## Resumo

programação estruturada é sequencial. tem seus 3 pontos fundamentais na estrutura e segue esses pontos.

programação orientada a objetos, na analogia do carro, Podemos dizer que o carro novo é um objeto, onde suas características[duas portas, azul] são seus atributos (dados atrelados ao objeto) e seus comportamentos[acelerar, frear] são ações ou métodos.

Nela tem o conceito de encapsulamento, você esconde o ‘como é feito’ e mostra simplesmente o resultado.

A herança, que herdam de outros pais, como No caso do carro, dizemos então que um Honda Fit "Cross" é um tipo de Honda Fit, e o que muda são alguns atributos (paralama reforçado, altura da suspensão etc), e um dos métodos da classe (acelerar, pois agora há tração nas quatro rodas), mas todo o resto permanece o mesmo, e o novo modelo recebe os mesmos atributos e métodos do modelo clássico.

polimorfismo é quando um objeto tem o mesmo método mas foi implementado de formas diferentes tipo o carro que toca música mas em um é em CD e no outro é em pendrive

componentização é tipo. vc tem um programa que executa uma função A e outro que executa outra função B. vc pega esse programas isolados e transforma eles em componentes de um programa C que é completamente separado de B e A mas executa as mesmas tarefas

## 

## **Programação orientada a objetos e programação estruturada**

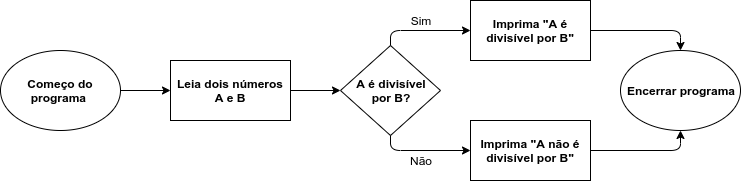
Como a maioria das atividades que fazemos no dia a dia, programar também possui modos diferentes de se fazer. Esses modos são chamados de **paradigmas de programação** e, entre eles, estão a **programação orientada a objetos** (POO) e a programação estruturada. Quando começamos a utilizar linguagens como [Java](https://www.alura.com.br/curso-online-java-introducao-orientacao-objetos), [C#](https://www.alura.com.br/curso-online-csharp-parte-2-introducao-orientacao-objetos), [Python](https://www.alura.com.br/curso-online-python-3-intro-orientacao-objetos) e outras que possibilitam o paradigma orientado a objetos, é comum errarmos e aplicarmos a programação estruturada achando que estamos usando recursos da orientação a objetos.

Na programação estruturada, um programa é composto por três tipos básicos de estruturas:

* sequências: são os comandos a serem executados
* condições: sequências que só devem ser executadas se uma condição for satisfeita (exemplos: if-else, switch e comandos parecidos)
* repetições: sequências que devem ser executadas repetidamente até uma condição for satisfeita (for, while, do-while etc)

Essas estruturas são usadas para processar a entrada do programa, alterando os dados até que a saída esperada seja gerada. Até aí, nada que a programação orientada a objetos não faça, também, certo?

A diferença principal é que na programação estruturada, um programa é tipicamente escrito em uma única rotina (ou função) podendo, é claro, ser quebrado em subrotinas. Mas o fluxo do programa continua o mesmo, como se pudéssemos copiar e colar o código das subrotinas diretamente nas rotinas que as chamam, de tal forma que, no final, só haja uma grande rotina que execute todo o programa.



Além disso, o acesso às variáveis não possuem muitas restrições na programação estruturada. Em linguagens fortemente baseadas nesse paradigma, restringir o acesso à uma variável se limita a dizer se ela é visível ou não dentro de uma função (ou módulo, como no uso da palavra-chave static, na linguagem C), mas não se consegue dizer de forma nativa que uma variável pode ser acessada por apenas algumas rotinas do programa. O contorno para situações como essas envolve práticas de programação danosas ao desenvolvimento do sistema, como o uso excessivo de variáveis globais. Vale lembrar que variáveis globais são usadas tipicamente para manter estados no programa, marcando em qual parte dele a execução se encontra.

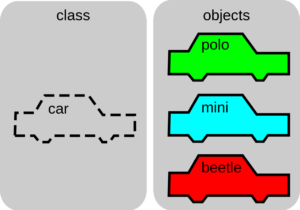
A **programação orientada a objetos** surgiu como uma alternativa a essas características da programação estruturada. O intuito da sua criação também foi o de aproximar o manuseio das estruturas de um programa ao manuseio das coisas do mundo real, daí o nome "objeto" como uma algo genérico, que pode representar qualquer coisa tangível.

Esse novo paradigma se baseia principalmente em dois conceitos chave: **classes** e **objetos**. Todos os outros conceitos, igualmente importantes, são construídos em cima desses dois.

## **O que são classes e objetos?**

Imagine que você comprou um carro recentemente e decide modelar esse carro usando programação orientada a objetos. O seu carro tem as características que você estava procurando: um motor 2.0 híbrido, azul escuro, quatro portas, câmbio automático etc. Ele também possui comportamentos que, provavelmente, foram o motivo de sua compra, como acelerar, desacelerar, acender os faróis, buzinar e tocar música. Podemos dizer que o carro novo é um *objeto*, onde suas características são seus *atributos* (dados atrelados ao objeto) e seus comportamentos são ações ou *métodos*.

Seu carro é um objeto seu mas na loja onde você o comprou existiam vários outros, muito similares, com quatro rodas, volante, câmbio, retrovisores, faróis, dentre outras partes. Observe que, apesar do seu carro ser único (por exemplo, possui um registro único no Departamento de Trânsito), podem existir outros com exatamente os mesmos atributos, ou parecidos, ou mesmo totalmente diferentes, mas que ainda são considerados *carros*. Podemos dizer então que seu objeto pode ser classificado (isto é, seu *objeto pertence à uma classe*) como um carro, e que seu carro nada mais é que uma *instância* dessa *classe* chamada "carro".



Assim, abstraindo um pouco a analogia, uma classe é um conjunto de características e comportamentos que definem o conjunto de objetos pertencentes à essa classe. Repare que a classe em si é um conceito abstrato, como um molde, que se torna concreto e palpável através da criação de um objeto. Chamamos essa criação de *instanciação da classe*, como se estivéssemos usando esse molde (classe) para criar um objeto.

### **Exemplo em Java**

public class Carro {

Double velocidade;

String modelo;

public Carro(String modelo) {

this.modelo = modelo;

this.velocidade = 0;

}

public void acelerar() {

*/\* código do carro para acelerar \*/*

}

public void frear() {

*/\* código do carro para frear \*/*

}

public void acenderFarol() {

*/\* código do carro para acender o farol \*/*

}

}

### **Exemplo em Python**

class Carro:

def \_\_init\_\_(self, modelo):

self.modelo = modelo;

self.velocidade = 0

def acelerar(self):

*# Codigo para acelerar o carro*

def frear(self):

*# Codigo para frear o carro*

def acenderFarol(self):

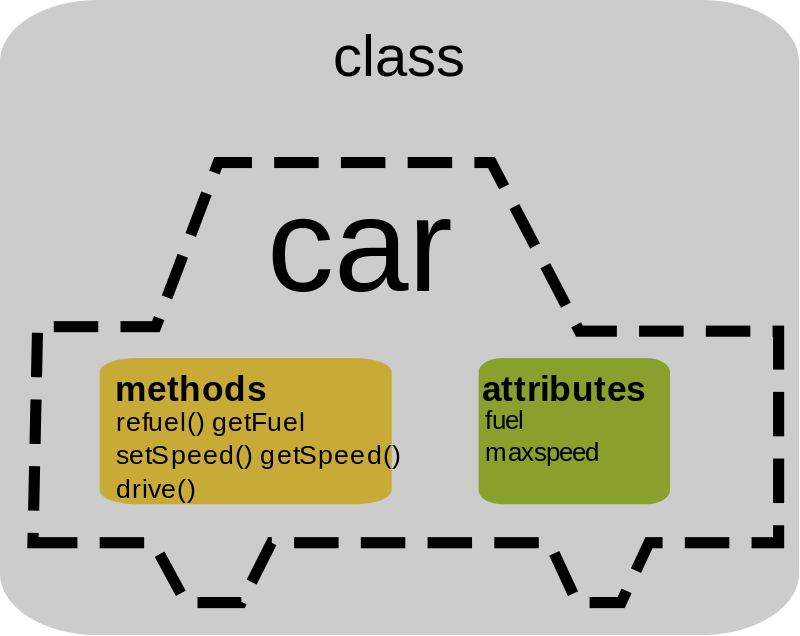
*# Codigo para acender o farol do carro*

## **Encapsulamento, herança e polimorfismo: as principais características da POO**

As duas bases da **POO** são os conceitos de classe e objeto. Desses conceitos, derivam alguns outros conceitos extremamente importantes ao paradigma, que não só o definem como são as soluções de alguns problemas da **programação estruturada**. Os conceitos em questão são o *encapsulamento*, a *herança*, as *interfaces* e o *polimorfismo*.

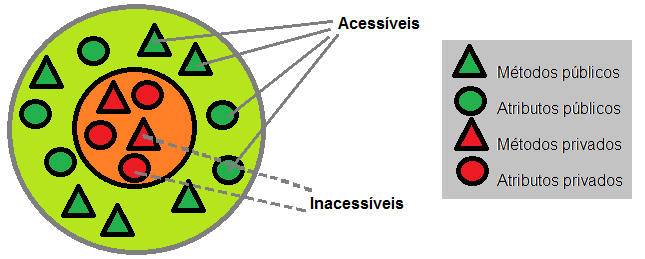
### **Encapsulamento**

Ainda usando a analogia do carro, sabemos que ele possui atributos e métodos, ou seja, características e comportamentos. Os métodos do carro, como acelerar, podem usar atributos e outros métodos do carro como o tanque de gasolina e o mecanismo de injeção de combustível, respectivamente, uma vez que acelerar gasta combustível.



No entanto, se alguns desses atributos ou métodos forem facilmente visíveis e modificáveis, como o mecanismo de aceleração do carro, isso pode dar liberdade para que alterações sejam feitas, resultando em efeitos colaterais imprevisíveis. Nessa analogia, uma pessoa pode não estar satisfeita com a aceleração do carro e modifica a forma como ela ocorre, criando efeitos colaterais que podem fazer o carro nem andar, por exemplo.

Dizemos, nesse caso, que o método de aceleração do seu carro não é visível por fora do próprio carro. Na POO, um atributo ou método que não é visível de fora do próprio objeto é chamado de "privado" e quando é visível, é chamado de "público".



Mas então, como sabemos como o nosso carro acelera? É simples: não sabemos. Nós só sabemos que para acelerar, devemos pisar no acelerador e de resto o objeto sabe como executar essa ação sem expor como o faz. Dizemos que a aceleração do carro está *encapsulada*, pois sabemos o que ele vai fazer ao executarmos esse método, mas não sabemos como - e na verdade, não importa para o programa como o objeto o faz, só que ele o faça.

O mesmo vale para atributos. Por exemplo: não sabemos como o carro sabe qual velocidade mostrar no velocímetro ou como ele calcula sua velocidade, mas não precisamos saber como isso é feito. Só precisamos saber que ele vai nos dar a velocidade certa. Ler ou alterar um atributo encapsulado pode ser feito a partir de *getters e setters* (colocar referência).

Esse *encapsulamento* de atributos e métodos impede o chamado *vazamento de escopo*, onde um atributo ou método é visível por alguém que não deveria vê-lo, como outro objeto ou classe. Isso evita a confusão do uso de variáveis globais no programa, deixando mais fácil de identificar em qual estado cada variável vai estar a cada momento do programa, já que a restrição de acesso nos permite identificar quem consegue modificá-la.

#### **Exemplo em Java**

public class Carro {

private Double velocidade;

private String modelo;

private MecanismoAceleracao mecanismoAceleracao;

private String cor;

*/\* Repare que o mecanismo de aceleração é inserido no carro ao ser construído, e*

*não o vemos nem podemos modificá-lo, isto é, não tem getter nem setter.*

*Já o "modelo" pode ser visto, mas não alterado. \*/*

public Carro(String modelo, MecanismoAceleracao mecanismoAceleracao) {

this.modelo = modelo;

this.mecanismoAceleracao = mecanismoAceleracao;

this.velocidade = 0;

}

public void acelerar() {

this.mecanismoAceleracao.acelerar();

}

public void frear() {

*/\* código do carro para frear \*/*

}

public void acenderFarol() {

*/\* código do carro para acender o farol \*/*

}

public Double getVelocidade() {

return this.velocidade

}

private void setVelocidade() {

*/\* código para alterar a velocidade do carro \*/*

*/\* Como só o próprio carro deve calcular a velocidade,*

*esse método não pode ser chamado de fora, por isso é "private" \*/*

}

public String getModelo() {

return this.modelo;

}

public String getCor() {

return this.cor;

}

*/\* podemos mudar a cor do carro quando quisermos \*/*

public void setCor(String cor) {

this.cor = cor;

}

}

#### **Exemplo em Python**

*# Exemplo da classe Carro em Python*

class Carro:

def \_\_init\_\_(self, modelo, mecanismoAceleracao):

self.\_\_modelo = modelo;

self.\_\_velocidade = 0

self.\_\_mecanismoAceleracao = mecanismoAceleracao

def acelerar(self):

mecanismoAceleracao.acelerar()

def frear(self):

*# Codigo para frear o carro*

def acenderFarol(self):

*# Codigo para acender o farol do carro*

def getVelocidade(self):

return self.velocidade

def \_\_setVelocidade(self):

*# Codigo para alterar a velocidade por dentro do objeto*

def getModelo(self):

return self.modelo

def getCor(self):

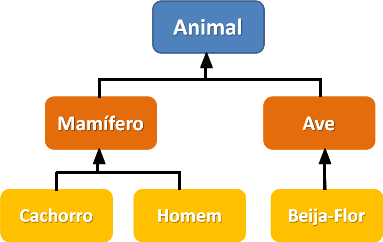
return self.cor

def setCor(self, cor):

self.cor = cor

### **Herança**

No nosso exemplo, você acabou de comprar um carro com os atributos que procurava. Apesar de ser único, existem carros com exatamente os mesmos atributos ou formas modificadas. Digamos que você tenha comprado o modelo Fit, da Honda. Esse modelo possui uma outra versão, chamada WR-V (ou "Honda Fit Cross Style"), que possui muitos dos atributos da versão clássica, mas com algumas diferenças bem grandes para transitar em estradas de terra: o motor é híbrido (aceita álcool e gasolina), possui um sistema de suspensão diferente, e vamos supor que além disso ele tenha um sistema de tração diferente (tração nas quatro rodas, por exemplo). Vemos então que não só alguns atributos como tambm alguns mecanismos (ou métodos, traduzindo para POO) mudam, mas essa versão "cross" ainda é do modelo Honda Fit, ou melhor, *é um tipo* do modelo.



Quando dizemos que uma classe A *é um tipo de* classe B, dizemos que a classe A *herda* as características da classe B e que a classe B é *mãe* da classe A, estabelecendo então uma relação de **herança** entre elas. No caso do carro, dizemos então que um Honda Fit "Cross" *é um tipo de* Honda Fit, e o que muda são alguns atributos (paralama reforçado, altura da suspensão etc), e um dos métodos da classe (acelerar, pois agora há tração nas quatro rodas), mas todo o resto permanece o mesmo, e o novo modelo recebe os mesmos atributos e métodos do modelo clássico.

#### **Exemplo em Java**

*// "extends" estabelece a relação de herança dom a classe Carro*

public class HondaFit extends Carro {

public HondaFit(MecanismoAceleracao mecanismoAceleracao) {

String modelo = "Honda Fit";

*// chama o construtor da classe mãe, ou seja, da classe "Carro"*

super(modelo, mecanismoAceleracao);

}

}

#### **Exemplo em Python**

*# As classes dentro do parênteses são as classes mãe da classe sendo definida*

class HondaFit(Carro):

def \_\_init\_\_(self, mecanismoAceleracao):

modelo = "Honda Fit"

*# chama o construtor da classe mãe, ou seja, da classe "Carro"*

super().\_\_init\_\_(self, modelo, mecanismoAceleracao)

### **Interface**

Muitos dos métodos dos carros são comuns em vários automóveis. Tanto um carro quanto uma motocicleta são classes cujos objetos podem acelerar, parar, acender o farol etc, pois são coisas comuns a automóveis. Podemos dizer, então, que ambas as classes "carro" e "motocicleta" *são* "automóveis".

Quando duas (ou mais) classes possuem comportamentos comuns que podem ser separados em uma outra classe, dizemos que a "classe comum" é uma *interface*, que pode ser "herdada" pelas outras classes. Note que colocamos a interface como "classe comum", que pode ser "herdada" (com aspas), porque uma interface não é exatamente um classe, mas sim um conjunto de métodos que todas as classes que herdarem dela devem possuir (implementar) - portanto, uma interface não é "herdada" por uma classe, mas sim *implementada*. No mundo do desenvolvimento de software, dizemos que uma interface é um "contrato": uma classe que implementa uma interface deve fornecer uma implementação a **todos** os métodos que a interface define, e em compensação, a classe implementadora pode dizer que ela é do tipo da interface. No nosso exemplo, "carro" e "motocicleta" são classes que implementam os métodos da interface "automóvel", logo podemos dizer que qualquer objeto dessas duas primeiras classes, como um Honda Fit ou uma motocicleta da Yamaha, são automóveis.

Um pequeno detalhe: uma interface não pode ser herdada por uma classe, mas sim implementada. No entanto, uma interface pode herdar de outra interface, criando uma hierarquia de interfaces. Usando um exemplo completo com carros, dizemos que a classe "Honda Fit Cross" herda da classe "Honda Fit", que por sua vez herda da classe "Carro". A classe "Carro" implementa a interface "Automóvel" que, por sua vez, pode herdar (por exemplo) uma interface chamada "MeioDeTransporte", uma vez que tanto um "automóvel" quanto uma "carroça" *são* meios de transporte, ainda que uma carroça *não seja* um automóvel.

#### **Exemplo em Java**

public interface Automovel {

void acelerar();

void frear();

void acenderFarol();

}

public class Carro implements Automovel {

*/\* ... \*/*

@Override

public void acelerar() {

this.mecanismoAceleracao.acelerar();

}

@Override

public void frear() {

*/\* código do carro para frear \*/*

}

@Override

public void acenderFarol() {

*/\* código do carro para acender o farol \*/*

}

*/\* ... \*/*

}

public class Moto implements Automovel {

*/\* ... \*/*

@Override

public void acelerar() {

*/\* código específico da moto para acelerar \*/*

}

@Override

public void frear() {

*/\* código específico da moto para frear \*/*

}

@Override

public void acenderFarol() {

*/\* código específico da moto para acender o farol \*/*

}

*/\* ... \*/*

}

#### **Exemplo em Python**

class Automovel():

def acelerar(self):

raise NotImplementedError()

def frear(self):

raise NotImplementedError()

def acenderFarol(self):

raise NotImplementedError()

class Carro(Automovel):

*# ...*

def acelerar(self):

*# Codigo para acelerar o carro*

def frear(self):

*# Codigo para frear o carro*

def acenderFarol(self):

*# Codigo para acender o farol do carro*

*# ...*

class Moto(Automovel):

*# ...*

def acelerar(self):

*# Codigo para acelerar a moto*

def frear(self):

*# Codigo para frear a moto*

def acenderFarol(self):

*# Codigo para acender a moto*

*# ...*

Nota: criar um erro do tipo NotImplementedError é apenas uma conveção para que, caso uma classe filha tente executar um método da classe mãe sem tê-la implementado, ocorra o erro. Em Python, as interfaces são criadas como classes normais que são herdadas pelas classes filhas. Existem formas de forçar a implementação por parte das classes filhas, mas para nosso exemplo essa abordagem é suficiente.

### **Polimorfismo**

Vamos dizer que um dos motivos de você ter comprado um carro foi a qualidade do sistema de som dele. Mas, no seu caso, digamos que a reprodução só pode ser feita via rádio ou *bluetooth*, enquanto que no seu antigo carro, podia ser feita apenas via cartão SD e *pendrive*. Em ambos os carros está presente o método "tocar música" mas, como o sistema de som deles é diferente, a forma como o carro toca as músicas é diferente. Dizemos que o método "tocar música" é uma forma de **polimorfismo**, pois dois objetos, de duas classes diferentes, têm um mesmo método que é implementado de formas diferentes, ou seja, um método possui *várias formas*, várias implementações diferentes em classes diferentes, mas que possuem o mesmo efeito ("polimorfismo" vem do grego *poli* = muitas, *morphos* = forma).

#### **Exemplo em Java**

public class Main {

public static void main(String[] args) {

Automovel moto = new Moto("Yamaha XPTO-100", new MecanismoDeAceleracaoDeMotos())

Automovel carro = new Carro("Honda Fit", new MecanismoDeAceleracaoDeCarros())

List<Automovel> listaAutomoveis = Arrays.asList(moto, carro);

for (Automovel automovel : listaAutomoveis) {

automovel.acelerar();

automovel.acenderFarol();

}

}

}

#### **Exemplo em Python**

def main():

moto = Moto("Yahama XPTO-100", MecanismoDeAceleracaoDeMotos())

carro = Carro("Honda Fit", MecanismoDeAceleracaoDeCarros())

listaAutomoveis = [moto, carro]

for automovel in listaAutomoveis:

automovel.acelerar()

automovel.acenderFarol()

Repare que apesar de serem objetos diferentes, moto e carro possuem os mesmos métodos acelerar e acenderFarol, que são chamados da mesma forma, apesar de serem implementados de maneira diferente.

## **Design Patterns**

Alguns problemas aparecem com tanta frequência em **POO** que suas soluções se tornaram padrões de design de sistemas e modelagem de código orientado a objeto, a fim de resolvê-los. Esses **padrões de projeto**, (ou **design patterns**) nada mais são do que formas padronizadas de resolver problemas comuns em linguagens orientadas a objetos. O livro "Design Patterns", conhecido como **Gof:Gang of Four**, é a principal referência nesse assunto, contendo os principais padrões usados em grandes projetos. A Alura também oferece cursos de *Design Patterns* em linguagens de programação como [Java](https://www.alura.com.br/curso-online-design-patterns), [Python](https://www.alura.com.br/curso-online-design-patterns-python) e [C#](https://www.alura.com.br/curso-online-design-patterns-dotnet).

## **Clean code e SOLID**

Em projetos desenvolvidos com POO, assim como em qualquer outro, o cdigo pode se tornar desorganizado e dificil de manter a médio e longo prazo. Em vista dessa situação, alguns princípios de boas práticas de programação e código limpo foram desenvolvidos como, por exemplo:

* KISS (*Keep It Simple, Stupid*, "Mantenha as coisas simples"): Sempre que um código for escrito, ele deve ser escrito da forma mais simples possível, para manter o código mais legível. Códigos complexos demais são mais difíceis de se manter, j que é mais difícil entender o que ele faz e como ele faz.
* DRY (*Don't Repeat Yourself*, "Não se repita"): Todo código escrito para resolver um problema deve ser escrito apenas uma vez, a fim de evitar repetição de código. É quase uma variação do KISS, dado que a repetição de código o torna mais confuso e difícil de manter e corrigir, se necessário.

Alguns cursos da Alura abordam esses assuntos, como o curso de [SOLID com Java](https://www.alura.com.br/curso-online-orientacao-a-objetos-avancada-e-principios-solid), [SOLID com PHP](https://www.alura.com.br/curso-online-solid-php-principios-orientacao-a-objetos), e também nosso artigo sobre [o que é Clean Code](https://www.alura.com.br/artigos/o-que-e-clean-code).

Além dos [**design patterns**](https://www.alura.com.br/curso-online-introducao-design-patterns-java) e dos princípios de código limpo existe um conjunto de técnicas, mais generalizadas que os design patterns, que ajudam a criar código orientado a objeto de forma a deixá-lo mais maleável, possibilitando uma manutenção e expansão mais suave e descomplicada do código ao longo do tempo.

No blog da Caelum, você também pode ver sobre os grandes erros da orientação a objetos, como o [uso de herança](https://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-heranca/) de maneira excessiva e a [criação de getters e setters](https://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/) sem necessidade. Há também o capítulo de Arquitetura Java que fala sobre [interface e implementação](https://www.alura.com.br/especial/poo-interface-implementacao).

## Componentização/ programação orientada a componentes

A componentização de software pode ser descrita como uma unidade de software que pode ser unida a outras unidades de software para formar um sistema de maior tamanho.

s componentes costumam ser úteis pela capacidade de reutilização em outras aplicações. Com isso, várias etapas de desenvolvimento podem ser banidas. Através dos componentes é possível desenvolver um processo de criação de ativos digitais, já que são elementos de software que podem ser plugados com uma nova interface totalmente definida.

Vantagens da utilização da componentização:

* Facilita a convergência tecnológica através do uso de tecnologia orientada a componentes/serviços;
* Costuma ser base para modelo de camadas, já que é útil para evitar a existência de programas de maior tamanho que exercem várias funções, e que poderiam ser divididos em programas/componentes menores;
* Colabora na integração de informações entre vários canais de acesso;
* Manutenção e evolução mais fácil, já que os módulos são independentes entre si, o que possibilita a extensão de software com baixo impacto no software que está sendo usado.