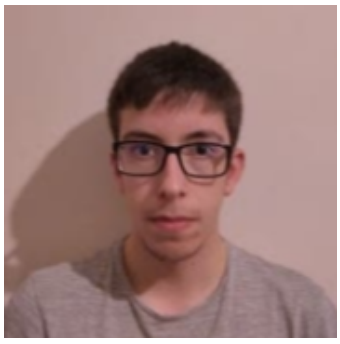
Universidade do Minho

Licenciatura em Engenharia Informática

Programação Orientada aos Objetos

Trabalho Prático - Grupo 31



Diogo Marques - A100897

Duarte Leitão - A100550

Lingyun Zhu - A100820

14 de maio de 2023

Índice

Introdução

No âmbito da Unidade Curricular de Programação Orientada aos Objetos foi-nos proposto um trabalho prático que visava a interação de vários utilizadores com uma plataforma de compra e venda de artigos.

Tal como acontece na vida real, dentro de uma plataforma deste género um utilizador pode realizar determinadas operações, como ver o dinheiro que gastou, ou os produtos que comprou, no entanto muitas outras funcionalidades estão apenas acessíveis ao administrador do sistema, sendo que o contrário também se aplica, visto que por norma o administrador não tem a capacidade de comprar/vender artigos.

Tendo em conta que este sistema deve ser implementado com base nos princípios da programação orientada aos objetos (encapsulamento das estruturas de dados, compatibilidade de tipos...), torna-se fundamental criar uma hierarquia de classes bem organizada que permita o crescimento sustentável do projeto, até porque futuramente devem ser inseridos novos artigos sem que isso implique uma mudança da implementação já realizada.

Assim, ao longo deste relatório, pretendemos apresentar e justificar a hierarquia de classes que adotámos, bem como detalhes de arquitetura que nos parecem ser de extrema relevância.

Arquitetura MVC

O sistema que se pretende implementar apresenta várias funcionalidades distintas, como a leitura de *input*, a interpretação do mesmo, e acima de tudo, as alterações que este causa nas estruturas de dados centrais. Assim sendo, torna-se fundamental adotar um padrão arquitetural a fim de distribuir cada uma destas funcionalidades pelos vários módulos que este possua.

Tendo em conta que foi referenciado nas aulas teóricas, escolhemos o padrão do MVC (*Model-View-Controller*) como arquitetura do sistema, todavia a versão definida por nós permite a interação entre o *Model* e a *View*, visto que não faz muito sentido o *Controller* ser um intermediário nas situações em que se limita a passar instruções sem sequer as alterar.

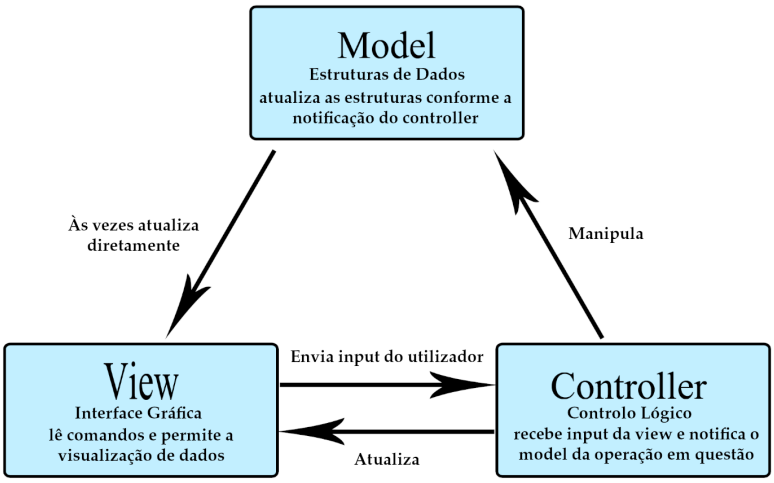


Figura 1. Arquitetura MVC

View

Um fator que julgamos ser bastante problemático diz respeito ao facto de a *View* ler o *input* e apresentar o *output*, pois sendo estes dois tão distintos um do outro não há qualquer razão para estarem incluídos no mesmo código, como tal decidimos dividir a *View* em duas classes designadas de *Leitor* e *Escritor,* que tal como o nome indica, são responsáveis por ler e escrever separadamente.

Perante isto, o *Leitor* comunica o *input* ao *Controlador,* enquanto que o *Escritor* recebe dados do *Controlador* e do *Gestor* (Modelo) tendo em vista a apresentação dos mesmos.

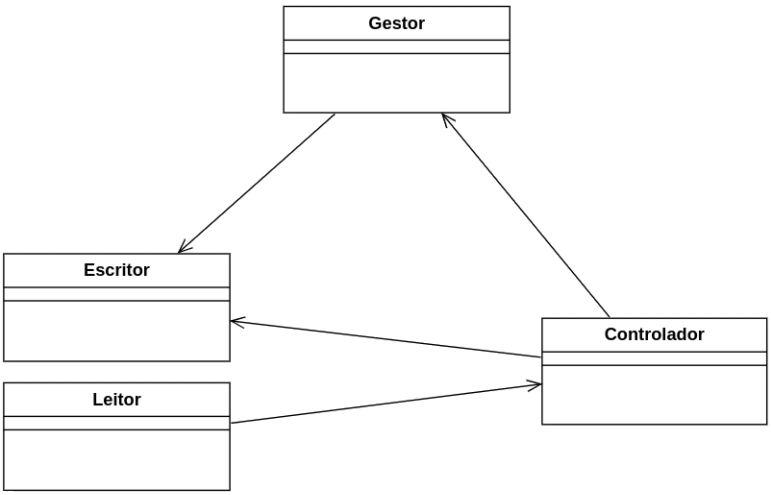


Figura 2. Padrão arquitetural final

Assim, com esta versão arquitetural fazemos uma clara distinção daquilo que são dados de *input/output*, e consequentemente o *Leitor* nem necessita de saber que o *Escritor* existe e vice-versa, permitindo uma maior independência entre classes.

Controller

Conforme vai recebendo dados do *Leitor,* o *Controlador* tem a capacidade de fazer uma interpretação dos mesmos e informar o *Gestor* de quais as operações que deve efetuar sobre as suas estruturas de dados, todavia o sistema dispõe de várias funcionalidades completamente distintas umas das outras, como a alteração de registos e cálculos estatísticos, portanto não há qualquer razão para o *Controlador* saber todas as ordens que deve dar, até porque isso iria originar uma classe extremamente grande e complexa.

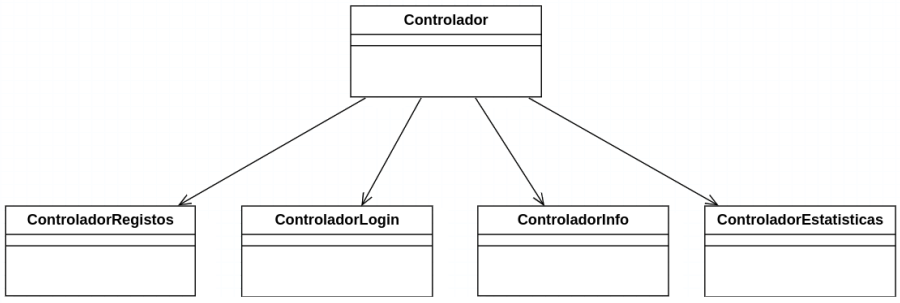


Figura 3. Divisão das funcionalidades por várias classes

Mais uma vez, a forma de resolver este problema passa pela distribuição e atribuição de tarefas a várias classes independentes entre si.

Controlador

Recebe uma mensagem do *Leitor* e a partir daí faz uma interpretação de qual a funcionalidade requerida pelo utilizador, de seguida essa mesma mensagem é reenviada, juntamente com uma *flag* a indicar a funcionalidade, para o respetivo controlador.

ControladorRegistos

Ao receber a mensagem e *flag* enviadas pelo *Controlador,* sabe de que forma deve notificar o *Gestor* sem que a mensagem tenha de voltar a ser interpretada, como se trata do controlador de registos, é natural que todas as suas notificações estejam relacionadas com a alteração das estruturas de dados presentes no *Gestor.*

ControladorLogin

Procede exatamente da mesma forma que o controlador anterior, contudo a notificação que envia diz respeito à autenticação de um utilizador a partir do seu *email.*

ControladorInfo

Notifica o *Gestor* relativamente a pedidos de informação, sendo que mais tarde essa mesma informação é fornecida pelo *Gestor* e reencaminhada para o *Escritor.*

ControladorEstatisticas

Notifica o *Gestor* relativamente a questões estatísticas, tais como a listagem dos maiores vendedores/compradores/transportadoras. Ao receber uma notificação deste género, o *Gestor* passa diretamente os dados ao *Escritor* sem que haja qualquer reencaminhamento por parte deste controlador.

Justificação

É certo que não era extremamente necessário criar novas classes para auxiliar o *Controlador,* podendo este tratar de tudo, todavia esta implementação permite que o *Controlador* apenas interprete mensagens enquanto que os restantes comunicam diretamente com o *Gestor.*

Assim, uma vez que o *Controlador* não conhece o *Gestor,* este funciona como uma espécie de fachada, visto que qualquer alteração dos demais controladores no máximo reflete-se no *Controlador.*

Model

É dentro desta secção que estão incluídas as estruturas de dados que suportam por completo o correto funcionamento do sistema, como tal é aqui que a classe *Gestor* opera conforme as notificações que recebe dos controladores.

Uma vez que o *Gestor* tem de salvaguardar os registos de objetos como utilizadores e artigos, este deve possuir estruturas de dados que permitam o acesso direto a um determinado objeto. Contudo, mais uma vez, não é correto implementar um *Gestor* que possua coleções de objetos tão distintos.

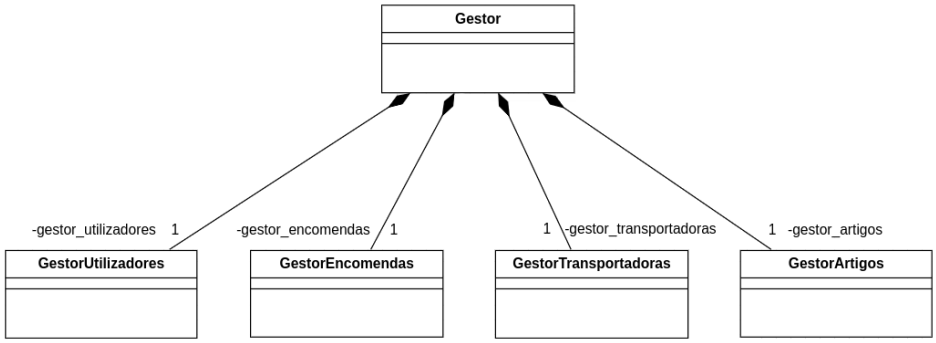


Figura 4. Distribuição das estruturas de dados pelas classes

Ao criar novas classes de gestores, nas quais cada uma controla e manipula as coleções referentes a um determinado objeto, o *Gestor* pode alterar indiretamente os registos de qualquer coleção, bastando apenas invocar métodos dos demais gestores.

Assim, tal como no caso anterior, o *Gestor* funciona como uma fachada, visto que as alterações nos gestores propagam-se até ao *Gestor* no prior dos casos.

Coleções

Tendo em vista a máxima rapidez no acesso a dados por parte dos utilizadores do sistema, deve ser dada relevância à forma como os registos são armazenados, visto que estes devem poder ser consultados em tempo constante.

A forma mais fácil de garantir isto é obviamente através da utilização de *Map's,* contudo em alguns casos a sua utilização não é necessária, podendo mesmo ser utilizadas listas.

Ainda dentro das coleções, podemos adotar uma estratégia de agregação ou composição, e tendo em conta que a composição é em certa medida mais segura, dado que implica a criação de cópias, optámos por seguir essa abordagem, mesmo sabendo que por outro lado está a ser desperdiçada memória que pode vir a ser útil no futuro.

Utilizador

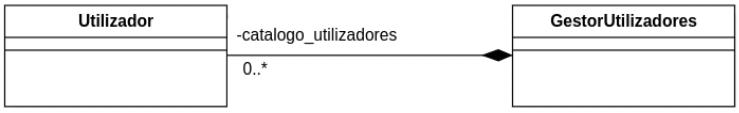


Figura 5. Coleção de Utilizadores

A coleção de utilizadores é o único caso em que não é utilizado um *Map,* mas sim uma lista, pois tendo em conta que os utilizadores são identificados por um número, podemos associar esse mesmo número ao índice da sua possui na lista, garantido assim o acesso direto.

Encomenda

Figura 6. Coleção de Encomendas

Tal como os utilizadores, as encomendas também são identificadas por um número, portanto seria admissível utilizar a mesma estratégia, contudo optámos por utilizar um *Map* sem nenhuma razão em específico.

Transportadora

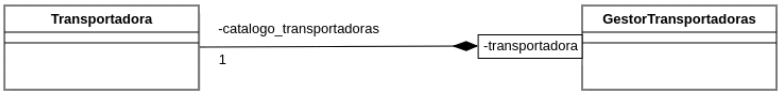


Figura 7. Coleção de Transportadoras

Ao contrário dos casos anteriores, em que o objeto era identificado univocamente por um inteiro, neste caso uma transportadora tem associada a si um nome que a caracteriza, como tal cada objeto desta classe é guardado num *Map* cuja chave é o *hashCode* do nome da transportadora.

Artigo

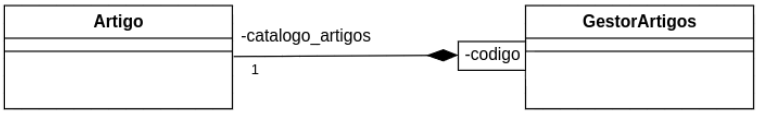


Figura 8. Coleção de Artigos

A exemplo de uma transportadora, um artigo é caracterizado por um código alfanumérico, ou seja, uma *String*, portanto a coleção implementada é a mesma.

Fatura

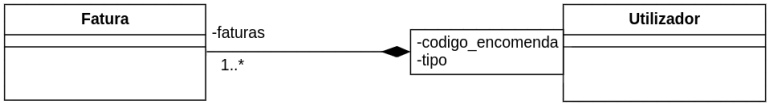


Figura 9. Coleção de Faturas

A criação de uma classe fatura a princípio não parece óbvia, contudo se queremos saber o dinheiro ganho/gasto por um determinado utilizador, convém ter esses valores previamente calculados a fim de responder rapidamente às interrogações estatísticas.

Como tal, quando uma encomenda é finalizada, o utilizador recebe uma fatura por cada artigo que esta possui, e assim, à medida que as faturas são inserida no *Map,* apenas uma é preservada, sendo que o preço desta corresponde ao somatório do preço das restantes.

Além disso, há que ter em atenção que numa encomenda existem dois lados, o de quem compra, e o de quem vende, como tal a chave de uma entrada do *Map* não é somente o código da encomenda, mas sim uma combinação entre este e o tipo, que indica se um determinado utilizador age como vendedor ou comprador perante uma encomenda, podendo inclusive agir nos dois papéis em simultâneo.

Hierarquia de Classes

Um dos pontos fortes da programação orienta aos objetos diz respeito à compatibilidade de tipos, ou seja, para além de uma classe ser compatível com ela própria, é também compatível com outra qualquer, o que consequentemente permite uma certa abstração do tipo de dados.

Tendo em conta que no futuro é expectável inserir novos tipos de artigos, podemos muito facilmente identificar uma pequena hierarquia de classes, na qual um *Artigo* representa a superclasse e as suas especializações (*Sapatilha, Mala* e *Tshirt*) as respetivas subclasses, desta forma os artigos são tratados sem distinção pelos gestores, não havendo por isso a necessidade que estes conheçam em específico os dados que estão a manusear.

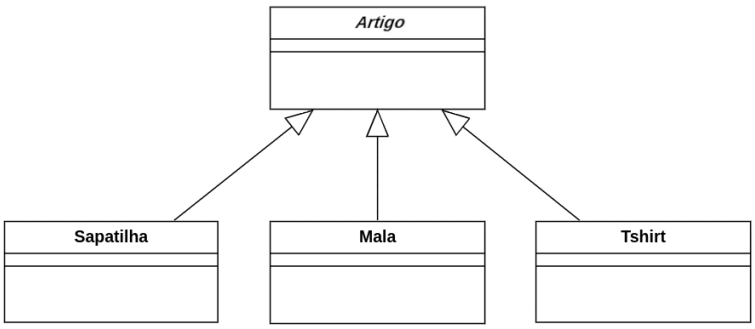


Figura 10. Hierarquia de classes

Uma vez que cada tipo de artigo possui uma fórmula de cálculo diferente, é necessário obrigar cada uma das subclasses a implementar um método conforme essas fórmulas, assim sendo é possível definir que a classe *Artigo* é abstrata e que o método *calculaPreço* também o é, pois consequentemente as subclasses são obrigadas a implementarem esse método. Desta forma criámos um certo polimorfismo, visto que ao invocar o método *calculaPreço* na classe *Artigo*, são utilizadas diferentes fórmulas sem que de facto nos apercebamos disso.

Além disso, esta noção de hierarquia também apresenta várias vantagens ao nível da codificação, visto que as variáveis e métodos da classe *Artigo* são herdados pelas subclasses, permitindo assim uma reutilização do código.

Noção de Premium

O sistema deve ainda prever a existência de artigos e transportadoras *premium,* o que a princípio parece ser facilmente resolvido através da implementação de uma interface, contudo não foi esta a decisão que tomámos.

Quando dizemos que uma classe implementa uma determinada interface, estamos na verdade a dizer que essa classe segue um dado comportamento, assim sendo não faz qualquer sentido que a classe *Artigo* ou *Transportadora* implementem a interface *Premium,* visto que existem artigos e transportadoras que não são *premium*, e portanto não seguem esse comportamento já predefinido.

Uma eventual forma de contornar este problema passaria pela criação de classes que especializassem *Transportadora/Artigo* e fossem compatíveis com *premium*, ou seja, criar algo como *TransportadoraPremium, MalaPremium, SapatilhaPremium e TshirtPremium,* visto que desta forma as classes poderiam implementar *Premium* sem qualquer problema de compatibilidade.

Todavia esta solução parece um pouco rebuscada, e portanto resolvemos este problema da forma que nos pareceu mais adequada, ou seja, com a inserção de uma variável de instância (booleano) que define se um objeto segue a noção de *premium.*

Hierarquia de Interfaces

Aquando do início da construção do sistema, não estávamos a prever a implementação de qualquer interface, todavia na fase final deparámo-nos com um grande problema relativamente ao envio de dados para o *Escritor*, pois estando este fora do Modelo, não há qualquer razão para que conheça de facto os tipos de dados que vai apresentar.

De forma a corrigir isso, é necessário criar um tipo de dados que o *Escritor* conheça (sabe que métodos pode invocar), e que ao mesmo tempo seja compatível com as classes do Modelo.

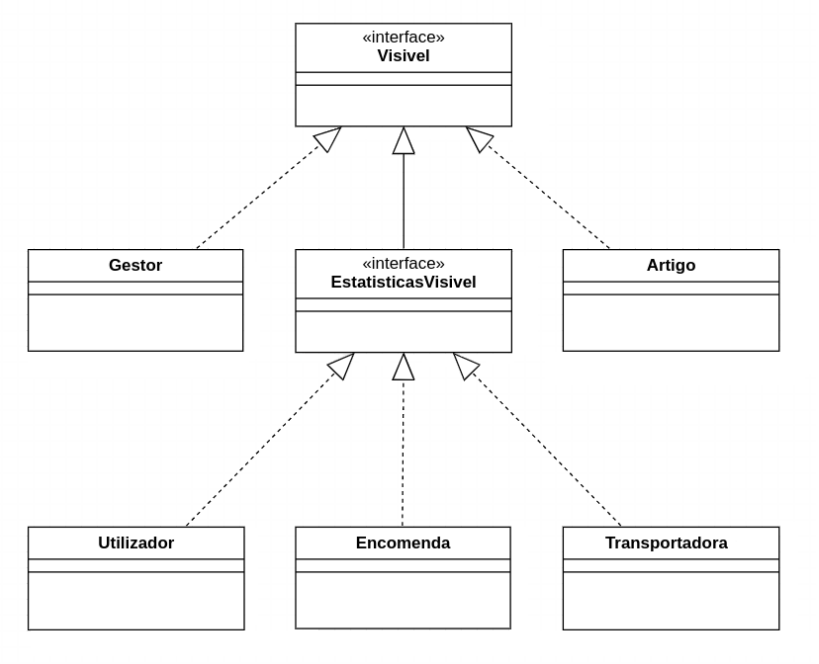


Figura 11. Hierarquia de Interfaces

Após muito ponderarmos, chegámos à noção de uma interface *Visivel*, ou seja, todos os objetos que implementem esta interface são reconhecidos pelo *Escritor*, e portanto podem ser apresentados sem que este saiba que classes estão implícitas.

Contudo existem classes que, para além de serem "visiveis", têm de apresentar estatísticas, e portanto é necessário criar uma outra interface que estenda a noção de *Visivel* e permita a apresentação dessas mesmas estatísticas, daí a criação da interface *EstatisticasVisivel.*

Interpretação do Enunciado

Ao longo do desenvolvimento deste trabalho prático foram sendo levantadas imensas dúvidas relativamente a requisitos do sistema, e tendo em conta que nem sempre as respostas obtidas pelos docentes foram convergentes, decidimos nós próprios definir esses mesmos requisitos.

- A comissão que a *Vintage* aplica só tem impacto para o vendedor, enquanto que o comprador paga o preço dos artigos na sua totalidade.
- Os preços de transporte não são assegurados pelos utilizadores, visto que as transportadoras têm uma parceria com a *Vintage,* e como tal o dinheiro angariado com as comissões serve, inclusive, para pagar os custos de transporte.
- Quando um artigo é inserido numa encomenda, este deixa de estar disponível em *stock*, no entanto não é dado como vendido, visto que a encomenda ainda está no estado *pendente.*
- No momento em que uma encomenda é finalizada, os artigos contidos nesta são dados como vendidos/adquiridos, consequentemente são passadas as respetivas faturas aos compradores/vendedores.
- Um utilizador pode comprar os seus próprios artigos e ter várias encomendas no estado *pendente/finalizada/expedida*.
- Ao fim de 48 horas, as encomendas finalizadas são expedidas pelas transportadoras e o utilizador dispõem de um prazo de 48 horas para realizar a devolução.
- Quando uma encomenda é devolvida, o registo desta é eliminado juntamente com todas as movimentações de dinheiro que esta gerou, excetuando o caso das transportadoras, que uma vez efetuado o serviço, não devem ver o dinheiro cobrado ser devolvido.
- Depois de uma encomenda ser devolvida, os artigos que esta continha passam para o *stock,* sendo a sua compra novamente possível.
- O código de um artigo é definido pelo utilizador, pois ao serem gerados códigos aleatórios pelo sistema, torna-se praticamente impossível realizar testes unitários sobre o *stock* de artigos.

Funcionalidades

- Inserir utilizadores ⇒ Administrador do sistema
- Inserir transportadoras ⇒ Administrador do sistema
- Inserir artigos no *stock* ⇒ Utilizador
- Remover artigos do *stock* ⇒ Utilizador
- Criar encomendas ⇒ Utilizador
- Inserir artigos numa encomenda ⇒ Utilizador
- Remover artigos de uma encomenda ⇒ Utilizador
- Finalizar encomendas ⇒ Utilizador
- Expedir encomendas ⇒ Automático
- Devolver encomendas ⇒ Utilizador
- Alterar taxas das transportadoras ⇒ Administrador do sistema
- Alterar comissão da *Vintage* ⇒ Administrador do sistema
- Alterar preços de artigos ⇒ Utilizador
- Alterar a data ⇒ Administrador do sistema
- *Login* ⇒ Utilizador
- *Logout* ⇒ Utilizador
- Visualizar a data ⇒ Qualquer um
- Visualizar os catálogos ⇒ Qualquer um
- Visualizar a comissão da *Vintage* ⇒ Qualquer um
- Conferir o lucro da *Vintage* ⇒ Qualquer um
- Identificar as encomendas emitidas por um certo vendedor ⇒ Qualquer um
- Identificar os maiores vendedores/compradores/transportadoras ⇒ Qualquer um

Conclusão

Chegados a este ponto, pensamos ter cumprido praticamente na totalidade com os princípios da programação orientada aos objetos (encapsulamento, abstração do tipo de dados, polimorfismo...), além disso foram implementadas todas as funcionalidades necessárias ao bom funcionamento do sistema, bem como outras que são meramente estatísticas.

Assim, tendo em conta estes pontos, julgamos ter realizado um bom trabalho, até porque a qualquer momento é possível inserir novos tipos de artigos sem que isso gere alterações do código de outra classes.

Em suma, este trabalho prático serviu essencialmente para melhorarmos as nossas capacidades relativamente a uma linguagem que até há pouco tempo atrás desconhecíamos, e perceber o quão relevante é o paradigma dos objetos.