

Unidade 3

Boas práticas e patterns (padrões)



Introdução da Unidade

Com o crescimento da demanda na área de desenvolvimento de sistemas web, começaram a surgir alguns erros e problemas dentro das aplicações devido à forma de programar sistemas. Foi então que surgiram as boas práticas e os patterns (padrões) de desenvolvimento.

Na aula 01, vamos conhecer como surgiram algumas boas práticas e patterns de desenvolvimento e como eles auxiliam na organização da construção de sistemas web.

Na aula 02, vamos entrar em detalhes no funcionamento de alguns patterns, com exemplos e sugestões de como podem ser aplicados.

Objetivos

- Conhecer as boas práticas do desenvolvimento web.
- Compreender como aplicar na prática alguns patterns de desenvolvimento de sistemas web.

Conteúdo programático

Aula 01 — Boas práticas no desenvolvimento web

Aula 02 — Patterns: como aplicar?

Aula 01 — Boas práticas no desenvolvimento web

Olá, aluno, tudo bem?

Durante muitos anos, foi se desenvolvendo sistemas web de um modo que a aplicação deveria funcionar para atender às necessidades do cliente; muitas vezes a frase mais utilizada pelo desenvolvedor era: “Se está funcionando, então tudo certo, só entregar”, porém isso resultou em alguns problemas.

Um sistema web funcional não necessariamente foi bem desenvolvido: existem diversos sistemas web que se enquadram nessa situação, tem diversas aplicações que funcionam, mas são lentas e de difícil manutenção.

Um cliente não deseja esperar dois meses para que uma correção no sistema que utiliza seja entregue. Visando isso, existem algumas práticas que foram aprendidas e documentadas durante os anos.

Dessa forma se construiu um conjunto de boas práticas para o desenvolvimento web.



Videoaula 1

Utilize o QR Code para assistir!

Assista ao vídeo para entender melhor o assunto.



Alguns dos aspectos apresentados nesta aula sobre boas práticas são baseados em conceitos e dados de Clean Code (Código Limpo). São conceitos e práticas que precisam ser levadas em conta na hora de desenvolver web de

forma limpa e eficaz, melhorando muito a qualidade do código e ocasionalmente a aplicação como um todo.

Nomenclatura

A nomenclatura dos componentes dentro dos códigos ocasionava um dos principais problemas identificados durante o desenvolvimento de sistemas web, responsável pelo alto tempo para realizar uma manutenção em um sistema.

Figura 1 — Método nomenclatura confusa

```
Integer alterDataAggregationTwo(Integer a, Integer b) {  
    return a + b;  
}
```

Fonte: O próprio autor (2022).

A figura 1 demonstra um exemplo de método desenvolvido na linguagem Java, em que se somou dois ao valor inteiro que chega por parâmetro no método. Veja o nome do método: **alterDataAggregationTwo**. Observe que esse nome indica a alteração do dado, agregando dois. Pode ser que isso gere um problema em um sistema grande.

O nome do método, classe, objeto e até mesmo o nome da aplicação web devem estar condizentes com o que ele realiza. Sempre que aplicada essa boa prática, fica mais fácil a manutenção no futuro.

Figura 2 — Método nomenclatura clara

```
Integer sumOfValues(Integer a, Integer b) {  
    return a + b;  
}
```

Fonte: O próprio autor (2022).

A figura 2 apresenta a escrita de um método de modo mais claro. Veja que o método é responsável por somar dois valores, então seu nome se apresenta **sumOfValues**, ou seja, soma de valores.

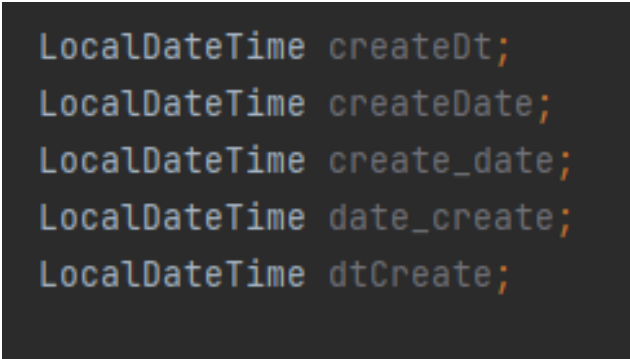
Vale ressaltar que os padrões de escrita também podem ser definidos pela empresa, em sua maioria os projetos são desenvolvidos utilizando-se todas as nomenclaturas em inglês, porém existem empresas brasileiras que padronizam a escrita em português. Isso é relativo e normalmente é acordado no início do projeto com todo o time.

Padronização

A padronização ocorre porque todos precisam compreender o dado como ele é. Durante o desenvolvimento web, mais de uma pessoa trabalha na construção do código e, sem essa padronização, pode ocorrer um sério problema, que é cada pessoa tratar o dado de uma forma diferente.

Imagine um projeto grande, em que cerca de cinco desenvolvedores web estão trabalhando juntos. Sem a padronização, pode ocorrer algo como isto:

Figura 3 — Padronização exemplo



```
LocalDateTime createDt;  
LocalDateTime createDate;  
LocalDateTime create_date;  
LocalDateTime date_create;  
LocalDateTime dtCreate;
```

Fonte: O próprio autor (2022).

A figura 3 relata um problema que pode ocorrer pela falta de padronização. Imagine diversos arquivos de código da mesma aplicação web, nos quais a data de criação do dado tem uma nomenclatura diferente.

Pode acontecer que cada desenvolvedor compreenda que aquele dado é diferente, e isso acarreta um problema, pois fica difícil entender que essa nomenclatura é a mesma em todos os arquivos.

A figura 3 apresenta um exemplo simples, porém, em grandes projetos, a simples falta de padronização pode resultar em dificuldade de manutenção do código e até em alguns erros no futuro próximo, visto que as aplicações web estão em constante atualização.

Grandes classes

Possuir classes e códigos com muitas linhas pode gerar um grande problema, isso porque, quando você tem muitos códigos em um local só, a própria escrita do código fica confusa e de difícil manutenção.

Imagine um arquivo do seu projeto com 930 linhas, no qual você precisa fazer uma alteração em uma linha de código, porém, ao realizar um teste, ocorre um erro na linha 530. Porém, ao chegar à linha 530, você percebe que ela está vazia. Nessa situação, onde está o erro?

Isso acontece diariamente com alguns desenvolvedores web, por conta disso é recomendado que os arquivos de código de um sistema web possuam poucas linhas e que o código seja reutilizado. Se você está escrevendo muitas linhas em um mesmo arquivo, cuidado! Você pode estar criando um monstro!

Parâmetros e objetos

Figura 4 — Método diversos parâmetros

```
void userAge(Long identifie,  
             String name,  
             String birthDate,  
             String documentNumber,  
             LocalDateTime currentDate,  
             String phoneNumber) {
```

Fonte: O próprio autor (2022).

A figura 4 apresenta um método recebendo como parâmetro muitos dados, isso pode ocasionar problemas nas tratativas dentro do método, quando estamos a utilizar uma linguagem de programação orientada a objetos, como Java, C# etc.

Escrever um método que recebe muitos parâmetros é uma quebra de boa prática. Imagine que, nesse método nomeado **userAge**, é necessário manipular todos os dados que recebeu por parâmetro, isso não é interessante.

Sempre que houver vários parâmetros em um método utilizando uma linguagem OO (Orientada a Objetos), pode ser que esses parâmetros se tornem um objeto. Isto é uma vantagem da OO: quando bem aplicada, reduz o código e o torna mais legível.

Figura 5 — Método diversos parâmetros

```
class User {  
    Long identifiern;  
    Usage  
    String name;  
    String birthDate;  
    String documentNumber;  
    LocalDateTime currentDate;  
    String phoneNumber;  
  
    String getUsername(User user) {  
        return user.name;  
    }  
}
```

Fonte: O próprio autor (2022).

Analisando a figura 5, encontra-se a construção de um método utilizando o OO. Não são enviados vários parâmetros dentro de um método; é enviado um objeto, e este possui diversos atributos. Para acessar esses recursos, é necessário somente indicar qual deles deseja acessar, como **user.name**, no exemplo da figura 5.



Videoaula 2

Utilize o QR Code para assistir!

Assista ao vídeo para entender melhor o assunto.



Reutilização

É preciso desenvolver sistemas web já imaginando as possíveis necessidades futuras, isso porque, como já aprendemos nas aulas anteriores, um sistema web precisa ser longo. A utilização dessa prática também é conhecida como DRY (Don't Repeat Yourself; em português, "não se repita").

Quando se escreve muitos códigos, é comum acabar replicando if's, condições e funções. Essa boa prática vem dizer que isso precisa ser evitado; caso já exista um, não o repita, utilize o já existente.

Uma das práticas que favorecem a longevidade de uma aplicação web é a reutilização: sempre que possível, construa códigos que possam ser reutilizados no futuro, para que outras funcionalidades possam utilizá-lo.

Pensando nisso, algumas linguagens de programação já possuem alguns recursos para serem reutilizados, como conversão de tipo de valores. Imagine você escrever o código de validação novamente sempre que precisar validar o CPF de um usuário. Em casos como conversão de valores e validação de dados, normalmente se constrói o código somente uma vez e ele é reutilizado diversas vezes.

Desenvolver em blocos

Imagine que um sistema web é um grande quebra-cabeça, em que as peças vão se encaixando e, no final, com todas as peças encaixadas, obtém-se uma aplicação web. Agora imagine uma peça de quebra-cabeça que possa se encaixar com outras peças. É assim que funciona a reutilização de códigos.

Desenvolver em blocos facilita muito a reutilização de códigos, além de tornar mais assertivo o desenvolvimento, visto que cada um tem sua responsabilidade e, com isso, cada objeto; classe; método possui seu objetivo claro e realiza um tipo de operação.

Indicação de Vídeo

Como desenvolver boas práticas de programação? Com Fabio Akita. Disponível em: <https://www.youtube.com/watch?v=GUanHEGlje4>. Acesso em: 30 set. 2022.

Evite comentários

Comentar no próprio código é uma forma de documentar também o que ele realiza corretamente, então por que evitar comentários?

Quando você tem muitos comentários para explicar o que aquele determinado trecho de código está realizando, isso pode ser um problema, pois seu código por si só não está escrito de maneira clara.

O código precisa expor o que ele mesmo realiza, sem muita necessidade de comentários ou que outro desenvolvedor explique o que ele está fazendo. Dessa forma, o código precisa **conversar e expor** o que ele está fazendo para o desenvolvedor que o está analisando.



Videoaula 3

Utilize o QR Code para assistir!

Assista ao vídeo para entender melhor o assunto.



Trate os erros

Por menores que sejam os erros em um sistema web, eles precisam ser sempre resolvidos, pois, quando não solucionados, resultam em uma péssima experiência para o usuário, o que torna ruim a imagem do software.

Além disso, quando diversos erros pequenos não são solucionados, pode ser que no futuro ocorra uma falha no sistema, gerando interrupções desagradáveis, ainda mais quando se fala de um software na web. Isso pode gerar até custos para a empresa que contratou o desenvolvimento desse software.

Então, sempre que possível, busque corrigir todos os erros, por menores que sejam, pois essa é uma boa prática.

Testes limpos e claros

Realizar testes do código e da aplicação como um todo é algo bem comum para um desenvolvedor web, porém esses testes, assim como o código,

precisam ser bem escritos para que se tenha certeza das validações que eles executam.

Para desenvolver testes limpos e claros, é necessário considerar alguns pontos, como:

- Testes rápidos: um teste não pode demorar muito, precisa ser ágil em sua validação.
- Testes independentes: testar qualquer parte ou trecho de código da aplicação precisa ser um gesto independente, não pode impactar o sistema inteiro.
- Passíveis de validação: um teste precisa apresentar um retorno, para que se mensure se foi um sucesso ou um fracasso.
- Pontuais: os testes precisam fazer sentido no contexto de código atual, sendo assim algumas práticas defendem a escrita antecipada dos códigos, antes mesmo do desenvolvimento.

Realizar testes é importante, até porque com eles é que se tem a certeza do funcionamento do software. Em algumas empresas, sistemas web sem testes simplesmente não podem ser entregues ao cliente.

A aplicação dessas boas práticas nem sempre é possível; durante o desenvolvimento de um projeto, porém, vale lembrar que são **práticas**, e não **regras**, portanto não queremos dizer que existe uma forma correta de desenvolver sistemas web e uma forma incorreta.

Essas práticas simplesmente foram sendo construídas durante o tempo de acordo com as dificuldades e necessidades que foram vistas como comuns entre vários projetos. Em alguns casos, pode ser que, para entregar rapidamente o projeto ou para corrigir um erro que está acontecendo no sistema, seja necessário colocar as boas práticas de lado e depois refatorar o código.

Enfim, compreender que essas práticas são voláteis e que podem ser aplicadas em qualquer linguagem de programação, de acordo com o que é necessário, é essencial para que um desenvolvedor web construa aplicações cada vez mais coerentes.

Aula 02 — Patterns: como aplicar?

Olá, aluno, tudo bem? Nesta aula vamos conhecer melhor o que são *patterns*, mais detalhadamente, *Design Patterns* e como eles funcionam.

Antes de qualquer coisa, é preciso saber o que é um *pattern*. Um pattern, traduzindo para o português, é um padrão, como se fosse um template (modelo) de algo que deve ser construído.

Mas o que é *Design*?

Bom *Design* tem vários apontamentos dentro do dicionário, porém no contexto de programação de sistemas web, design é um conjunto de objetos criados sob tais critérios.

Dessa forma, *Design Patterns* são modelos de código criados sob alguns critérios. Contudo, até chegar a esse modelo, alguns desenvolvedores tiveram diversas dificuldades até conseguir idealizar modelos que pudessem auxiliar a todos durante o desenvolvimento web.



Videoaula 1

Utilize o QR Code para assistir!

Assista ao vídeo para entender melhor o assunto.



Os Design Patterns que iremos conhecer nesta disciplina foram apresentados por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides e estão presentes no livro: *Design patterns: elements of reusable object-oriented software*.

O livro foi escrito há tempos e é considerado referência em Design Patterns, pois traz de modo sucinto o funcionamento de cada modelo, onde cada um também tem um nome específico e com detalhes para implementação.

Os padrões que vamos conhecer podem ser aplicados em diversas linguagens de programação e não são dependentes de framework ou IDE (Ambiente de Desenvolvimento Integrado), mas são independentes e funcionam em diversos aspectos.

Nesta aula vamos conhecer e demonstrar exemplos dos patterns mais utilizados. Enfim, os padrões (patterns) são divididos em três grandes grupos, sendo eles:

Creational Design Patterns

São padrões relacionados à criação de objetos e classes, em que há exemplos de modos de criar objetos de acordo com cada necessidade. Mas sempre são modelos de boas práticas relacionadas à criação. São patterns desse grupo:

Abstract Factory (Fábrica Abstrata)

Permite a criação de objetos sem especificar seu tipo concreto. Esse modelo de *patterns* defende a utilização de interfaces para manter os dados da mesma família, combinando o que eles têm em comum.

Builder (Construtor)

Usada para criar objetos complexos. Imagine que você está construindo um notebook. Ele tem diversos atributos: tela, teclado, *touchpad*, processador, placa-mãe, entrada USB B, webcam e outros.

Nesse caso é simples: posso ter um construtor gigante com todos esses dados já carregados na hora que criar esse objeto. Porém, depois de dois meses que o meu sistema estava no ar, surgiu a demanda de vender agora notebooks sem a webcam. E, agora, o que vou fazer?

A proposta do Builder é, em vez de ter um construtor gigante com diversos parâmetros, dividi-los em outros pequenos construtores, e com isso cada objeto carrega somente os dados que pretende utilizar.

Figura 6 — Builder

```
Memoria builderMemoria(){};  
Webcam builderWebcam() {};  
Mouse builderMouse() {};  
Teclado builderTeclado() {};
```

Fonte: O próprio autor (2022).

A figura 6 demonstra de forma abstrata o funcionamento do builder, dessa forma é possível carregar de modo separado cada objeto que precisar possuir um novo atributo. É um modelo muito utilizado quando se possui um construtor muito complexo e grande.

Factory Method (Método de fábrica)

Cria objetos sem especificar a classe exata a ser criada. Por exemplo, você vai construir um sistema de transporte com carros, e, depois de um tempo, seu sistema é um sucesso e tem pessoas querendo que ele também trate do transporte com bicicletas, só que, para fazer isso, você precisa alterar todo o seu objeto carro no seu sistema.

A solução proposta por este pattern é criar um objeto que vai receber as chamadas tanto de carro quanto de bicicleta, dessa forma criando uma abstração para representar esses dois objetos.

Prototype (Protótipo)

Cria um objeto a partir de um objeto existente. A proposta é criar uma interface que possibilite clonar os objetos, e todos os objetos que podem ser clonados implementam essa interface para que possam sê-lo.

Isso pode acontecer, pois, ao tentar clonar um objeto sem esse pattern, pode ocorrer de faltar algum dado, pois o objeto pode possuir atributos privados.

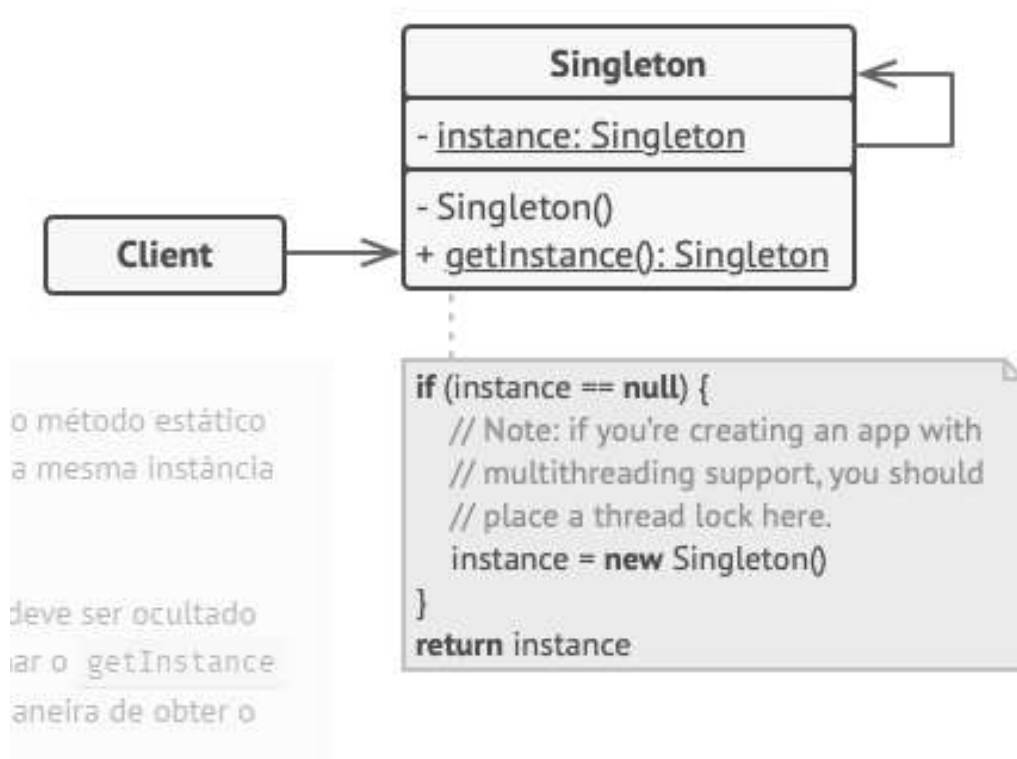
A principal vantagem é clonar objetos sem precisar acoplar os códigos de clonagem a um objeto concreto, ou seja, a clonagem é responsabilidade da interface.

Singleton (Único)

Garante que apenas uma instância de um objeto seja criada. Esse é um pattern fortemente utilizado e pode ser que você já até o tenha utilizado para realizar alguma tarefa universitária ou laboral.

A proposta do Singleton é garantir que somente uma instância de um objeto seja criada e que todos tenham acesso à mesma instância do objeto. Normalmente é utilizado para conexões com banco de dados, para que não seja criada sempre uma instância os projetos web em sua maioria utilizam o modelo Singleton.

Figura 7 — Modelo Singleton



Fonte: Refactoring Guru (2022).

A figura 7 demonstra como funciona a construção de uma classe com o modelo Singleton, o qual, sempre antes de criar uma nova instância, valida se já existe uma instância criada; caso exista, considera-a como tal.



Videoaula 2

Utilize o QR Code para assistir!

Assista ao vídeo para entender melhor o assunto.



Structural Design Patterns

São padrões tratados no design para identificar uma maneira simples de relacionamento entre diversas entidades, e estão sempre vinculados à estrutura das entidades e a como funcionam os seus relacionamentos.

Adapter (Adaptador)

Permite que duas classes incompatíveis trabalhem juntas, envolvendo uma interface em torno de uma das classes existentes. A proposta desse pattern é criar um objeto que torne possível a interação entre objetos diferentes.

Para isso a criação de um adaptador é como um conversor, em que os dados vão de um objeto a outro objeto de acordo com a necessidade.

Bridge (Ponte)

Desacopla uma abstração para que duas classes possam variar independentemente, permitindo que uma classe muito grande seja dividida em duas hierarquias separadas.

Imagine um objeto camiseta, este objeto pode possuir duas hierarquias: tamanho e cor, pois, em meu sistema, quero vender camisetas de acordo com tamanho e cor, utilizando do modelo Bridge é possível realizar isso, tornando as hierarquias independentes uma da outra.

Composite (Composto)

Leva um grupo de objetos em um único objeto. A proposta é compor objetos como uma estrutura de árvore. Em vez de levar cada objeto separado, você pode consolidar os objetos todos dentro de uma árvore e acessá-la.

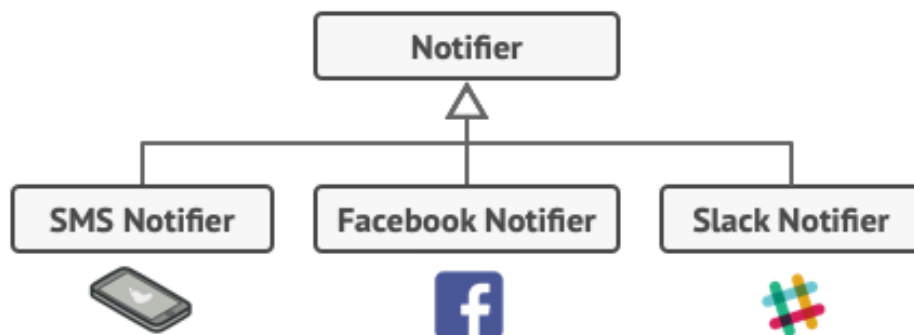
Funciona como a estrutura de uma escola, quando você considera que, dentro de um objeto escola, você precisa ter outros objetos dentro, como salas, turmas, professores, alunos e cursos.

Utilizando esse pattern, ao chamar o objeto escola, juntamente com ele se recebe todos os outros objetos que o compõem.

Decorator (Decorador)

Permite que o comportamento de um objeto seja estendido dinamicamente em tempo de execução, colocando esses objetos dentro de novos objetos especiais que possuem esses comportamentos.

Figura 8 — Modelo Decorator



Fonte: Refactoring Guru (2022).

A figura 8 apresenta o funcionamento do modelo Decorator, em que as classes SMS Notifier, Facebook Notifier e Slack Notifier são subclasses que têm o comportamento do Notifier.

Facade (Fachada)

Fornece uma interface simples para um objeto subjacente mais complexo. A proposta é sempre apresentar uma fachada para chamar um objeto ou um conjunto de objetos. Imagine diversos objetos de conversão de áudio, mas, em vez de chamar cada um separadamente, chamar somente o objeto de fachada.

Proxy (Procuração)

Fornece uma interface de espaço reservado para um objeto subjacente, a fim de controlar o acesso e reduzir custos ou a complexidade, deixando esse objeto em um local reservado e, assim, controlando o acesso a ele.

Dessa forma, é possível executar com o pattern Proxy uma operação antes ou depois de acessar o objeto que está reservado para ser acessado.



Videoaula 3

Utilize o QR Code para assistir!

Assista ao vídeo para entender melhor o assunto.



Behavior Design Patterns

São padrões que envolvem a comunicação comum entre objetos e outros modelos de dados dentro de um sistema web bem como seu comportamento, com objetivo de identificar o que é comum na comunicação entre esses objetos e a transição de dados.

Chain of Responsibility (Cadeia de Responsabilidade)

Delega comandos para uma cadeia de objetos de processamento. Desse modo um tipo de request pode possuir várias etapas para serem realizadas, então nesse modelo existem manipuladores.

Os manipuladores podem escolher realizar uma operação e finalizar aqui o processamento da cadeia ou passar para o próximo manipulador realizar outro processamento.

Command (Comando)

Cria objetos que realizam encapsulamento de ações e parâmetros. Dessa forma construindo da chamada um objeto autônomo, com isso implementando de cada modo diferente e dando a liberdade para que esse objeto autônomo trate das execuções do modo que deseja.

Iterator (Iterador)

Acessa os itens de um objeto de modo sequencial sem expor sua representação subjacente. Normalmente é utilizado quando se tem uma estrutura de dados complexa como uma estrutura em árvore onde os dados não estão somente em uma lista.

Mediator (Mediador)

Possibilita diminuir o acoplamento entre as classes por ser única a classe que possui o conhecimento dos seus dados e comportamentos. Com isso restringindo a comunicação direta entre objetos, criando um mediador entre a comunicação.

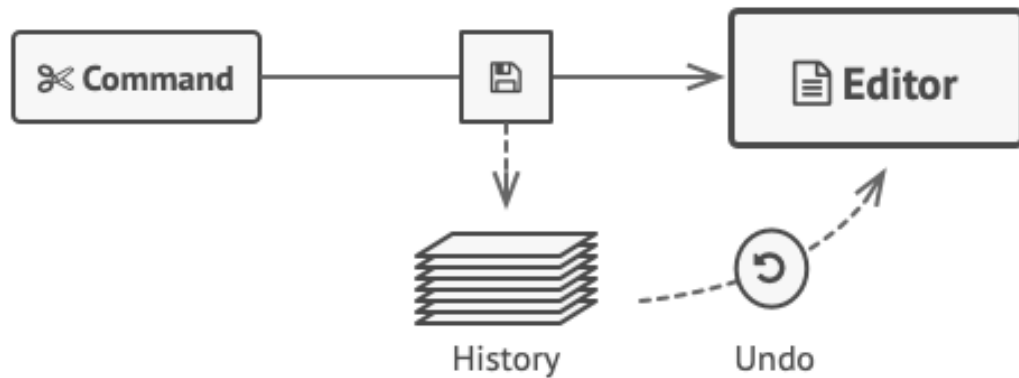
Como se fosse um semáforo que media a comunicação entre os objetos.

Memento (Lembrança)

Possibilita a restauração de um objeto ao seu estado anterior. Com isso você tem um objeto chamado Computador e um chamado Memento, que armazena o estado do objeto e suas mudanças com suas propriedades e comportamentos.

Dessa forma, sempre que necessário você pode visualizar o estado anterior do objeto e recuperar como o objeto estava no momento anterior basta realizar a restauração. Assim sendo desacoplando alguns conceitos.

Figura 9 — Exemplo Memento



Fonte: Refactoring Guru (2022).

A figura 9 representa um exemplo do funcionamento do pattern Memento, onde antes de realizar qualquer coisa dentro do objeto principal neste caso Editor, essa execução é salva, formando um modelo de histórico sobre o objeto.

Observer (Observador)

É um padrão que permite que vários objetos observadores vejam um evento. Em projetos e frameworks que trabalham orientados a eventos, o pattern observer é grandemente utilizado, visto que torna possível que vários objetos observem o mesmo evento e realizem operações diferentes.

State (Estado)

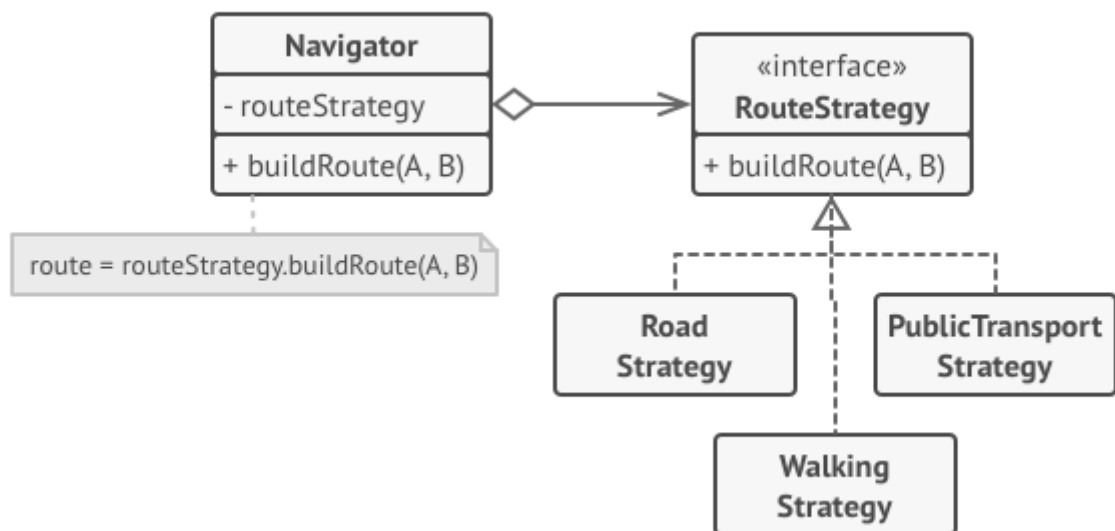
Possibilita ao objeto mudar seu comportamento de acordo com o seu estado, por isso o nome State. É um pattern baseado em um modelo de uma máquina de estados finitos, dessa forma, em qualquer momento, o objeto pode estar somente se comportando de acordo com o estado que recebeu.

Esses estados modificam o comportamento dentro do sistema, possibilitando a visualização e realização de operações de acordo com cada comportamento vinculado ao estado recebido no objeto.

Strategy (Estratégia)

Permite definir um conjunto de algoritmos e colocar cada um em classes separadas e tornar seus objetos intercambiáveis. Normalmente, quando se tem uma classe que pode realizar uma operação de diversas formas, utiliza-se esse modelo, segregando em diversas classes essas operações e algoritmos — classes que são nomeadas de estratégias — e então é preciso sempre saber em qual estratégia vai ser utilizada, por isso também é necessário criar uma classe principal.

Figura 10 — Modelo Strategy



Fonte: Refactoring Guru (2022).

A figura 10 representa a implementação do modelo Strategy, cuja classe principal chama Navigator e sabe qual é a rota da estratégia que vai ser utilizada, porém só visualiza a interface RouteStrategy, em que é implementada por um conjunto de classes com algoritmos diferentes.

Template (Modelo)

Permite definir uma estrutura de algoritmo dentro de uma superclasse, porém permite que as subclasses realizem alterações de acordo com as necessidades, mas sem alterar a estrutura definida na superclasse.

Neste modelo é necessário abstrair o que existe em comum entre as subclasses para a superclasse, mesmo que uma subclasse precise, no momento, alterar algum algoritmo, pois isso é permitido.

Visitor (Visitante)

Esse pattern permite categorizar os algoritmos dos objetos que ele opera, visitando e acessando os dados nos objetos que é necessário o algoritmo conhecer. Quando se possui dúvidas de quão grandes podem se tornar as opções de um sistema web, é recomendado utilizar este pattern.

Porém também é um pattern bem conhecido e apoiado por muitos desenvolvedores, pois pode ser implementado em sistemas já existentes com facilidade.

Muitas vezes alguns desenvolvedores já utilizaram algum pattern e acabaram não sabendo que o estão utilizando, pois alguns são modelos já adotados e amplamente favorecidos dentro das empresas.

Conhecer o conceito de cada pattern e como são distribuídos favorece o momento de desenvolver sistemas web, em que é preciso, muitas vezes, aplicar alguns modelos para que o desenvolvimento seja mais fluido e com qualidade.

Encerramento da Unidade

Chegamos ao final da nossa terceira unidade, na qual você teve contato com os conceitos dos patterns de desenvolvimento e boas práticas. Com certeza, você já tinha uma pequena noção de muito do que foi mostrado aqui.

Na aula 1, conhecemos algumas boas práticas de desenvolvimento web, como podem ser aplicadas e os cuidados necessários.

Na aula 2, foram apresentados e discutidos os patterns de desenvolvimento e como podem ser aplicados.

Juntando as duas aulas, obtivemos descobertas interessantes de como tratar o desenvolvimento de um sistema web. Caro aluno, pense um pouco em como os conhecimentos desta unidade podem te ajudar nas suas atividades e até mesmo no que você pode se aprofundar no tocante a esses conhecimentos. Até a próxima.

Bons estudos!

Referências

DESIGN patterns. **Refactoring Guru**. Disponível em: <https://refactoring.guru/design-patterns>. Acesso em: 29 jul. 2022.

BERTAGNOLLI, S.C; MILETTO, E.M. **Desenvolvimento de software II: introdução ao desenvolvimento web com HTML, CSS, JavaScript e PHP**. Porto Alegre: Bookman, 2014.

GAMMA, Erich. *et al.* **Design patterns**: elements of reusable object-oriented software. Boston: Addison-Wesley, 2000.



UNIFIL.BR