CS590DE MIDTERM_EXAM_2020

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

10 marks

For distributed systems, only 2 of the following characteristics can be achieved at same time:

1- Availability
2- Partition Tolerance
3- Consistency

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

2. 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.

10 marks

1- Event message
   Example: aProductSoldEvent
2- Document message
   Example: aPurchaseOrder
3- Command message
   Example: getLastStockPrice

3. 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.

10 marks

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 dene 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
-

4. 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 CustomerNotier class:

```
@Service
public class CustomerNotier {
 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 CustomerNotier class should be as loosely coupled as possible. Write the code of the PackageReceiver and the CustomerNotier so that we get the desired behavior.

10 marks

PackageReceiver Component:

```
@Service

public class PackageReceiver {

@AutoWired

private ApplicationEventPublisher publisher;

public void receivePackage(Package thePackage) {

 publisher.publishEvent(thePackage);

}

}
```

CustomerNotier Component

```
@Service

public class CustomerNotier {

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

5. 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?

   10 marks

REST:

- Use for synchronous requests
- Can work publicly, across multiple organizations
- No standard or service denitions 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

6. 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 dierent 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)

20 marks

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

7. 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 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)

25 marks

*** 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: <>{
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);
 }
```

SI question

ANOTHER PAPER December 2020

[10 minutes]

Explain clearly why value object should be immutable? In other words, why do we prefer immutable classes instead of mutable classes.

An immutable class is less error prone. It is thread safe and is easy to share. Basically nothing can go wrong with immutable classes.

[15 minutes]

a. Explain the difference between horizontal and vertical scaling
   vertical scaling: use a bigger box (more memory, faster cpu, etc) horizontal scaling: use more servers
b. Explain the 3 ways of horizontal scaling. For every of these 3 options explain shortly how they work.
   ♣ Replication: replicate data over multiple servers
   ♣ Partitioning: Split up data in functional areas
   ♣ Sharding: Split the data into pieces(shards) and store them on different nodes

[20 minutes]

As an software architect you need to discover and communicate the nonfunctional requirements (NFR's) of a particular system. We learned the different steps that you need to do to find the NFR's.

a. Describe clearly the different steps you need to do to find the NFR's.

1. Have a workshop with different stakeholders

2. Explain the vision of the system

3. Explain the different qualitities

4. Everone votes (everyone gets $10 to divide)

5. Discuss the result

6. Vote again

7. Create scenario's for the top qualitites

8. Prioritice the scenario's (vote)

b. What is the end result if you applied all the steps in part a. Give an example of how this end result might look like.

1. All actions must respond in 3 seconds

2. Complex actions must respond in 5 seconds

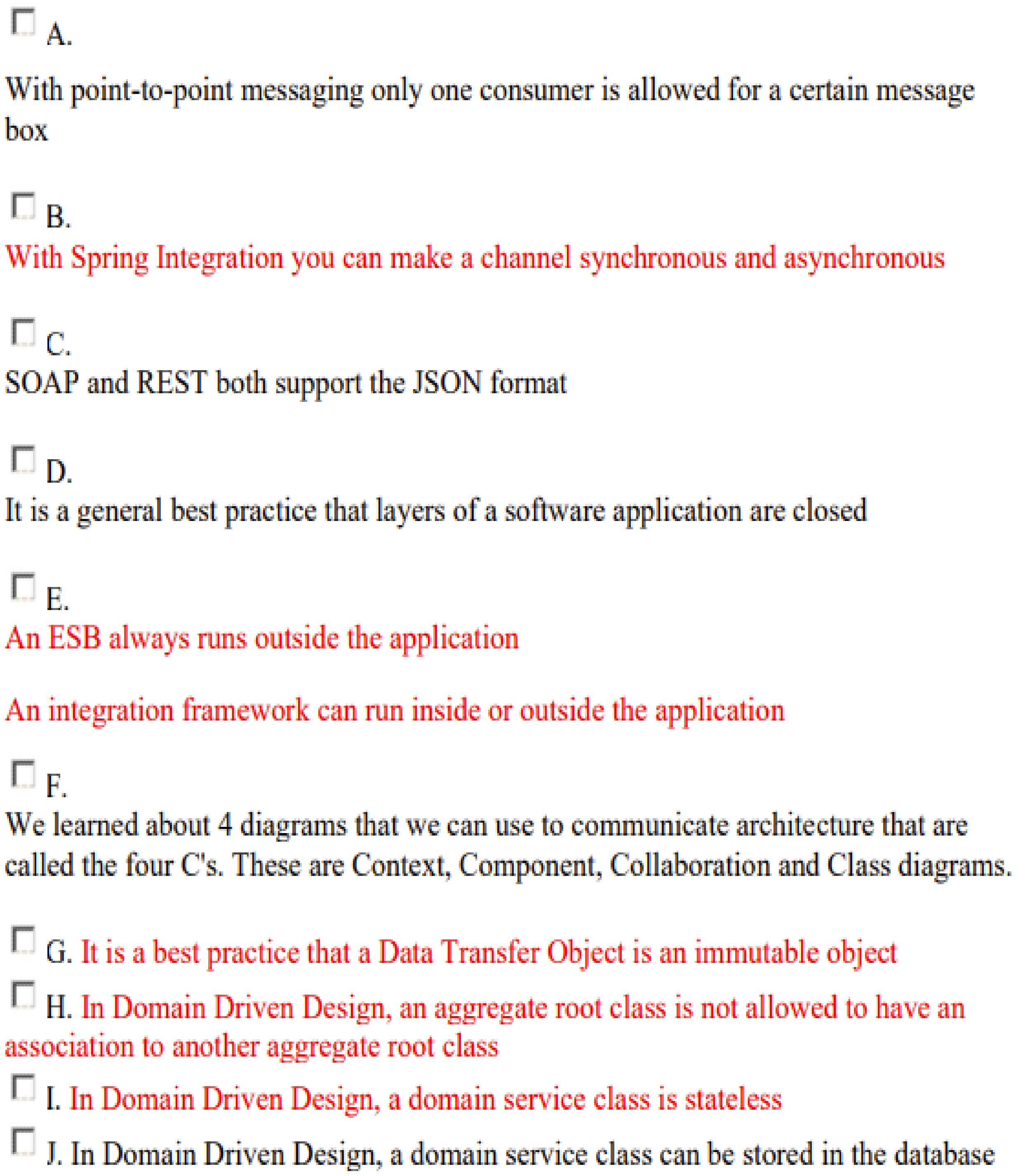
3. The system should be able to handle 5.000 concurrent users

4. The system should have 80% test coverage

5. All errors should be logged

[10 minutes]

Select all statements that are true



☐ A.

With point-to-point messaging only one consumer is allowed for a certain message box

☐ B.

With Spring Integration you can make a channel synchronous and asynchronous

☐ C.

SOAP and REST both support the JSON format

☐ D.

It is a general best practice that layers of a software application are closed

☐ E.

An ESB always runs outside the application

An integration framework can run inside or outside the application

☐ F.

We learned about 4 diagrams that we can use to communicate architecture that are called the four C's. These are Context, Component, Collaboration and Class diagrams.

☐ G. It is a best practice that a Data Transfer Object is an immutable object

☐ H. In Domain Driven Design, an aggregate root class is not allowed to have an association to another aggregate root class

☐ I. In Domain Driven Design, a domain service class is stateless

☐ J. In Domain Driven Design, a domain service class can be stored in the database

[30 minutes]

Suppose we need to design a course scheduling application for the Computer Science department at MIU using Components and DDD. This application has the following requirements:

The system keeps track of all courses (coursenumber, course name, course description) given by the department. The system also keeps track of prerequisites for certain courses.

The system also keeps track of all students in the department. For every student we need to know the studentnumber, name, email, phone and on-campus postbox number. For every student we also keep track of the grades this student got for the courses he/she took.

The system also keeps track of all professors in the department. For every professor we need to know the name, email, phone and on-campus postbox number. For every professor we also need to know which courses this professor can teach.

Every course will be taught a few times a year. Students can sign-up for certain course offerings. For every course offering the system keeps track of the students in the course, the professor who teaches it, the location where the course is given (building name, room number) and the start and end date of the course

If a student signs up for a course offering, the system will automatically check if this student has the required prerequisites for this particular course. If the student has not the required prerequisites then the student cannot sign up for this course. If the student has successfully signed up for a course, the system will send an email to this student. All data will be stored in the database.

Components talk to each other using REST webservices.

You have to design this system using the **best practices we learned in the course**. You have to use **component based design**.

For every component give the following details:

- Component name
- Layers used within the component
- ALL classes used in each layer
- For the domain classes, indicate what type of class it is according to DDD (entity, value object,...)

Your answer should look like the following example:

Component

A Layer 1: Class K

Layer 2: Class L

Layer 3: Class M (Entity), Class N (Value object)

Layer 4: Class O Component B

Layer 1: Class P Layer 2: Class Q

Layer 3: Class R (Entity), Class S (Value object)

Layer 4: Class T Do NOT specify class attributes or methods.

Component: Courses

web layer:CourseController

service layer: CourseService, CourseDTO

domain layer: Course(Entity)

data access layer: CourseDAO

integration layer:

Component: Students

web layer:StudentController

service layer: StudentService, StudentDTO

domain layer: Student(Entity), ContactInfo(Value object),

CourseGrade(Value object) data access layer: StudentDAO

integration layer:

Component: Professors

web layer: ProfessorController

service layer: ProfessorService, ProfessorDTO

domain layer: Professor(Entity), ContactInfo(Value object), Course(Value object) data access layer: ProfessorDAO

integration layer:

Component: CourseOfferings

web layer: CourseOfferingController

service layer: CourseOfferingService, CourseOfferingDTO, ...(and other DTO's)

domain layer:CourseOffering(Entity), Student(Value object), Course(Value object), Professor(Value object), Location(Value object)

data access layer: CourseOfferingDAO

integration layer: EmailSender, CoursesGateway

[15 minutes]

Suppose we have a Customer class with the attributes name and age. We want to use Spring integration to connect different components.

a. Suppose we want to write a **custom router** with **Spring integration** so that all customers who are younger than 50 will be send to the **youngcustomers** channel and all customers of 50 and older will be send to the **oldcustomers** channel. Write the java code of this customer router.

```
public class CustomerRouter {

public String route(Customer customer) {

String destinationChannel = null;

if (customer.getAge()< 50)

destinationChannel = "youngcustomers";

else

destinationChannel = "oldcustomers";

return destinationChannel;

}
```

}

b. Suppose we want to write a filter with Spring integration so that only customers who are older than 20 will be send to the next channel. Write the java code of this filter.

```
public class CustomerFilter {
public boolean filter(Customer customer) {
if (customer.getAge() > 20)
return true;
else return false;
 }
}
```

[10 minutes]

Describe how an ESB relates to one or more of the SCI principles you know. Your answer should be about 2 paragraphs. The number of points you get for this question depends on how well you explain the relationship between an ESB and the principles of SCI.

Practice midterm

Part 1

[10 minutes]

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

/ databases.

From the 3 qualities Availability, Consistency and Partition tolerance we can have only 2 of these qualities for 100%.

This means we can have Availability and Consistency but then we are not easy to scale horizontally

If we have Availability and Partition tolerance we cannot have strict consistency, only eventual consistency

If we have Consistency and Partition tolerance we don't have availability.

[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.

Event message: lowBudged message

Command message: getLastTradePrice message

Document message: newPurchaseOrder message

[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.

Entity: has identity, mutable Example: Order

Value object: has no identity, idempotent, self validating, ... Example: Address

Domain service: stateless. Example: shippingCostCalculator

Domain event: idempotent. Example: OrderReceivedEvent

[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.

```
@Service
public class PackageReceiver {
@Autowired
private ApplicationEventPublisher publisher;

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

@Service
public class CustomerNotifier {
@EventListener
public void onEvent(NewPackageEvent event) {

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

[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

When the communication is synchronous

When I do not need a buffer

When ApplicationA and ApplicationB are not within the same organization

Messaging

When I need a buffer.

When the communication is asynchronous

When ApplicationA and ApplicationB are within the same organization

When I need broadcasting

[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 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 **attribute**s 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>>

Attributes: accountNumber, balance

Methods:

void deposit()

void withdraw()

class Money:<<Value object>>

Attributes: amount, currency

Methods:

void add(Money money)

void subtract(Money money)

class AccountEntry:<<Value object>>

Attributes: amount, date, description

class TransferFundsService:<<Domain service>>

Methods:

void transferFund(Money money, Account fromAccount, Account toAccount, String description)

class LargeTransactionEvent:<<Domain event>>

Attributes: amount, date, description, fromAccountNumber, toAccountNumber, customerName

[10 minutes]

In Spring integration we have different types of channels. Give 4 different types of channels that are supported by Spring integration and explain how these channels are different in their behavior

- Synchronous point-to-point channel
- Asynchronous point-to-point channel
- Synchronous publish-subscribe channel
- Asynchronous publish-subscribe channel
- Datatype channel

[5 minutes]

Give 5 different examples of containers that go on a container diagram

- Web servers
- Application servers
- ESBs
- Databases
- Other storage systems
- File systems
- Windows services
- Standalone/console applications
- Web browsers

[10 minutes]

Give the advantages and disadvantages of the pipe-and-filter architectural style

Benefits

- Filters are independent
- Filters are reusable
- Order of filters can change
- Easy to add new filters
- Filters can work in parallel

Drawbacks

- Works only for sequential processing
- Sharing state between filters is difficult

[10 minutes]

Describe how a DDD aggregate relates to one or more of the SCI principles you

know. Your answer should be about 2 till 3 paragraphs. The number of points you get

for this question depends on how well you explain the relationship between a DDD

aggregate and the principles of SCI.

Another Question paper

[15 minutes]

a. [10 points]

Explain clearly what aspect of a relational database is the reason why it is difficult to scale out a relational database

Because a relational database is strict consistent and it is very difficult to scale out a relational database and be still available and strict consistent. It is very difficult to scale out writes to the database.

b. [10 points]

Explain clearly the consequence of scaling out a No-SQL database. In other words, what is the price you pay for the ability to scale out No-SQL databases?

The consequence is that we have eventual consistency with a No-SQL database

[15 minutes]

Describe clearly why it is not good to use an anemic domain model

- You do not use the powerful OO techniques to organize complex logic.
- Business logic (rules) is hard to find, understand, reuse, modify.
- The software reflects the data structure of the business, but not the behavioral organization
- The service classes become too complex
- No single responsibility
- No separation of concern

[15 minutes]

In Domain Driven Design we learned to divide our domain into subdomain types.

a. [10 points]

Give the name of the different subdomain types we learned and describe in one sentence the characteristic(s) of these subdomain types.

- Core subdomain: This is the reason you are writing the software.
- Supporting subdomain: Supports the core domain
- Generic subdomain: Very generic functionality(Email sending service, Creating reports service)

b. [10 points]

Explain clearly why it is important to identify these subdomain types.

If you know these subdomains, you know

- On which part of the application should I put the focus
- Where to put your most experienced developers
- What code should be of the highest quality
- Where to apply DDD

[30 minutes]

Suppose we need to design a **course scheduling application** for the Computer Science department at MIU using Components and DDD. This application has the following requirements:

The system keeps track of all courses (coursenumber, course name, course description) given by the department. The system also keeps track of prerequisites for certain courses. The system also keeps track of all students in the department. For every student we need to know the studentnumber, name, email, phone and on-campus postbox number. For every student we also keep track of the grades this student got for the courses he/she took.

The system also keeps track of all professors in the department. For every professor we need to know the name, email, phone and on-campus postbox number. For every professor we also need to know which courses this professor can teach.

Every course will be taught a few times a year. Students can sign-up for certain course offerings. For every course offering the system keeps track of the students in the course, the professor who teaches it, the location where the course is given (building name, room number) and the start and end date of the course

If a student signs up for a course offering, the system will automatically check if this student has the required prerequisites for this particular course. If the student has not the required prerequisites then the student cannot sign up for this course. If the student has successfully signed up for a course, the system will send an email to this student.

All data will be stored in the database.

Components talk to each other using REST webservices.

You have to design this system using the best practices we learned in the course. You have to use component based design.

For every component give the following details:

• Component name

• Layers used within the component

• ALL classes used in each layer

• For the domain classes, indicate what type of class it is according to DDD (entity, value object,...)

Your answer should look like the following example:

Component A

Layer 1: Class K

Layer 2: Class L

Layer 3: Class M (Entity), Class N (Value object)

Layer 4: Class O

Component B

Layer 1: Class P

Layer 2: Class Q

Layer 3: Class R (Entity), Class S (Value object)

Layer 4: Class T

Do NOT specify class attributes or methods.

Component: Courses

web layer:CourseController

service layer: CourseService, CourseDTO

domain layer: Course(Entity)

data access layer: CourseDAO

integration layer:

Component: Students

web layer:StudentController

service layer: StudentService, StudentDTO

domain layer: Student(Entity), ContactInfo(Value object), CourseGrade(Value object)

data access layer: StudentDAO

integration layer:

Component: Professors

web layer: ProfessorController

service layer: ProfessorService, ProfessorDTO

domain layer: Professor(Entity), ContactInfo(Value object), Course(Value object)

data access layer: ProfessorDAO

integration layer:

Component: CourseOfferings

web layer: CourseOfferingController

service layer: CourseOfferingService, CourseOfferingDTO, ...(and other DTO's)

domain layer:CourseOffering(Entity), Student(Value object), Course(Value object), Professor(Value object), Location(Value object)

data access layer: CourseOfferingDAO

integration layer: EmailSender, CoursesGateway

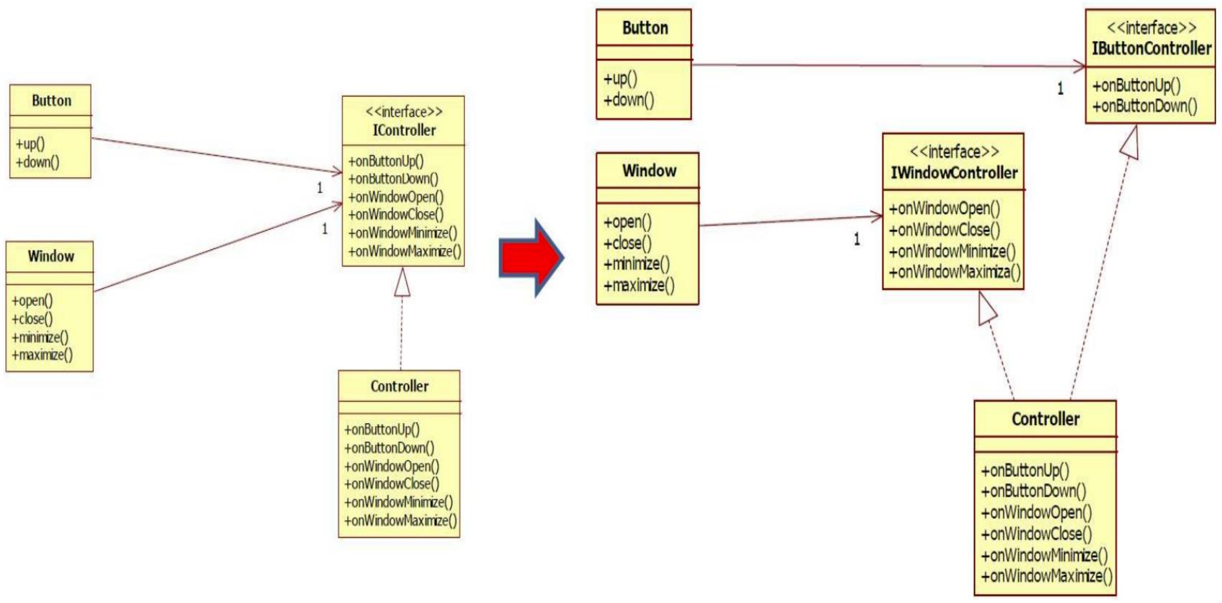
[15 minutes]

Explain what we mean with the Interface **Segregation Principle.**

Write a **clear example** that does **not** use this principle.

Write a **clear example** that does use this principle.

- Clients should not be forced to depend on methods (and data) they do not use



Button
+up()
+down()
Window
+open()
+close()
+minimize()
+maximize()
<<interface>>
IController
+onButtonUp()
+onButtonDown()
+onWindowOpen()
+onWindowClose()
+onWindowMinimize()
+onWindowMaximize()
1
1
Controller
+onButtonUp()
+onButtonDown()
+onWindowOpen()
+onWindowClose()
+onWindowMinimize()
+onWindowMaximize()
Button
+up()
+down()
<<interface>>
IButtonController
+onButtonUp()
+onButtonDown()
1
Window
+open()
+close()
+minimize()
+maximize()
<<interface>>
IWindowController
+onWindowOpen()
+onWindowClose()
+onWindowMinimize()
+onWindowMaximiza()
1
Controller
+onButtonUp()
+onButtonDown()
+onWindowOpen()
+onWindowClose()
+onWindowMinimize()
+onWindowMaximize()