CS590DE-2020-02A-05D >
Tests & Quizzes

Tests & Quizzes

Midterm exam

Return to Assessment List

Part 1 of 1 - 100.0 / 100.0 Points

Question 1 of 8 10.0 10.0 Points

[10 minutes]

Explain the brewers cap theorem and explain what this means for distributed systems / databases.

For distributed systems, only 2 of the following characteristics can be achieved at same time:
1- Availability
2- Partition Tolerance
3- Consistency

So if we want to achieve Availability with Partition tolerance, then we either have to choose eventual consistency. Otherwise if we need strict consistency then we will have less availability since components will need to wait for reads until transactions are persisted and system is in consistent state.

Question 2 of 8 10.0 10.0 Points

[10 minutes]

When we implement an integration broker / ESB, we typically send messages to channels. There are 3 different types of messages we can send. One of them is an **event** message. Give the 3 types of messages we can send, and give an example of each.

1- Event message
Example: aProductSoldEvent

2- Document message
Example: aPurchaseOrder

3- Command message
Example: getLastStockPrice

Question 3 of 8 10.0 10.0 Points

[10 minutes]

In domain driven design we have 4 different types of classes. One type of class is an **Entity**. Give the name of the other 3 types of classes. For every type of class, give its important characteristics and give an example of each.

1- Entity:
* A core model with identity
* Equality achieved through identity
* Mutable
* Has multiple states (i.e initiated, sent, approved, ...etc)
* Example: Customer

2- Value Object
* Immutable
* Doesn't have identity, its characteristics define its identity
* Equality achieved through all combined characteristics
* Can be shared with multiple entities
* Has some behavior (rich behavior)

* Testable
* Self validating
* Example: Address

3- Domain Service
* When a behavior doesn't belong to any entity/value object but is crucial for our domain, we add that behavior to a domain service.
* domain service can orchestrate multiple entities
* Doesn't have any state
* Example: PruchaseOrderingService class may have a method to create an order from a cart with list of items.

4- Domain Event
* Represent a change event in our domain
* Immutable
* A handler will listen for such events and take action (in same component or another component)
* Example: ProductSoldEvent

Question 4 of 8 10.0 10.0 Points

[15 minutes]

Suppose we have the following PackageReceiver class:

```
@Service
public class PackageReceiver {

  public void receivePackage(Package thePackage) {

   ...

  }
}
```

We need to implement the functionality that whenever we call **receivePackage(Package thePackage)** on the PackageReceiver, the **notifyCustomer(Package thePackage)** method is called on the following CustomerNotifier class:

```
@Service
public class CustomerNotifier {

  public void notifyCustomer(Package thePackage) {
    System.out.println("send email to customer that we received the
            package with code "+thePackage.getPackagecode());
  }
}
```

The most important requirement is that the PackageReceiver and the CustomerNotifier class should be as loosely coupled as possible.

Write the code of the PackageReceiver and the CustomerNotifier so that we get the desired behavior.

PackageReceiver Component:

```
@Service
public class PackageReceiver {

  @AutoWired
  private ApplicationEventPublisher publisher;

  public void receivePackage(Package thePackage) {
    publisher.publishEvent(thePackage);
  }
}
```

CustomerNotifier Component

```
@Service
public class CustomerNotifier {
  public void notifyCustomer(Package thePackage) {
    System.out.println("send email to customer that we received the
    package with code "+thePackage.getPackagecode());
  }
}

@EnableAsync
public class PackageListener
{
    @AutoWired
    private CustomerNotifier customerNotifier;

    @Async
    @EventListener
    public void onPackageRecieved(Package aPackage)
      {
        customerNotifier.notifyCustomer(aPackage);
      }
}
```

Question 5 of 8 10.0 10.0 Points

[10 minutes]

Suppose ApplicationA needs to call ApplicationB. One colleague tells you to use REST and another colleague tells you to use messaging. **Explain clearly** in what circumstances would you use **REST** and in what circumstances would you use **messaging**?

REST:
* Use for synchronous requests
* Can work publicly, across multiple organizations
* No standard or service definitions provided

* Used over HTTP/s
* When the exchanged data is only XML or JSON

Messaging:
* Use for asynchronous requests
* Within same organization
* When business process exist and it's somehow complex (Integration Logic).

* Examples: JMS, Spring Integration, other ESBs

Question 6 of 8 20.0 20.0 Points

[25 minutes]

Suppose you need to design a bank account application that allows users to perform the following actions:

- Deposit money to an account
- Withdraw money from an account
- View the details of an account
- Transfer money from one account to another account.

Our bank account application supports different currencies, so you can deposit in dollars, but also other currencies.
We need to implement the following business rules:

- You cannot withdraw more money than you have on your bank account
- Whenever the amount of a transaction is larger than $20.000, then the bank account application and maybe other applications need to know about this so they can check if this is not a fraudulent transaction

We need to design the bank account application using **Domain Driven Design**.

Give the **name of all domain classes** of the bank account application.
For every domain class specify the **type of the class (Entity,...).**
For every domain write its **attributes** and **method signatures** (name of the class, arguments and return value)

Write your solution like the following example:

Interface Counter
Methods:
void increment()
void decrement()
void setStartValue(int start)

class CounterImpl implements Counter :<<Entity>>
Attribute: value
Methods:
void increment()
void decrement()
void setStartValue(int start)

class Account: <<Entity>>
Attribute: accountNumber
attribute: accountEntries (dependency to AccountEntry class)

Methods:
boolean withdraw(decimal amount)
boolean deposit(decimal amount)
string getAccountDetails()

----

class AccountEntry: <<Value Object>>
Attribute: entryDate
Attribute: entryDescription
Attribute: money (dependency to Money class)

Methods:

----

class Money: <<Value Object>>
Attribute: amount
Attribute: currency
Methods:
void Add(decimal amount)
void subtract(decimal amount)
void equals(Money bMoney)

----

class AccountsTransferDomainService: <<Domain Service>>
Methods:
boolean TransferBetweenAccounts(Account aAccount, Account bAccount, Money money)

----

class LargeTransactionEvent : <<Domain Event>>
Attribute: accountNumber
Attribute: amount
Attribute: currency
Attribute: transactionDate

Question 7 of 8 25.0 25.0 Points

[30 minutes]

We wrote an rental application for a company that rents tools to customers.
We have the following domain classes:

```
public class Customer{
   private int customerNumber;
   private String name;
```

```
    private String email;
    private String phone;
    private List<Rental> rentals;
...
}

public class Tool{
    private int toolNumber;
    private String name;
    private double price;
    private Rental rental;
...
}

public class Rental{
    private int rentalNumber;
    private Tool tool;
    private Customer customer;
    private Date startDate;
    private Date endDate;
...
}
```

As you can see, every tool that a customer wants to rent is a new Rental. You cannot put multiple tools in one rental, but that is how the customer wants the application to work.

We have one service class with the following interface:

```
public interface RentalService {
    public void addCustomer(int customerNumber, String name, String email, String phone);
    public Customer getCustomer(int customerNumber);
    public void removeCustomer(int customerNumber);
    public void addTool(int toolNumber, String name, double price);
    public Tool getTool(int toolNumber);
    public void removeTool(int toolNumber);
    public void addRental(int rentalNumber, Tool tool, Customer customer, Date startDate, Date endDate);
    public Rental getRental(int rentalNumber);
    public void removeRental(int rentalNumber);
}
```

Now we decide to change the design of this application using **component based design** using the best practices we learned in this course.
a. For every component, write the names and attributes of every domain class
b. For every component, write the interface of every service class (show the return value, name of the method, attributes and its type)

*** Customer Component**

```
class Customer <<Entity>>{
    private int customerNumber;
    private String name;
    private String email;
    private String phone;
}
```

```
class CustomerRepository: << DAO >>{

    public void save(Customer customer);

    public Customer get(int customerNumber);

    public void remove(int customerNumber);

}
```

```
class CustomerService: <<Service>>{

    public void addCustomer(CustomerDto customerDto);
    public CustomerDto getCustomer(int customerNumber);
    public void removeCustomer(int customerNumber);
```

```
}


class CustomerDto: << Data Transfer Object>>{

     int customerNumber;

     String name;

     String email;

     String phone;

}


class CustomerAdapter: <<Mapper/Adapter>>{

  public Customer getCustomerFromDto(CustomerDto customerDto);

  public CustomerDto getDtoFromCustomer(Customer customer);

}


class CustomerController: <<Rest Controller>>{

    @PostMapping

    public void addCustomer(CustomerDto customerDto);

    @GetMapping("{customerNumber}")

    public CustomerDto getCustomer(@PathVariable int customerNumber);

    @DeleteMapping("{customerNumber}")

    public void removeCustomer(@PathVariable int customerNumber);
```

}

*** Tool Component**

```
class Tool <<Entity>>{
    private int toolNumber;
    private String name;
    private double price;
}



class ToolRepository: << DAO >>{

    public void save(Tool tool);

    public Tool get(int toolNumber);

    public void remove(int toolNumber);

}



class ToolService: <<Service>>{

    public void addTool(ToolDto toolDto);
    public ToolDto getTool(int customerNumber);
    public void removeTool(int customerNumber);

}



class ToolDto: << Data Transfer Object>>{
   int toolNumber;

   String name;

   double price;

}


class ToolAdapter: <<Mapper/Adapter>>{

    public Tool getToolFromDto(ToolDto ToolDto);

    public ToolDto getDtoFromTool(Tool tool);

}


class ToolController: <<Rest Controller>>{

    @PostMapping

    public void addTool(ToolDto toolDto);

    @GetMapping("{toolNumber}")

    public CustomerDto getTool(@PathVariable int toolNumber);

    @DeleteMapping("{toolNumber}")

    public void removeTool(@PathVariable int toolNumber);

}
```

*** Renal Component**

```
class Rental <<Entity>>{
    private int rentalNumber;
    private Tool tool;
    private Customer customer;
    private Date startDate;
    private Date endDate;
}



class Tool: << Value Object>>{

     int toolNumber;

     String name;

}



class Customer: << Value Object>>{

     int customerNumber;

     String name;

}



class RentalRepository: << DAO >>{

      public void save(Rental rental);

      public Customer get(int rentalNumber);
```

```
        public void remove(int rentalNumber);

    }



    class RentalDomainService: << Domain Service >>{

      boolean rentTool(Tool tool, Customer forCustomer);

      boolean isToolAvailable(Tool tool);

    }



    class RentalService: <<Service>>{

      public void addRental(RentalDto rentalDto);
      public RentalDto getRental(int rentalNumber);
      public void removeRental(int rentalNumber);

    }



    class RentalDto: << Data Transfer Object>>{

      int rentalNumber;

      Tool tool;

      Customer customer;

      Date startDate;

      Date endDate;

    }
```

```
class RentalAdapter: <<Mapper/Adapter>>{

  public Rental getRentalFromDto(RentalDto rentalDto);

  public RentalDto getDtoFromRental(Rental rental);

}


class CustomerProxy: <<REST Proxy>>{

  Customer getCustomer(int customerNumber);

}


class ToolProxy: <<REST Proxy>>{

  Tool getTool(int toolNumber);

}


class RentalController: <<Rest Controller>>{

   @PostMapping

   public void addRental(RentalDto rentalDto);

   @GetMapping("{rentalNumber}")

   public RentalDto getRental(@PathVariable int rentalNumber);

   @DeleteMapping("{rentalNumber}")

   public void removeRental(@PathVariable int rentalNumber);

}
```