E3.series**. [*S.l.*], 2022. Disponível em: https://www.e3seriescenters.com/pt/software-para-desenho-de-cabos-eletricos/e3-series. Acesso em: 12 nov. 2022.

MARTIN, Robert C. **Agile Software Development: Principles, Patterns, and Practices**. Boston: Pearson Education, 2002.

MICROSOFT. **Component Object Model**. [*S.l.*], 2022. Disponível em: https://learn.microsoft.com/pt-br/windows/win32/com/component-object-model--com--portal. Acesso em: 12 nov. 2022.

MICROSOFT. **Data binding and MVVM**. [*S.l.*], 2022. Disponível em: https://learn.microsoft.com/en-us/windows/uwp/data-binding/data-binding-and-mvvm. Acesso em: 12 nov. 2022.

MICROSOFT. **Data Types and Variables**. [*S.l.*], 2022. Disponível em: https://www.oreilly.com/library/view/vbscript-in-a/1565927206/ch03.html. Acesso em: 12 nov. 2022.

MICROSOFT. **Differences Between VBScript and VBA**. [*S.l.*], 2022. Disponível em: https://www.oreilly.com/library/view/vbscript-in-a/1565927206/ch01s03.html. Acesso em: 12 nov. 2022.

MICROSOFT. **Guia da área de trabalho (WPF .NET)**. [*S.l.*], 2022. Disponível em: https://learn.microsoft.com/pt-br/dotnet/desktop/wpf/overview/?view=netdesktop-6.0. Acesso em: 12 nov. 2022.

MICROSOFT. **What Is VBScript?** [*S.l.*], 2022. Disponível em: https://learn.microsoft.com/en-us/previous-versions//1kw29xwf(v=vs.85)?redirectedfrom=MSDN. Acesso em: 12 nov. 2022.

NULAB. **Kanban Guide**. Nulab. Disponível em: https://nulab.com/learn/project-management/kanban-guide/. Acesso em: 15 jun. 2023.

RODRIGUES, Jonatan. **Metodologia Ágil: descubra o que é, os principais tipos + 6 funções que ela cumpre nos projetos de qualquer área**. [*S.l.*], 2020. Disponível em: https://resultadosdigitais.com.br/marketing/metodologia-agil/. Acesso em: 28 nov. 2022.

SYDLE. **Framework Scrum: o que é e como funciona?** Sydle. Disponível em: https://www.sydle.com/br/blog/framework-scrum-5f6dc45f320703787497f887/. Acesso em: 15 jun. 2023.