Ciências ULisboa
Faculdade de Ciências da Universidade de Lisboa

2019/2020

Construção de Sistemas de Software

Padrões para a Camada de Negócio

Organização em Camadas

- Vimos que no caso das EAs as camadas principais são a **Apresentação**, o **Negócio** e de **Dados**.
- Existem várias formas padronizadas de implementar as responsabilidades de cada uma destas camadas
- Estes fazem parte do catálogo de ***Patterns of Enterprise Application Architecture***

https://martinfowler.com/eaaCatalog/

Padrões para as várias camadas

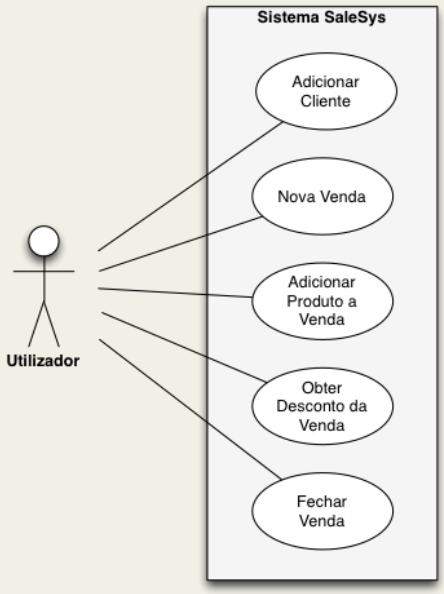
Apresentação	MVC Front controller Page controller Page template
Negócio	Transaction Script Table Module Domain Model
Dados	Row data gateway Table data gateway Active Record Data Mapper

SaleSys

- É um aplicação fictícia, muito simples, que é usada na disciplina ao longo do semestre para ilustrar os conceitos e as técnicas que vão sendo apresentadas

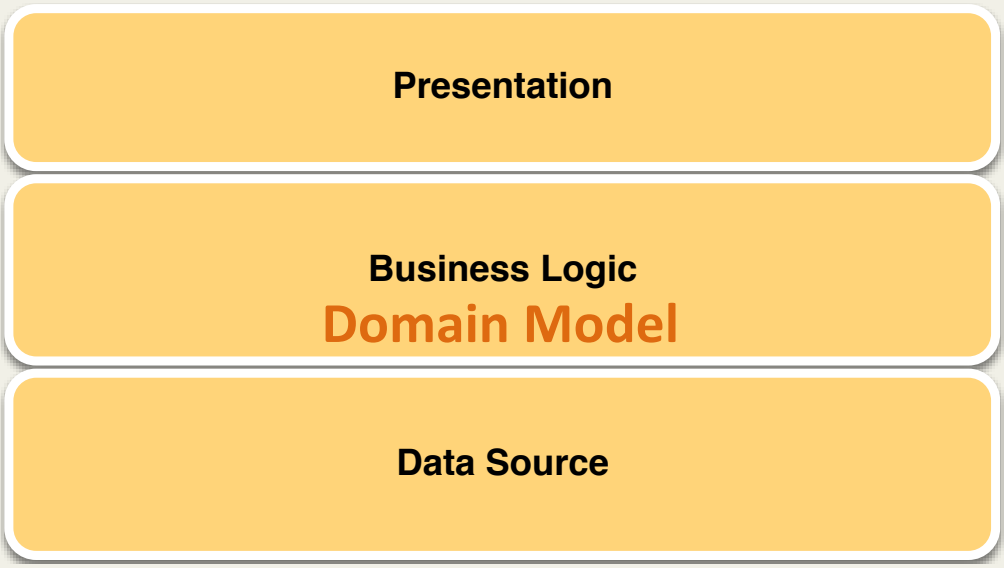
- O propósito da aplicação é fazer a gestão da carteira de clientes e de vendas de uma empresa
- Casos de uso:
  - Adicionar **cliente**
  - Criar **venda**
  - Acrescentar **produtos** a uma venda em aberto
  - Obter valor de **desconto** da venda
  - Concluir venda

Diagrama de Casos de Uso



Padrões para as várias camadas

Apresentação	MVC Front controller Page controller Page template
Negócio	Transaction Script Table Module Domain Model
Dados	Row data gateway Table data gateway Active Record Data Mapper



Domain Model

- O código da camada de negócio é organizado **em torno de** classes que são inspiradas pelos
  - **conceitos do domínio**,
  - nas suas **características** e
  - **associações**
- Segue uma abordagem de desenho orientado a objetos, como aprenderam em DCO
- É adequado para sistemas cuja lógica de domínio tem um grau de complexidade elevado

Domain Model

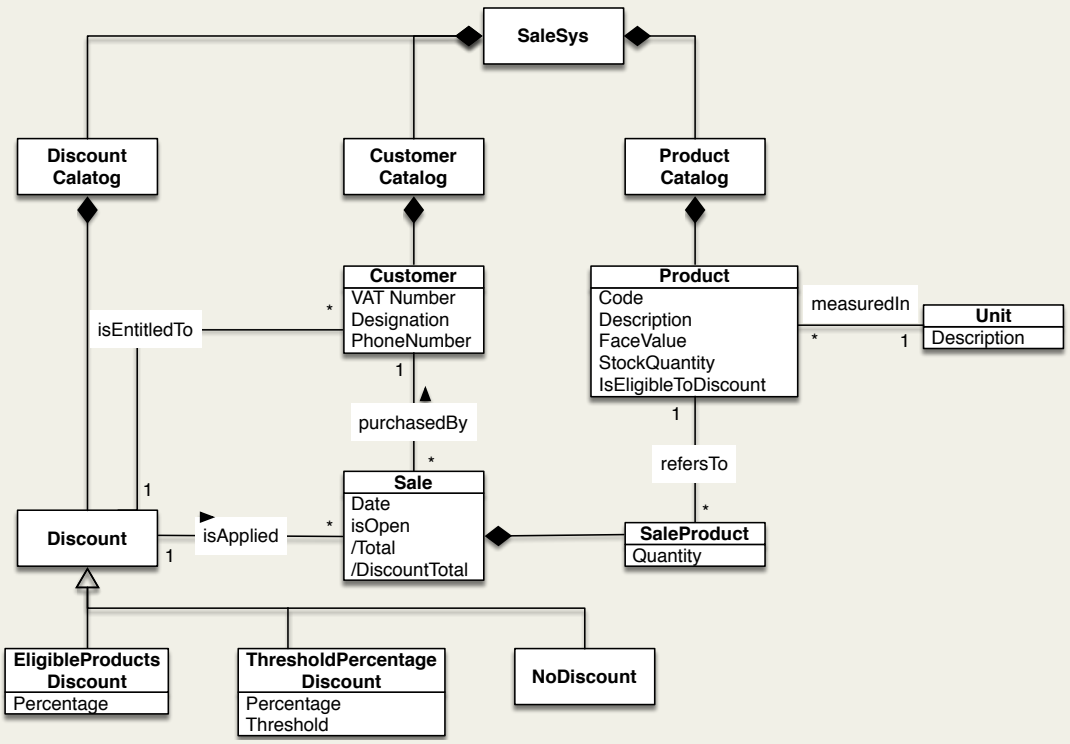
Vantagens

- Todas as que o OOP tem sobre a programação procedimental
  - Dispõe de primitivas poderosas que nos ajudam a lidar com a complexidade
  - Fácil de acomodar alterações futuras
  - …

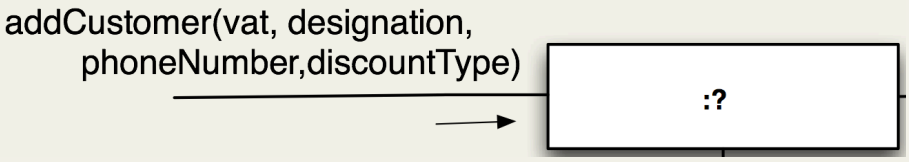
Desvantagens

- Todas as que o OOP tem sobre a programação procedimental
  - Requer uma equipa proficiente em OO
  - Desenho mais difícil de conceber à partida
  - Não tão adequado para sistemas simples, dado o esforço inicial de análise e desenho (a menos que a equipa seja perita em OO)
- A persistência de informação em base de dados relacional é muito mais difícil

SaleSys: Modelo de Domínio

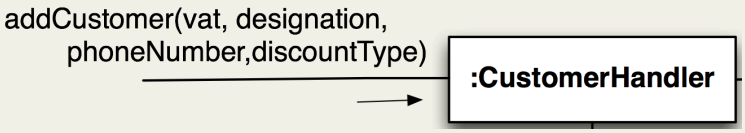


Exemplo: Adicionar Cliente



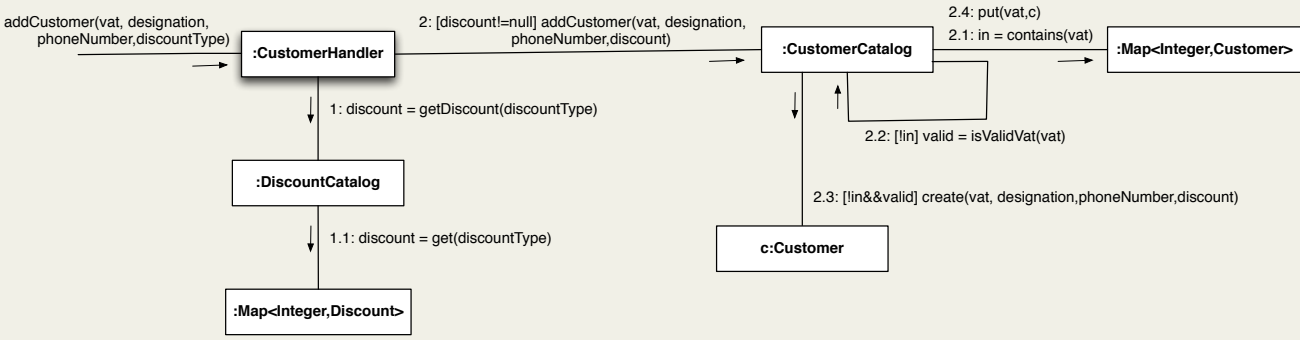
Se o código do tipo de desconto é válido
e o vat é válido
e o nome e o telefone estão preenchidos
adicionar o cliente

Exemplo: Adicionar Cliente

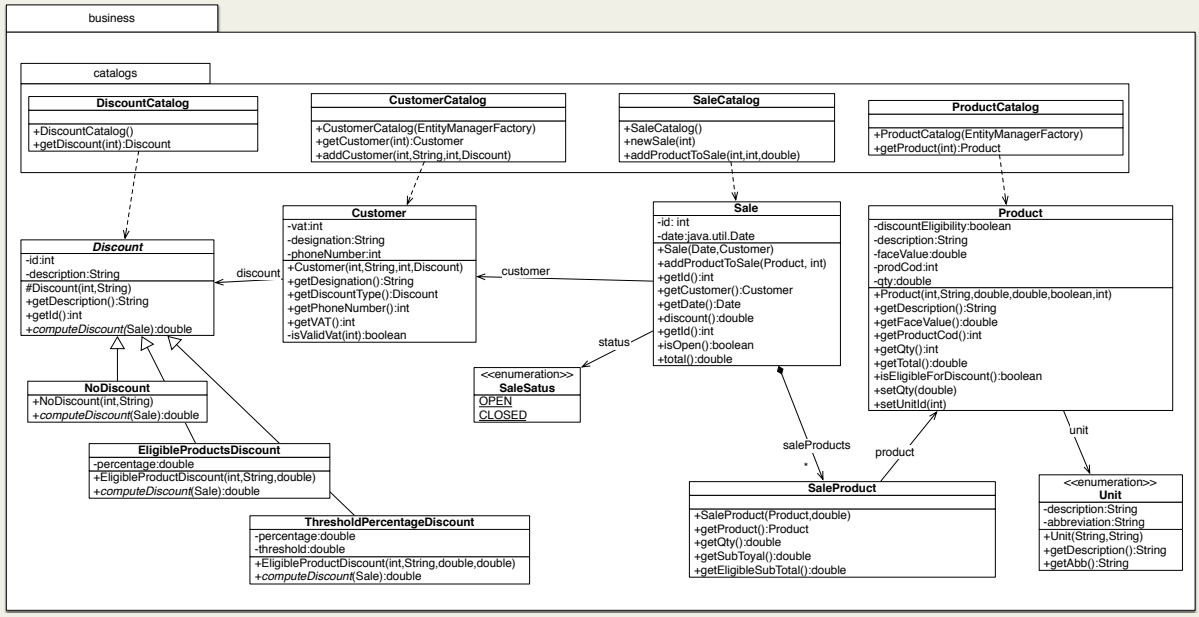


- Verificar se código do tipo de desconto é válido: **DiscountCatalog**
- Adicionar o cliente, fazendo verificações necessárias relativas à info dada: **CustomerCatalog**
  - Verificar se o vat é válido
  - Verificar se o nome e o telefone estão preenchidos

Exemplo: Adicionar Cliente



SaleSys: Domain Model & Data-In-Memory (No Persistence)



Esboço de Implementação: Adicionar Cliente

AddCustomerHandler

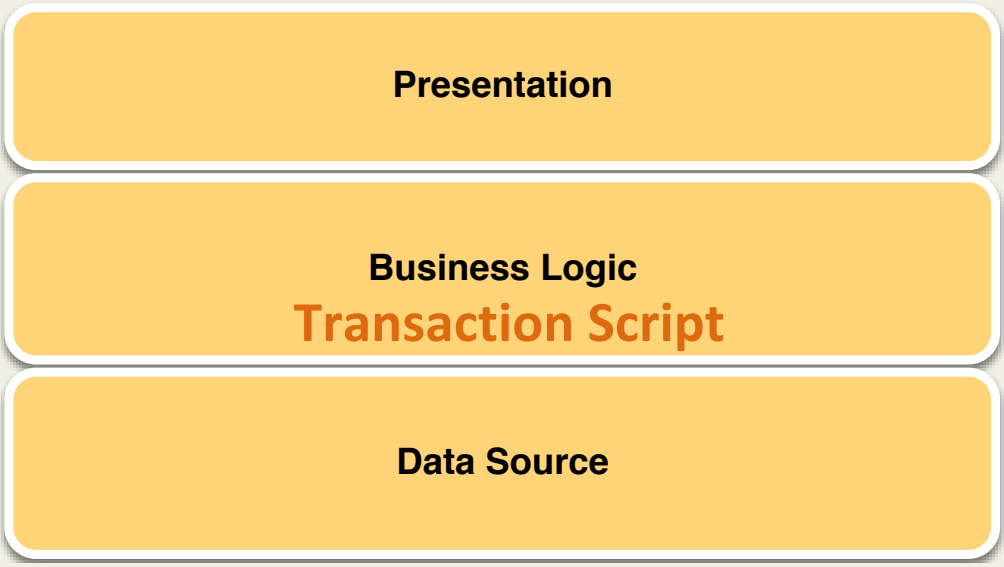
```
public void addCustomer (int vat, String denomination,
        int phoneNumber, int discountType)
        throws ApplicationException {
    Discount discount = discountCatalog.getDiscount(discountType);
    customerCatalog.addCustomer(vat, denomination, phoneNumber, discount);
}
```

CustomerCatalog

```
public void addCustomer (int vat, String designation, int phoneNumber, Discount discountType)
        throws ApplicationException {
    if (!Customer.isValidVAT(vat))
        throw new ApplicationException ("Invalid VAT number");
    if (customers.containsKey(vat))
        throw new ApplicationException ("Customer already exists");
    customers.put(vat, new Customer(vat, designation, phoneNumber, discountType));
}
```

DiscountCatalog

```
public Discount getDiscount (int discountType) throws ApplicationException {
    Discount d = discounts.get(discountType);
    if (d == null)
        throw new ApplicationException ("Discount type " + discountType + " not found.");
    return d;
}
```

Transaction Script

- O código da camada de negócio é organizado em torno das operações do sistema, seguindo uma abordagem procedimental
- Um **procedimento** (*script)* por **transação de negócio** que contém todos os passos necessários à sua execução
  1. recebe o input da camada de apresentação
  2. processa esse input fazendo validações e cálculos
  3. guarda os dados a persistir na base de dados e possivelmente invoca operações de sistemas externos
- Apropriado quando a **lógica do domínio é simples**
  - o que não é o mesmo que aplicações pequenas...
  - eg., aplicações CRUD (Create, Read, Update and Delete)

Transaction Script

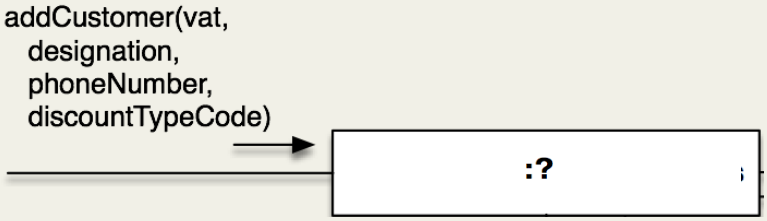
Vantagens

- Fácil de entender
- Fácil de implementar
- O código de cada operação encontra-se todo no mesmo local (unidade de codificação)
- A persistência de informação é, em regra geral, simples de efetuar

Desvantagens

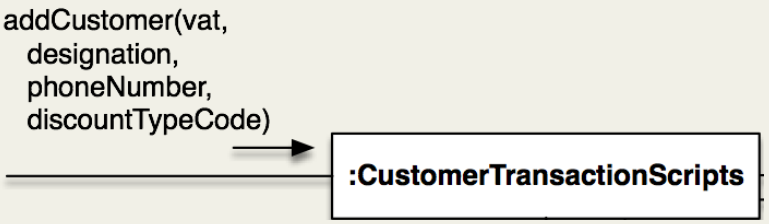
- Há tendência para que exista código duplicado
- Não permite tirar partido de primitivas poderosas como a herança e polimorfismo
- Muito difícil de aplicar quando a lógica do domínio é complexa

Exemplo: Adicionar Cliente



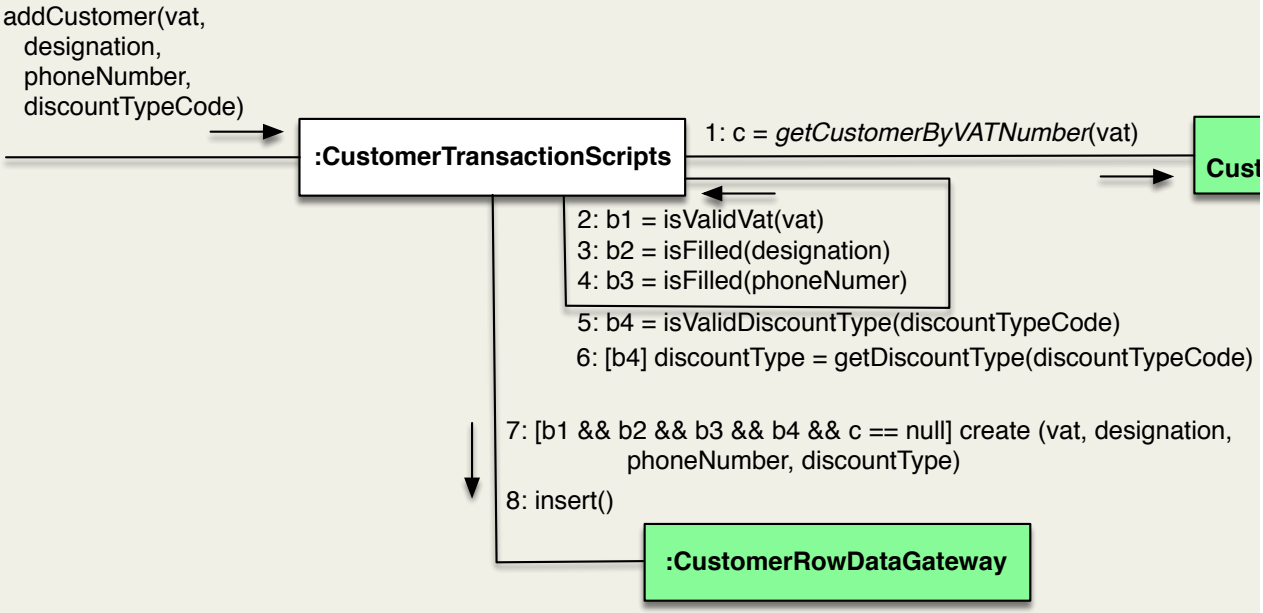
Se o código do tipo de desconto é válido
e o vat é válido
e o nome e o telefone estão preenchidos
adicionar o cliente

Exemplo: Adicionar Cliente



- *script* trata da validação
  - Verificar se código do tipo de desconto é válido
  - Verificar se o vat é válido
  - Verificar se o nome e o telefone estão preenchidos
- *script* faz todo o processamento
  - Recorrendo aos serviços prestados pela camada de dados, guardar registo com novo cliente na base de dados

Exemplo: Adicionar Cliente

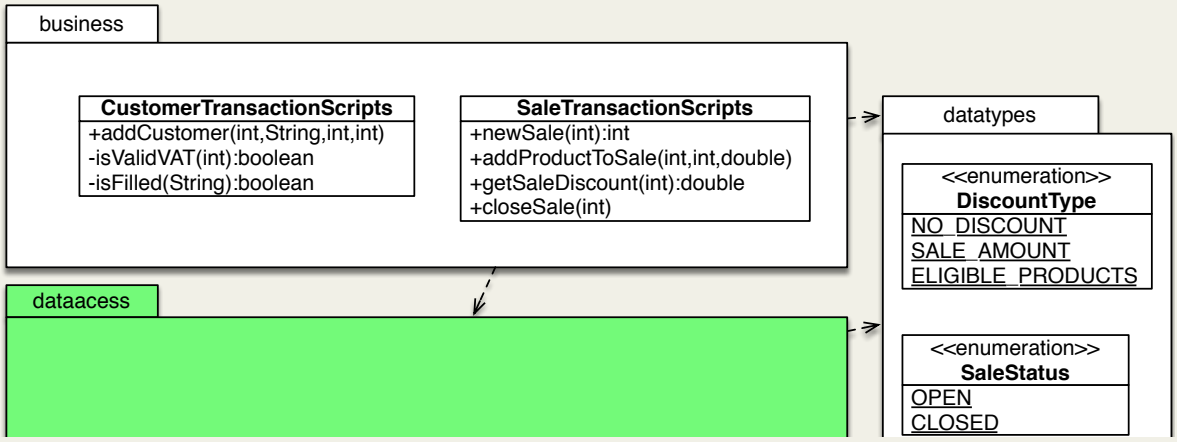


Elementos da camada de dados

Esboço de implementação: adicionar cliente

```
public void addCustomer(int vat, String denomination, int phoneNumber, int discountCode)
        throws ApplicationException {
    // Checks if it is a valid VAT number
    if (!isValidVAT(vat))
        throw new ApplicationException("Invalid VAT number: " + vat);

    // Checks if the discount code is valid.
    if (discountCode <= 0 || discountCode > DiscountType.values().length)
        throw new ApplicationException ("Invalid Discount Code: " + discountCode);

    // Checks that denomination and phoneNumber and filled in
    if (!isFilled(denomination) || phoneNumber == 0)
        throw new ApplicationException(
                "Both denomination and phoneNumber must be filled");

    /*
    Tries to stores the customer in the database. There is a
```

SaleSys: Transaction Script



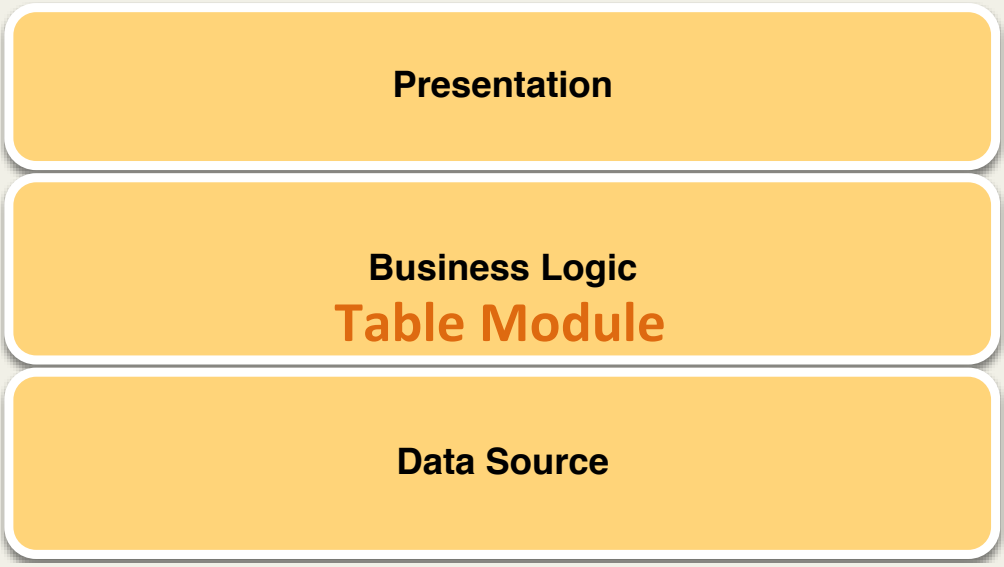


Table Module

- O código da camada de negócio é organizado **em torno das tabelas** usadas para persistir os dados e trabalha unicamente com dados tabulares na forma de um **conjunto de registos**

- Mais especificamente,
  - tem-se uma classe por cada tabela (ou vista), com todos métodos para atuar ou fazer cálculos sobre os dados que essa tabela armazena
  - qualquer outra classe da camada de negócio que precise de fazer ou saber alguma coisa sobre os dados desta tabela tem de usar um objeto do respetivo módulo
  - usa-se um único objeto desta classe para lidar com toda a lógica de domínio associada aos dados guardados nessa tabela

Table Module

- É uma proposta que endereça os problemas mais prementes dos padrões *Transaction Script* e *Domain Model*
  - Repetição de código
  - Dificuldade inicial em desenvolver o sistema
  - Dificuldade em persistir informação

- Os dados tabulares na forma de um **conjunto de registos (*Record Set/Table Data*)** podem ser obtidos por uma camada fininha que trata do acesso aos dados, tipicamente através de interrogações SQL à base de dados

Table Module

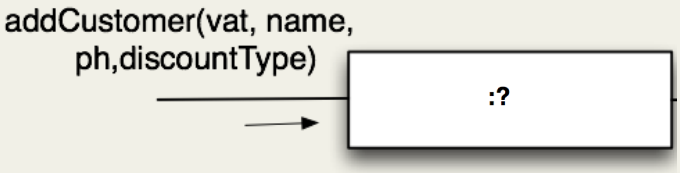
Vantagens

- Cada módulo corresponde a uma tabela ou vista da base de dados e por isso é fácil persistir
- Como o código se encontra fatorizado pelos módulos anula-se a repetição de código
- Integra especialmente bem com as ferramentas da Microsoft: *Visual Studio* e linguagens (e.g., .NET, C#)

Desvantagens

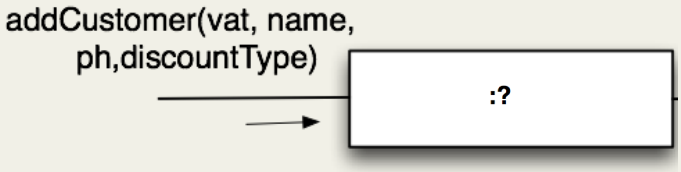
- Os módulos são diferentes dos objetos, em particular não têm identidade
- As primitivas poderosas disponíveis no OOP têm uma aplicação limitada
  - Polimorfismo
  - Herança

Exemplo: Adicionar Cliente



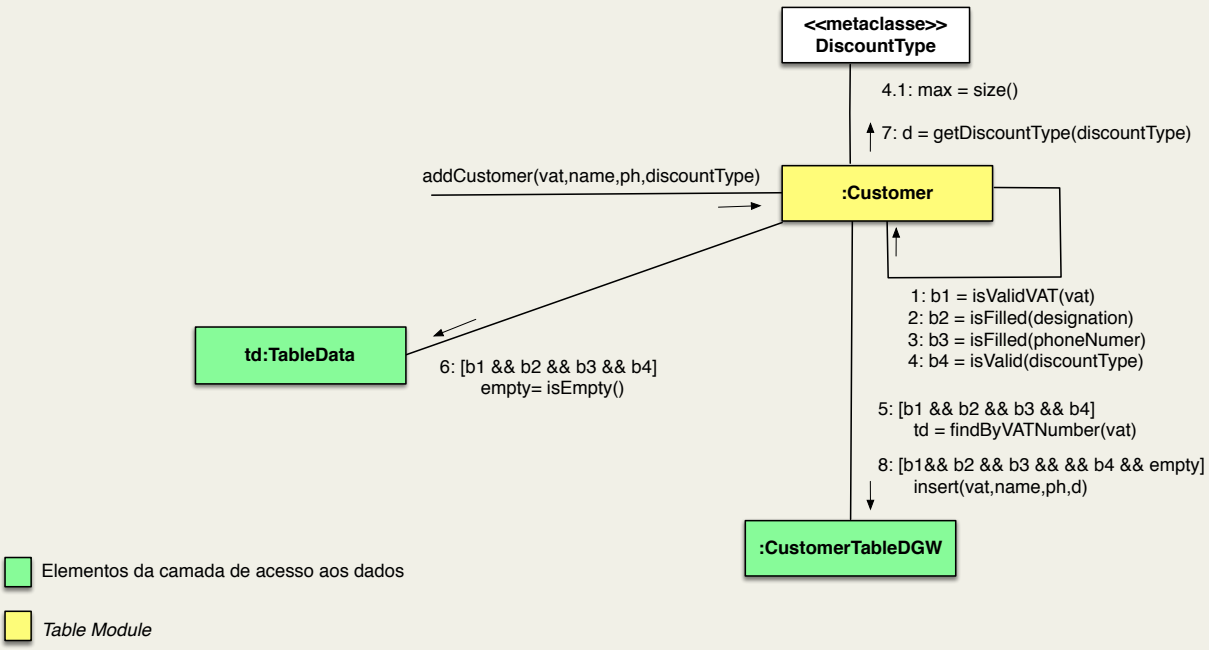
Se o código do tipo de desconto é válido
    e o vat é válido
    e o nome e o telefone estão preenchidos
adicionar o cliente

Exemplo: Adicionar Cliente



- Responsabilidades estão dependentes de dados guardados na tabela *Customer* pelo que são atribuídas ao *Customer Table Module*
  - Verificar se código do tipo de desconto é válido
  - Verificar se o vat é válido e se o nome e o telefone estão preenchidos
  - Recorrendo aos serviços prestados pela camada de dados, guardar registo com novo cliente na base de dados

Exemplo: Adicionar Cliente

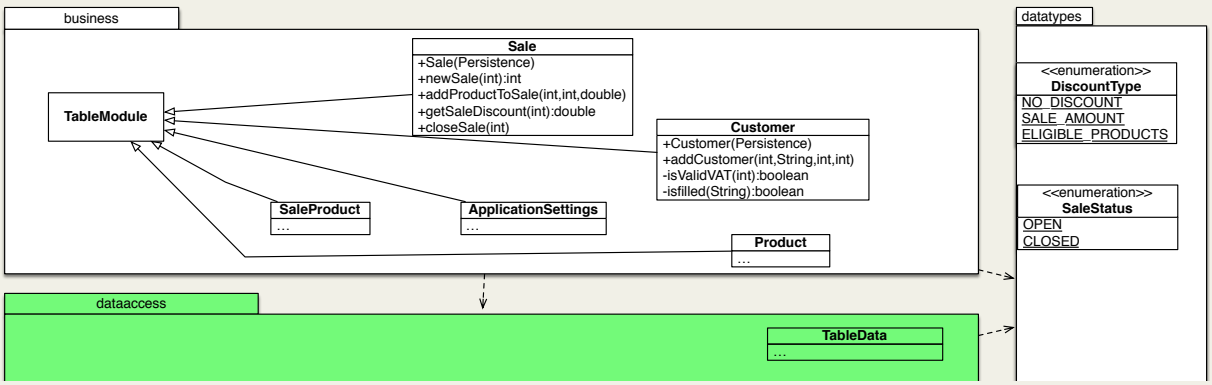


Esboço de Implementação: Adicionar Cliente

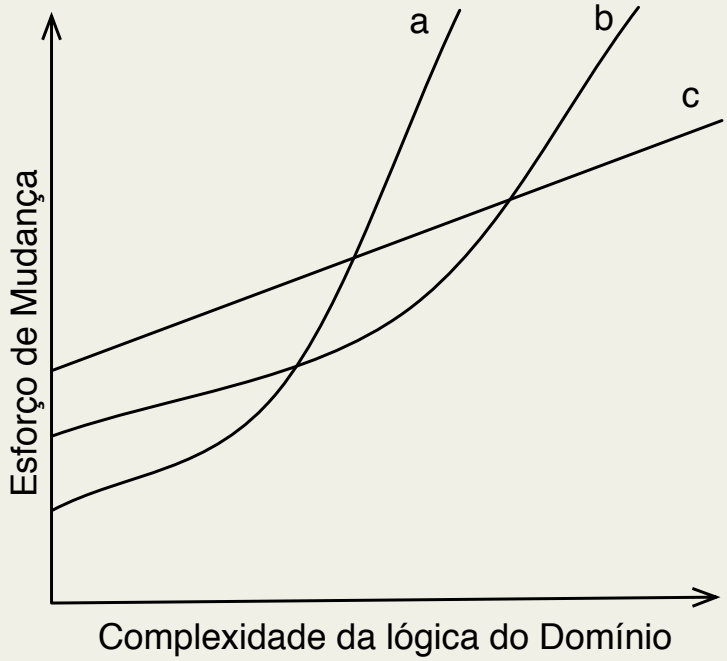
Customer (Table Module)

```
public void addCustomer (int vat, String designation, int phoneNumber,
        int discountCode) throws ApplicationException {
    // Checks if it is a valid VAT number
    if (!isValidVAT (vat))
        throw new ApplicationException ("Invalid VAT number: " + vat);

    // Checks if the discount code is valid.
    if (discountCode <= 0 || discountCode > DiscountType.values().length)
        throw new ApplicationException ("Invalid Discount Code: " + discountCode);

    // Checks that there is no other customer with the same VAT number
    if (existsCustomer(vat))
        throw new ApplicationException ("Customer with VAT number " + vat + " already exists!");

    try {
        // If the customer does not exists, add it to the database
        table.insert(vat, designation, phoneNumber, DiscountType.values()[discountCode-1]);
    } catch (PersistenceException e) {
        throw new ApplicationException ("Internal error adding a custumer", e);
    }
}
```


SaleSys: Table Module



Escolha de um padrão

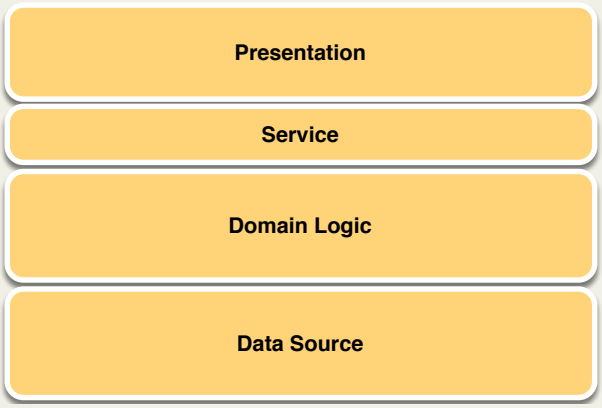


Como escolher?

- É preciso em primeiro lugar perceber a **complexidade da lógica de negócio *(domain logic)***.
  - Se a complexidade é baixa, a escolha óbvia é o *Transaction Script*.
  - Há medida que a complexidade aumenta, o custo que advém da utilização do *Transaction Script* ou do *Table Module* torna-se muito elevado — a adição de novas *features* torna-se exponencialmente mais difícil.
- E como avaliar a complexidade da lógica do negócio?
  - Uma tarefa difícil que deve ser realizada por alguém com experiência, depois de fazer uma análise inicial dos requisitos do sistema
- Outros fatores que podem pesar na escolha
  - a familiaridade da equipa com o *Domain Model* (e a sua qualidade)
  - o apoio do ambiente de desenvolvimento usado aos *Record Sets (eg., .NET)*

Camada de Serviços

- Quando se usa o padrão *Domain Model* ou *Table Module* é comum a divisão da **Camada de Negócio** em duas
  - **Camada de Serviços (*Service Layer*)**
    - responsável pela API da aplicação (e por isso também chamado ***Application Layer***)
  - **Camada de Domínio (*Domain Logic Layer*)**
    - responsável por tudo o que se relaciona com o domínio



Camada de Serviços

- **Camada de Serviços (*Service Layer*)**
  - responsável pela API da aplicação
  - define a lógica da aplicação, ou seja, quais são as responsabilidades da aplicação
  - em geral apenas uma **fachada** para a camada de apresentação, não definindo nada relacionado com a lógica de negócio
  - define a possibilidade de invocações remotas
  - também é onde é potencialmente colocado o controlo transacional e de segurança
- **Camada de Domínio (*Domain Logic Layer*)**
  - responsável por tudo o que se relaciona com o domínio do problema
  - concretiza os serviços colocados na camada acima

Quando se usa?

- A camada de serviços em geral não é necessária quando há apenas um tipo de cliente (por exemplo um UI) e o processamento inerente aos casos de uso não envolvem múltiplos recursos transacionais

- Em geral, quando é usado o padrão ***Transaction Script***, sendo a complexidade da lógica do domínio da aplicação baixa, não se justifica a divisão da camada de negócio.
- Existe então apenas uma camada e é comum essa ser referida como a ***Service Layer*** (porque *script* também referido como *service* ou *use-case controller*)
- Neste caso a implementação da camada consiste num conjunto de classes gordas (com os *transaction scripts)* que diretamente implementam a lógica da aplicação

Sumário

- Três padrões para a organização da camada de negócio: *domain model, transaction script* e *table module*.
- Discussão das vantagens e desvantagens da utilização de cada um dos padrões e eventuais restrições que colocam na forma como é realizado o acesso à informação.
- Identificação das situações em que se deve utilizar cada uma das três formas de organização da camada de negócio.
- A arquitetura em camadas com mais uma camada — Camada de Serviços
- Os vários padrões para organização da Camada de Serviços
- Ilustração com o exemplo do *SaleSys*