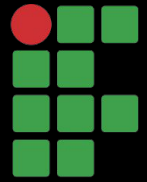
Instituto Federal da Paraíba - Campus Campina Grande
Engenharia de Computação



Projeto

Mobile Clean

Disciplina: Padrões de Projetos
Professor(orientador): Dr.Katyusco de Farias Santos
Alunos: Joab da Silva Maia,Edivam Enéas de Almeida Júnior

Projeto

- Nome: Mobile Cleaner
- Objetivo: O programa Mobile Cleaner tem como objetivo auxiliar um determinado setor da empresa responsável pela limpeza a controlar as limpezas dos celulares de todos os funcionários. Como uma forma de combate ao coronavírus, vírus esse que vem causando milhares de mortes diariamente no mundo inteiro. Obs: A ideia de criação do programa foi o combate ao coronavírus, porém por conta da limpeza ser algo extremamente importante para prevenção de várias doenças o programa tem uma característica abrangente, tendo em vista, que a higienização dos aparelhos celulares combate várias doenças, não apenas o coronavírus.

Funcionalidades do Projeto

- Login: Usuário
- Cadastro: Cliente, serviço, usuário
- Listagem: Cliente, serviço, usuário
- Agendamento: Do serviço a ser feito
- Exclusão: Cliente, serviço, usuário

Organização do sistema

Arquitetura utilizada: DDD + MVC

Por padrão a arquitetura sugere que se utilize 3 camadas.

3 camadas:

- Banco de dados: 5 padrões;
- Compartilhada: 2 padrões;
- Domínio: 4 padrões;

Padrões utilizados:

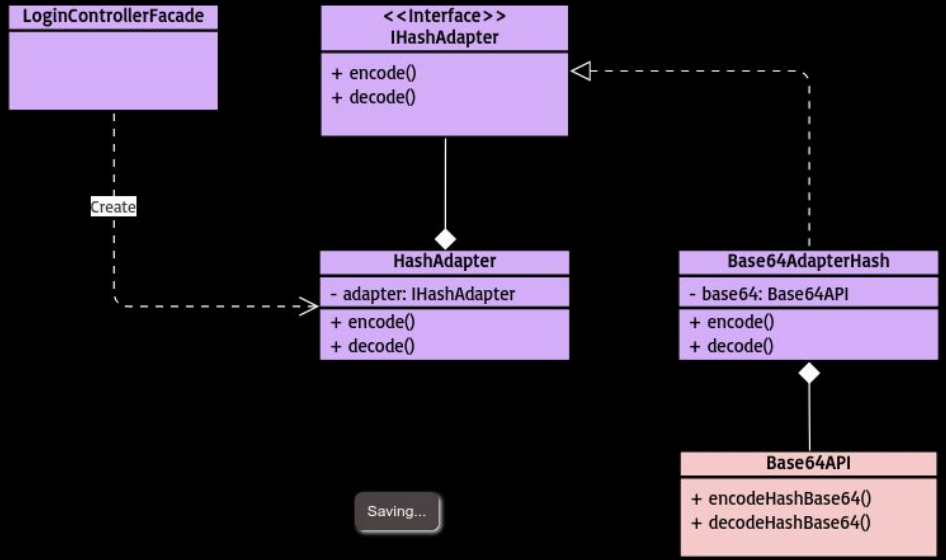
- 1 - Adapter
- 2 - Chain of responsibility
- 3 - Builder
- 4 - Strategy
- 5 - Proxy
- 6 - Singleton
- 7 - Facade
- 8 - Iterator

Conhecimentos adicionais de outros padrões :

- 9 - MVC
- 10 - DAO
- 11 - Helper
- 12 - Repository
- 13 - Static Factory Method

1. Adapter

Utilizado para encriptar e decriptar a senha do usuário.



Adapter

API

```java
import java.util.Base64;

public class Base64API {
    public String encodeHashBase64(String password) {
        return Base64.getEncoder().encodeToString(password.getBytes());
    }

    public String decodeHashBase64(String hashPassword) {
        byte[] decodedBytes = Base64.getDecoder().decode(hashPassword);
        return new String(Base64.getDecoder().decode(decodedBytes));
    }
}
```

Contrato

```java
public interface IHashAdapter {
    public String encode(String password);

    public String decode(String hashPassword);
}
```

Target

```java
public class HashAdapter implements IHashAdapter {
    private IHashAdapter _adapter;

    public HashAdapter(IHashAdapter adapter) {
        this._adapter = adapter;
    }

    @Override
    public String encode(String password) {
        return this._adapter.encode(password);
    }

    @Override
    public String decode(String hashPassword) {
        return this._adapter.decode(hashPassword);
    }
}
```

Adaptador da API

```java
public class Base64AdapterHash implements IHashAdapter {
    private Base64API _base64;

    public Base64AdapterHash() {
        this._base64 = new Base64API();
    }

    @Override
    public String encode(String password) {
        return this._base64.encodeHashBase64(password);
    }

    @Override
    public String decode(String hashPassword) {
        return this._base64.decodeHashBase64(hashPassword);
    }
}
```

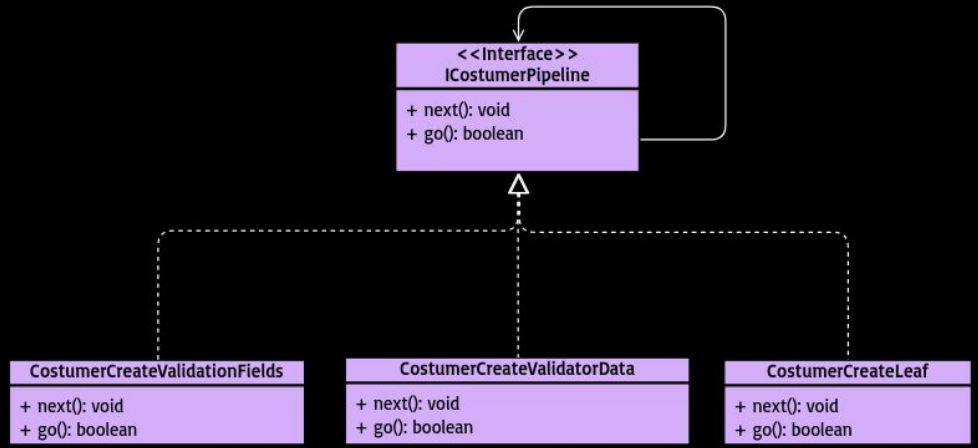
2. Chain of responsibility

Cria uma pipeline com 3 etapas para cadastro do cliente.

Etapa 1: Validação do formulário;

Etapa 2: Verificação do cliente já cadastrado;

Etapa 3: Classe folha (Leaf).



Chain of responsibility

Contrato

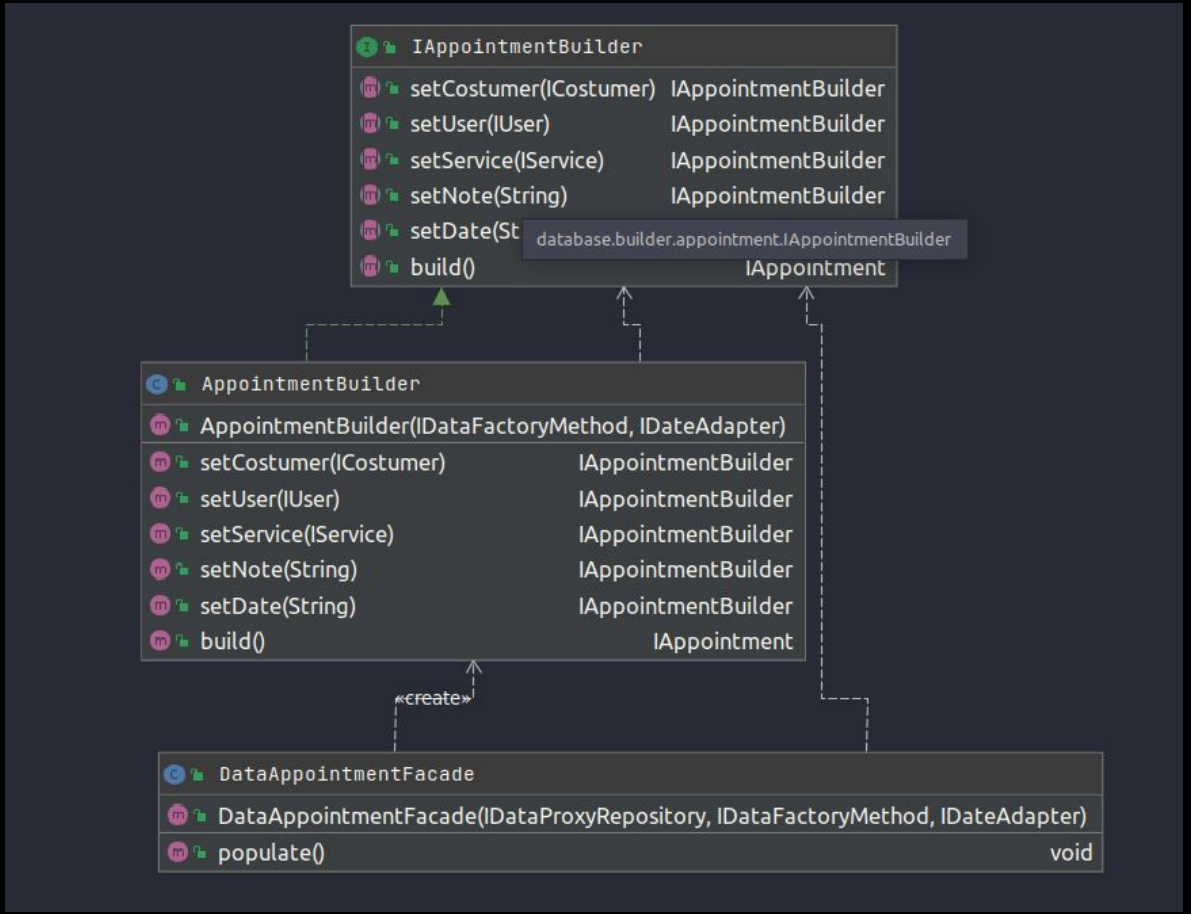
```
3  public interface ICostumerPipeline {
4      void next(ICostumerPipeline pipeline);
5
6      void go();
7  }
```

Encadeamento

```
16     public boolean execute() {
17         ICostumerPipeline validation = new CostumerCreateValidationFields(this._costumerCreateComponentHelper);
18         ICostumerPipeline verification = new CostumerCreateValidatorData(this._costumerCreateComponentHelper,
19                 this._costumerFindRepository);
20         ICostumerPipeline registration = new CostumerCreateLeaf();
21
22         validation.next(verification);
23         verification.next(registration);
24
25         boolean valid = validation.go();
26
27         return valid;
28     }
```

3. Builder

Utilizado para criar o agendamento.



Builder

Implementação

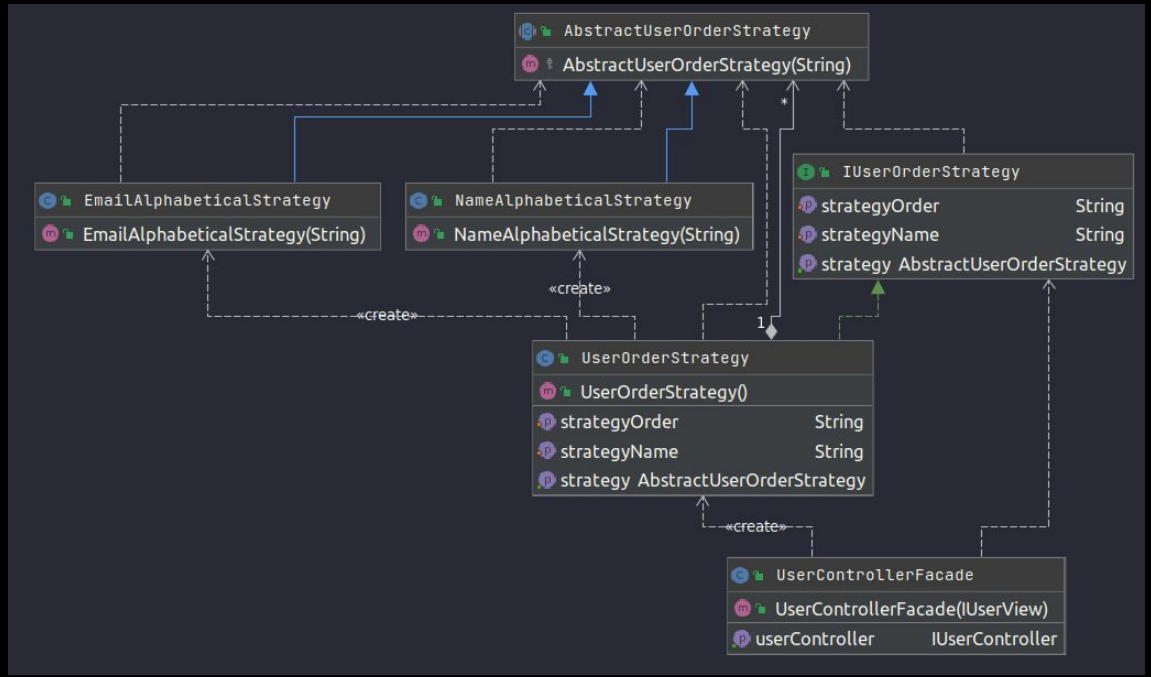
```
public class AppointmentBuilder implements IAppointmentBuilder {
    private ICostumer costumer;
    private IUser user;
    private IService service;
    private String note;
    private String date;
    private IDataFactoryMethod dataFactoryMethod;
    private IDateAdapter dateAdapter;

    public AppointmentBuilder(IDataFactoryMethod dataFactoryMethod, IDateAdapter dateAdapter) {…

    @Override
    public IAppointmentBuilder setCostumer(ICostumer costumer) {
        this.costumer = costumer;
        return this;
    }

    @Override
    public IAppointmentBuilder setUser(IUser user) {…

    @Override
    public IAppointmentBuilder setService(IService service) {…

    @Override
    public IAppointmentBuilder setNote(String note) {…

    @Override
    public IAppointmentBuilder setDate(String date) {…

    @Override
    public IAppointment build() {
        Date dateParse = this.dateAdapter.dateParse(this.date);

        IAppointment appointment = this.dataFactoryMethod.createAppointment(this.costumer, this.user, this.service,
                this.note, dateParse);

        return appointment;
    }
}
```

Utilizando

```
IAppointmentBuilder appointmentBuilder = new AppointmentBuilder(this.dataFactoryMethod,
                this.dateAdapter);

IAppointment appointmentA = appointmentBuilder.setCostumer(costumers.get(0)).setUser(users.get(0))
                .setService(services.get(0)).setNote("").setDate("15/10/2020").build();

IAppointment appointmentB = appointmentBuilder.setCostumer(costumers.get(1)).setUser(users.get(1))
                .setService(services.get(1)).setNote("").setDate("15/10/2020").build();

IAppointment appointmentC = appointmentBuilder.setCostumer(costumers.get(2)).setUser(users.get(2))
                .setService(services.get(2)).setNote("").setDate("15/10/2020").build();

this.repository.setAppointment(appointmentA);
this.repository.setAppointment(appointmentB);
this.repository.setAppointment(appointmentC);
```

4. Strategy

Utilizado para ordenação.



Strategy

Contrato

```java
public interface IUserOrderStrategy {
    public void subscribe(String name, AbstractUserOrderStrategy strategy);

    public void setStrategyName(String name);

    public void setStrategyOrder(String order);

    public AbstractUserOrderStrategy getStrategy();
}
```

Implementação

```java
public class UserOrderStrategy implements IUserOrderStrategy {
    private String nameStrategy;
    private String orderStrategy;
    private Map<String, AbstractUserOrderStrategy> _mapStrategies;

    public UserOrderStrategy() {
        this.setStrategyDefault();
        this.setupStrategies();
    }

    private void setStrategyDefault() {
        this.nameStrategy = "NAME_ALPHABETICAL";
        this.orderStrategy = "ASC";
    }

    private void setupStrategies() {
        this._mapStrategies = new HashMap<>();
        this._mapStrategies.put("NAME_ALPHABETICAL", new NameAlphabeticalStrategy(this.orderStrategy));
        this._mapStrategies.put("EMAIL_ALPHABETICAL", new EmailAlphabeticalStrategy(this.orderStrategy));
    }
```

Estratégia

```java
public class EmailAlphabeticalStrategy extends AbstractUserOrderStrategy {

    public EmailAlphabeticalStrategy(String order) {
        super(order);
    }

    @Override
    public int compare(IUser o1, IUser o2) {…
}
```

5. Proxy

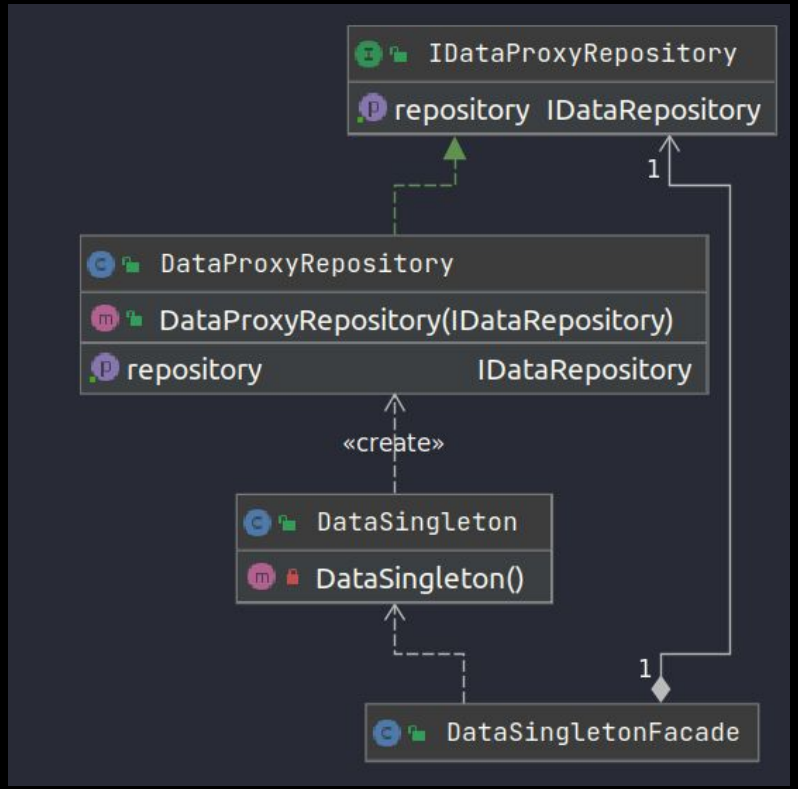
Utilizado para interceptar a criação

das entidades: usuário, cliente e agendamento.

Tem como principal objetivo, através

do CREATE, tanto criar como

atualizar um objeto existente.



Proxy

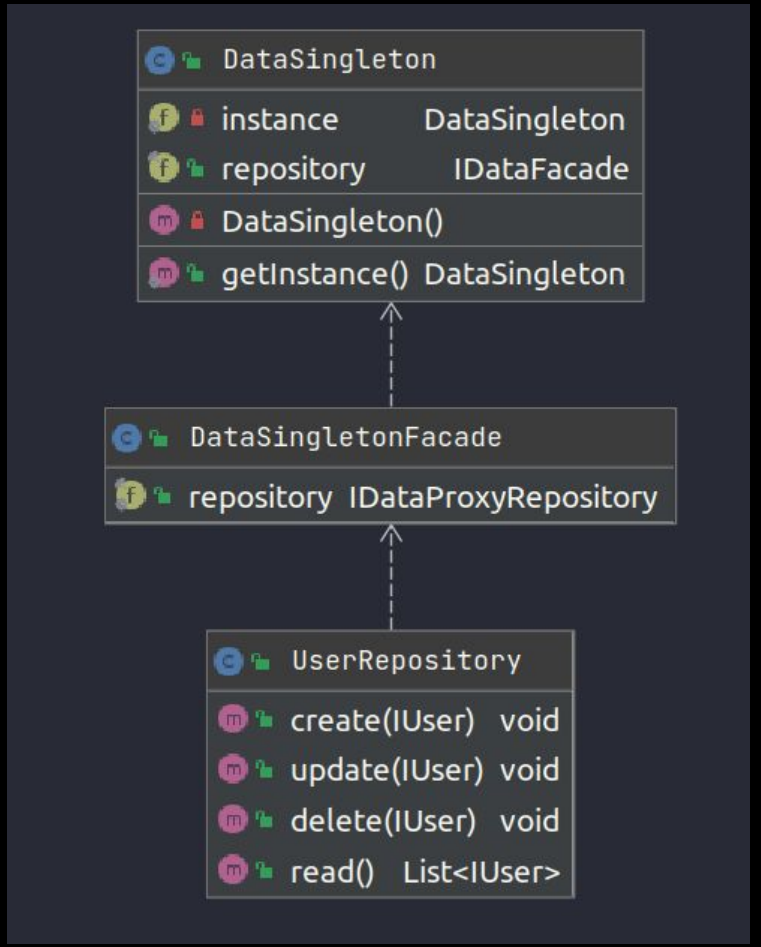
Implementação

```
10  public class DataProxyRepository implements IDataProxyRepository {
11      private IDataRepository repository;
12
13      public DataProxyRepository(IDataRepository repository) {
14          this.repository = repository;
15      }
16
17      @Override
18      public IUser setUser(IUser user) {
19          int userIndex = this.repository.getUsers().indexOf(user);
20
21          if (userIndex == -1)
22              return this.repository.setUser(user);
23
24          this.repository.getUsers().set(userIndex, user);
25
26          return user;
27      }
28
```

6. Singleton

Utilizado para criar um nó na memória

para manipulação da repository.



Singleton

Implementação

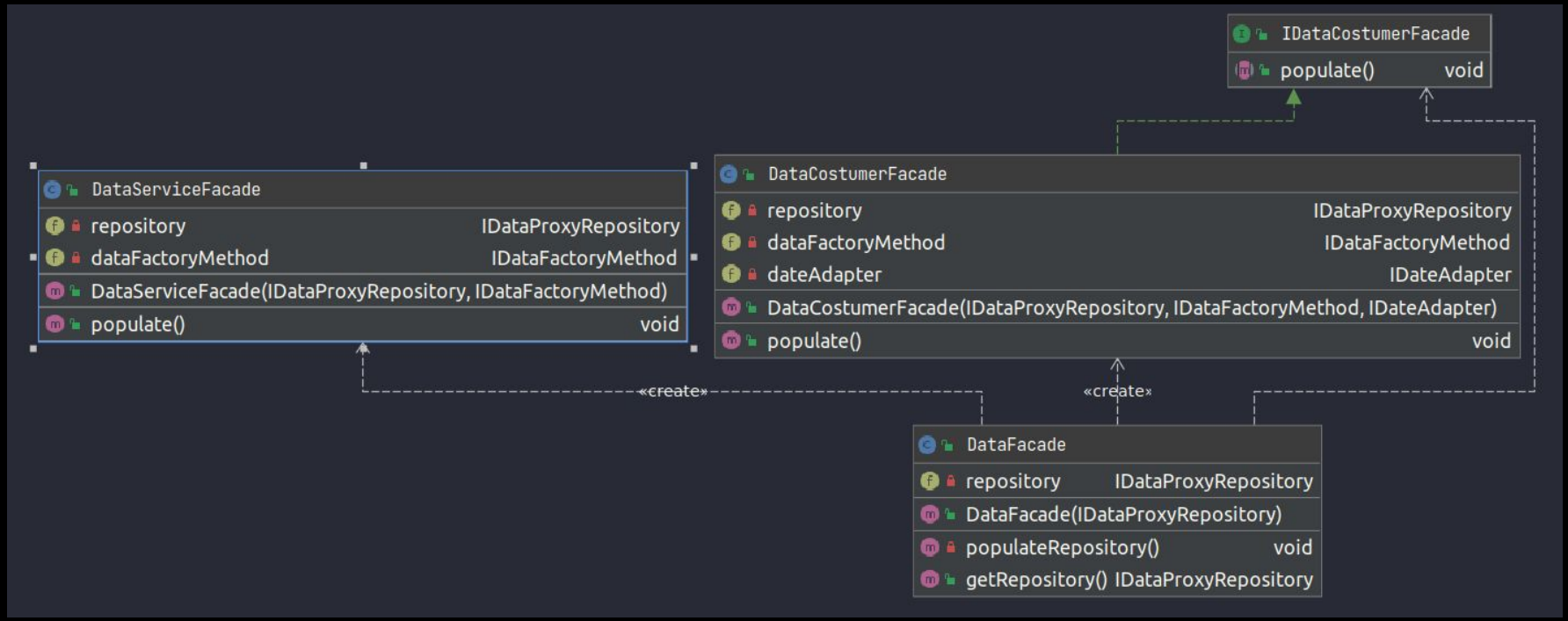
```
public class DataSingleton {
    private static DataSingleton instance;
    public final IDataFacade repository = new DataFacade(new DataProxyRepository(new DataRepository()));

    private DataSingleton() {
    }

    public static DataSingleton getInstance() {
        if (instance == null)
            return new DataSingleton();

        return instance;
    }
}
```

Utilizando

```
public class CostumerRepository implements ICostumerRepository {
    @Override
    public void create(ICostumer costumer) {
        DataSingletonFacade.repository.setCostumer(costumer);
    }

    @Override
    public void update(ICostumer costumer) {…
```

7. Facade

Criar as fachadas dos controllers.



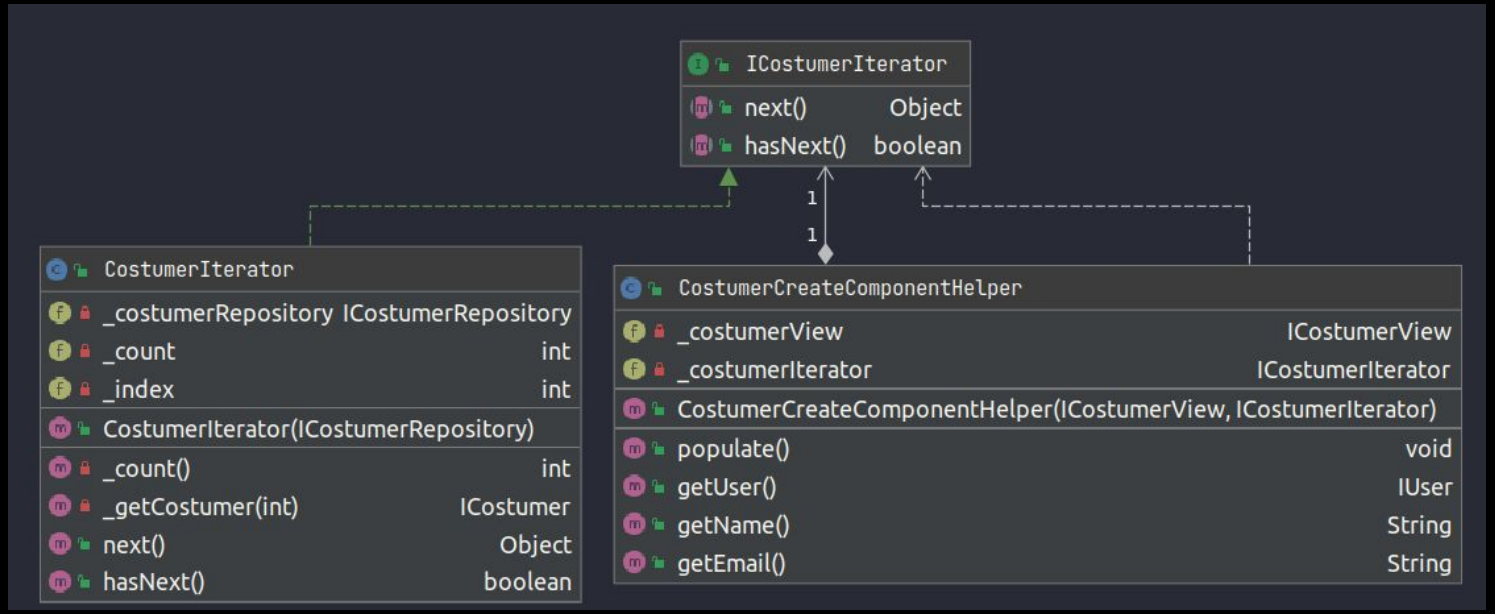
Implementação

Facade

```
25  public class CostumerControllerFacade implements ICostumerControllerFacade {
26      private ICostumerView costumerView;
27
28      public CostumerControllerFacade(ICostumerView costumerView) {
29          this.costumerView = costumerView;
30      }
31
32      @Override
33      public ICostumerController getCostumerController() {
34          ICostumerRepository costumerRepository = new CostumerRepository();
35          ICostumerFindRepository findCostumerRepository = new CostumerFindRepository(costumerRepository);
36          IUserRepository userRepository = new UserRepository();
37          INotificationRepository notificationRepository = new NotificationRepository();
38          IMessageStrategy costumerMessageStrategy = new CostumerMessageStrategy();
39
40          ICostumerDao costumerDao = new CostumerDao(costumerRepository, notificationRepository, costumerMessageStrategy);
41
42          ICostumerView costumerView = this.costumerView;
43
44          IFindUserHelper findUserHelper = new FindUserHelper(userRepository);
45          IDateAdapter dateAdapter = new DateAdapter();
46
47          ICostumerCreateComponentHelper costumerCreateComponentHelper = new CostumerCreateComponentHelper(costumerView);
48
49          ICostumerCreatePipeline costumerCreatePipeline = new CostumerCreatePipeline(costumerCreateComponentHelper,
50                  findCostumerRepository);
51
52          ICostumerController costumerController = new CostumerController(costumerDao, costumerView,
53                  findCostumerRepository, findUserHelper, dateAdapter, costumerCreatePipeline);
54
55          return costumerController;
56      }
57  }
```

8. Iterator

Encapsula as interações nos repositório.



Iterator

Contrato

```java
package domain.costumer.iterator;

public interface ICostumerIterator {
    Object next();

    boolean hasNext();
}
```

Implementação

```java
public class CostumerIterator implements ICostumerIterator {
    private ICostumerRepository _costumerRepository;
    private int _count;
    private int _index;

    public CostumerIterator(ICostumerRepository costumerRepository) {…

    private int _count() {…

    private ICostumer _getCostumer(int index) {…

    @Override
    public Object next() {
        return this._getCostumer(this._index++);
    }

    @Override
    public boolean hasNext() {
        return this._index >= this._count || this._getCostumer(this._index) == null ? false : true;
    }
}
```

Utilizando

```java
@Override
public void populateCostumers() {
    DefaultComboBoxModel comboBoxModel = (DefaultComboBoxModel) costumerView.getjComboBoxCliente().getModel();

    while (this.costumerIterator.hasNext()) {
        ICostumer costumer = (ICostumer) this.costumerIterator.next();

        comboBoxModel.addElement(costumer);
    }
}
```