Student ID_____ **Student Name**_____

Software Architecture

Midterm Exam July 5, 2018

PRIVATE AND CONFIDENTIAL

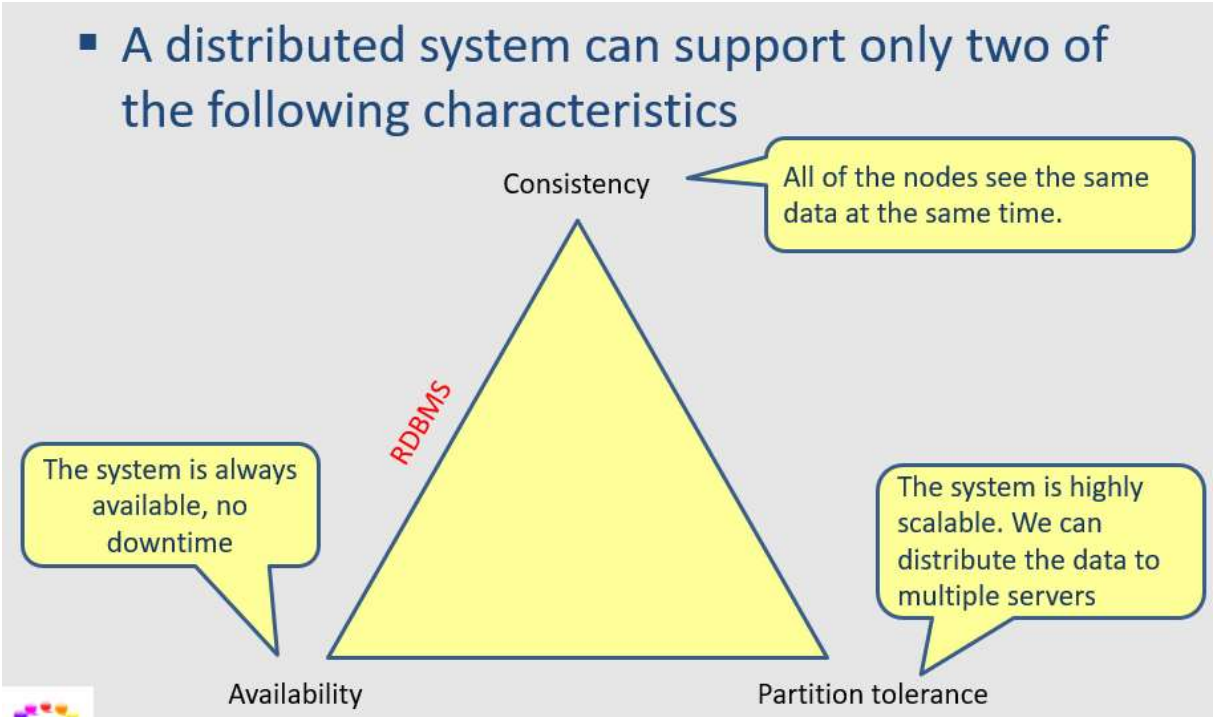
1. **Allotted exam duration is 3 hours.**
2. **Closed book/notes.**
3. **No personal items including electronic devices (cell phones, computers, calculators, PDAs).**
4. **No additional papers are allowed. Sufficient blank paper is included in the exam packet.**
5. **Exams are copyrighted and may not be copied or transferred.**
6. **Restroom and other personal breaks are not permitted.**

Write your name and student id at the top of this page.

Write your answers clearly. If I cannot read it, you don't get points for it.

Question 1 [10 points] {10 minutes}

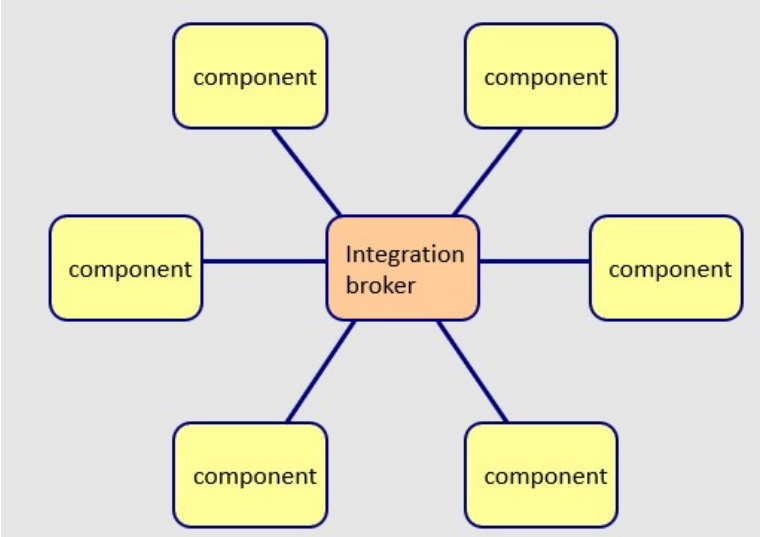
Explain the brewers cap theorem and explain what this means for distributed systems / databases.



This means that if we want to scale out and be partition tolerance, we either have eventual consistency together with availability, or we have strict consistency with less availability, because we have to wait till all systems become consistent before we can use them again.

Question 2 [5 points] {10 minutes}

Explain the hub and spoke architecture style.
When would you use this style?
What problem(s) does it solve?
Explain the advantages and disadvantages.



You would use this if you need to connect a lot of systems together.
It solves the integration problem, where every system needs to talk to many other systems, and these systems use different techniques, languages and data structures.

- Benefits
  - Separation of integration logic and application logic
  - Easy to add new components
  - Use adapters to plugin the integration broker
- Drawbacks
  - Single point of failure
  - Integration brokers are complex products
  - Integration broker becomes legacy itself

Question 3 [5 points] {5 minutes}

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

Type of message	**Example of this type of message**
1 Command message	getLastTradePrice
2 Document message	aPurchaseOrder
3 Event message	priceChangeEvent

Question 4 [10 points] {10 minutes}

Give the different techniques we studied for point-to-point connections between 2 applications.
For each of the different techniques, give the advantage(s), disadvantage(s) and when you would use this technique.

technique	advantage	disadvantage	When to use?
RMI		Only Java to Java High coupling	Never
Messaging	Buffer Low coupling Asynchronous	You need messaging middleware	Between applications within the same organization When you need asynchronous requests
SOAP	Standards for security, transactions, etc. Standard for interface description	Complex	When you need standards
REST	Simple	No standards for security, transactions, etc No standard for interface description	Everywhere you need synchronous requests
Serialized objects over HTTP	Simpler as RMI Webcontainer functionality	Only Java to Java High coupling	For fast Java to Java integration between applications managed by the same project
Database integration	Simple	High coupling	Never
File based integration		High coupling	If you have to communicate large files

Question 5 [15 points] {10 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.

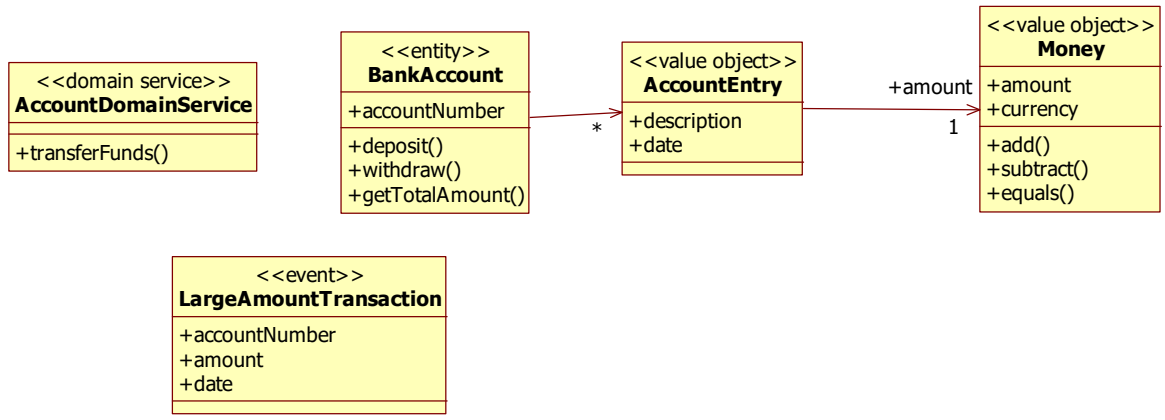
We need to implement the following business rules:

- 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**.

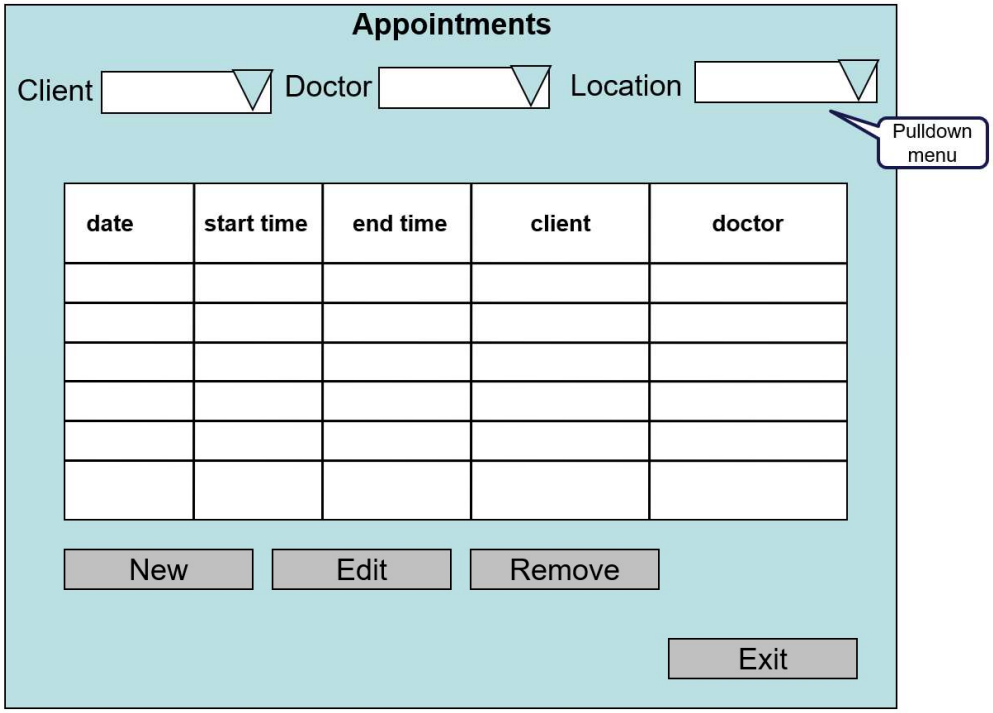
Draw the class diagram of **only the domain classes** of the bank account application. For every class show what type of class it is. Also show all attributes, methods and relations.

Your answer:

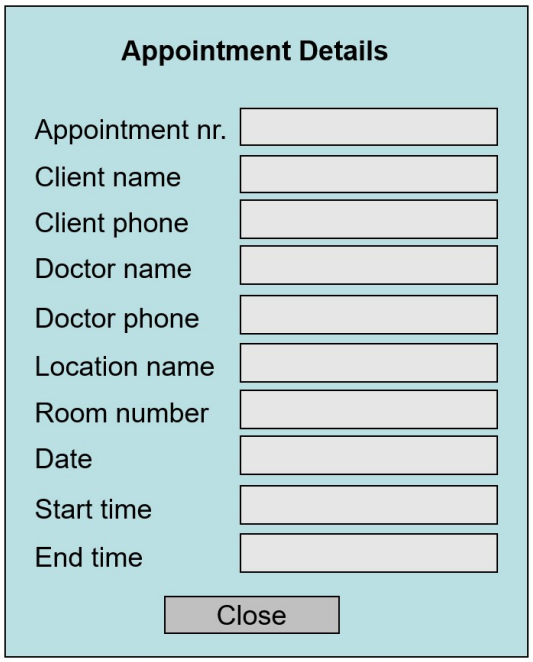


Question 6 [30 points] {55 minutes}

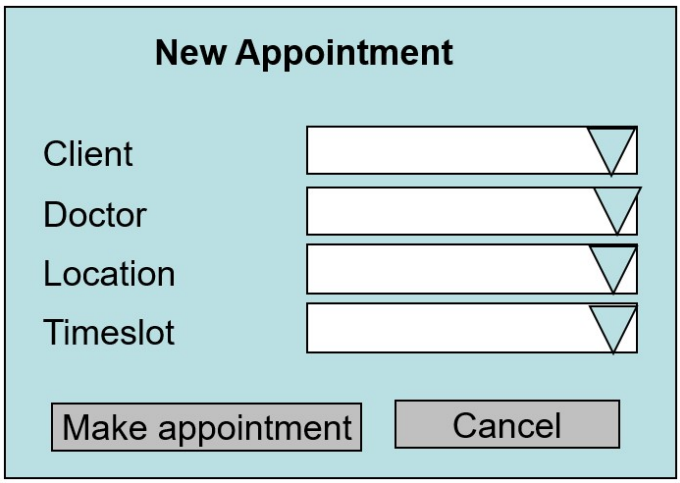
Suppose we need to design a **doctor's appointment application** using DDD. The user interface looks as follows:



The user can see the appointments for a selected client, doctor or location. The user can also add a new appointment, edit an existing appointment or remove an appointment. In the table that shows all appointments, you can double-click on an appointment which will show the following appointment details window:

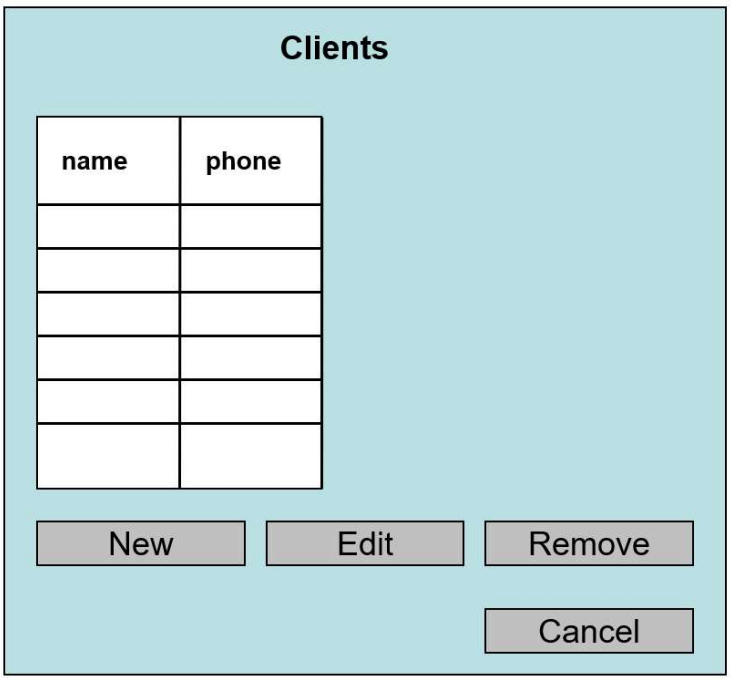


When you click the Add button you can add a new appointment using the following window:



The time is divided into fixed timeslots.
A client can make only 3 appointments in total. For every appointment the system needs to check if the location and the doctor is available at that timeslot.

The application also allows you to add, edit, remove or view **clients**, **doctors** and **locations**:

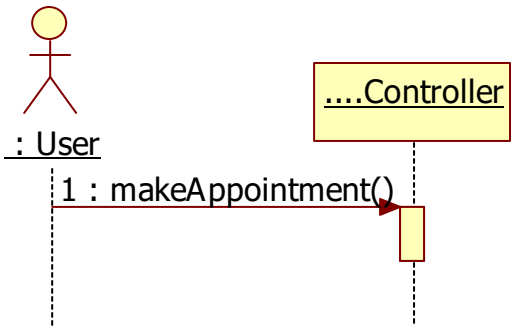


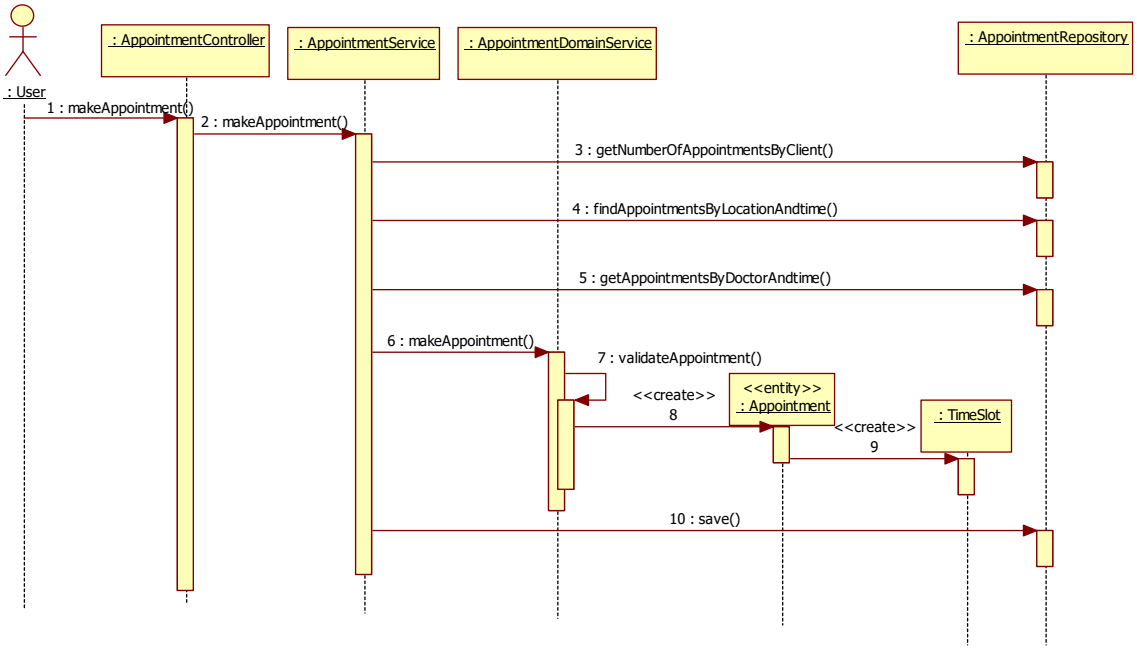
We have the following additional requirements:

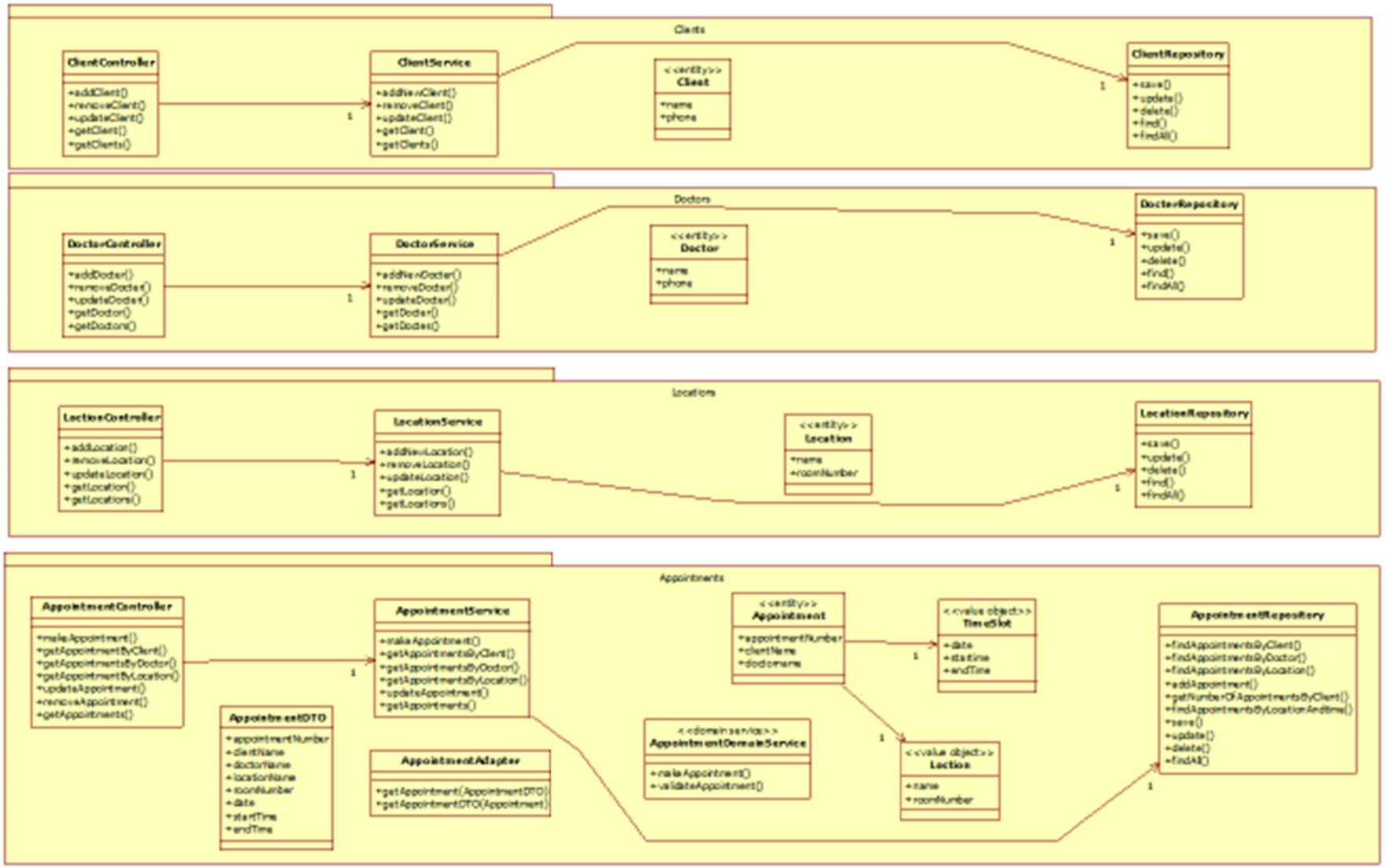
- We have to apply **Domain Driven Design**.
- It should be easy to add new functionality with minimal impact on the other parts of the system
- Because this will eventually a very large system, we need to be able to work with different agile scrum teams on this application.
- We are **NOT** allowed to use an ESB.

a. Draw the class diagram of your application. Show ALL classes that you need. Show clearly all attributes, methods, relationships, multiplicity, class type (entity, …)

b. Draw the sequence diagram of the scenario when the user makes a new appointment. Show all objects involved on your sequence diagram.
Your sequence diagram should start like this:



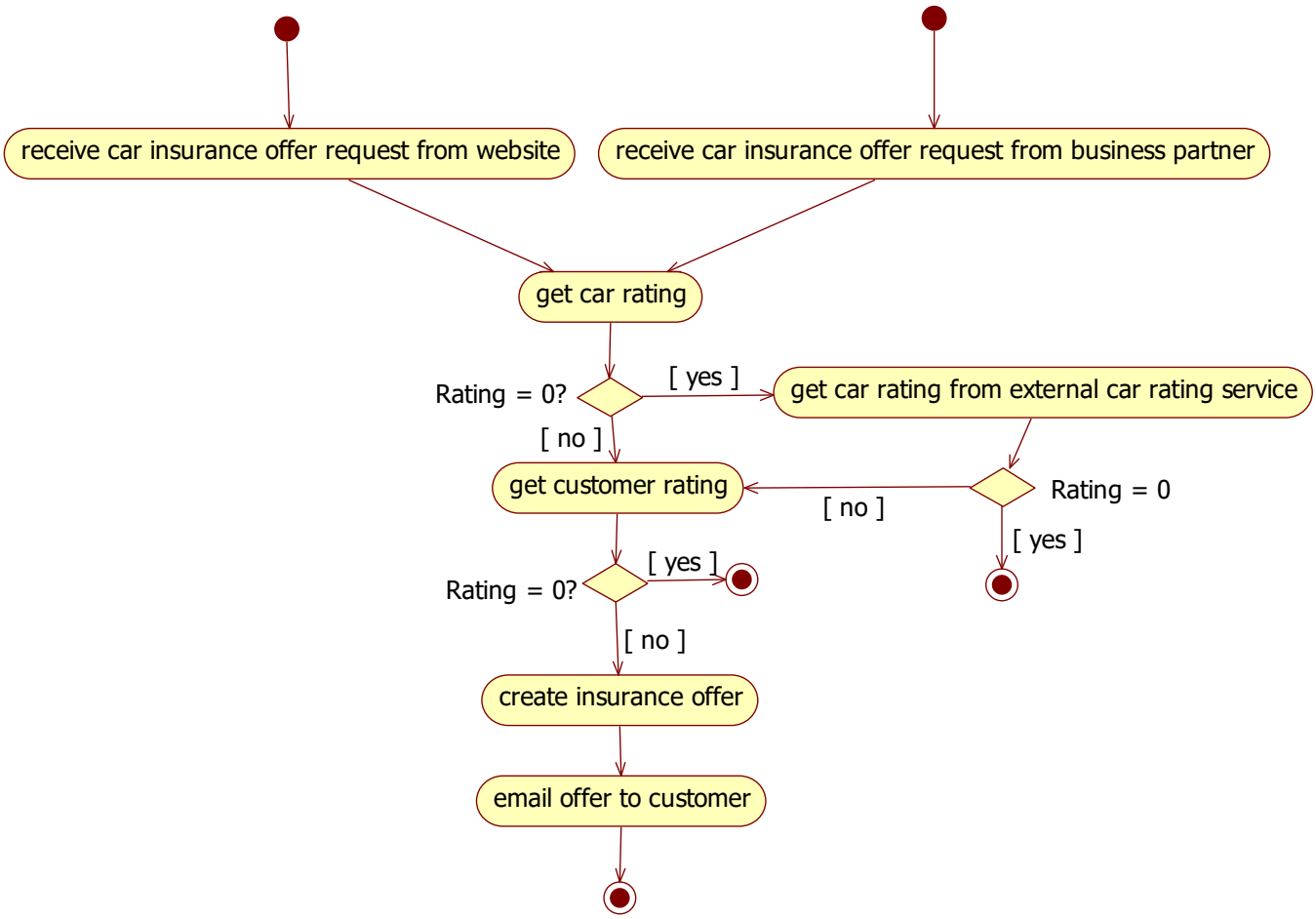



Clients
ClientController
+addClient()
+removeClient()
+updateClient()
+getClient()
+getClients()
ClientService
+addNewClient()
+removeClient()
+updateClient()
+getClient()
+getClients()
<<entity>>
Client
+name
+phone
ClientRepository
+save()
+update()
+delete()
+find()
+findAll()
Doctors
DoctorController
+addDoctor()
+removeDoctor()
+updateDoctor()
+getDoctor()
+getDoctors()
DoctorService
+addNewDoctor()
+removeDoctor()
+updateDoctor()
+getDoctor()
+getDoctors()
<<entity>>
Doctor
+name
+phone
DoctorRepository
+save()
+update()
+delete()
+find()
+findAll()
Locations
LoctionController
+addLocation()
+removeLocation()
+updateLocation()
+getLocation()
+getLocations()
LocationService
+addNewLocation()
+removeLocation()
+updateLocation()
+getLocation()
+getLocations()
<<entity>>
Location
+name
+roomNumber
LocationRepository
+save()
+update()
+delete()
+find()
+findAll()
Appointments
AppointmentController
+makeAppointment()
+getAppointmentByClient()
+getAppointmentsByDoctor()
+getAppointmentByLocation()
+updateAppointment()
+removeAppointment()
+getAppointments()
AppointmentDTO
+appointmentNumber
+clientName
+doctorName
+locationName
+roomNumber
+date
+startTime
+endTime
AppointmentService
+makeAppointment()
+getAppointmentsByClient()
+getAppointmentsByDoctor()
+getAppointmentsByLocation()
+updateAppointment()
+getAppointments()
AppointmentAdapter
+getAppointment(AppointmentDTO)
+getAppointmentDTO(Appointment)
<<entity>>
Appointment
+appointmentNumber
+clientName
+doctorname
<<value object>>
TimeSlot
+date
+startime
+endTime
<<domain service>>
AppointmentDomainService
+makeAppointment()
+validateAppointment()
<<value object>>
Loction
+name
+roomNumber
AppointmentRepository
+findAppointmentsByClient()
+findAppointmentsByDoctor()
+findAppointmentsByLocation()
+addAppointment()
+getNumberOfAppointmentsByClient()
+findAppointmentsByLocationAndtime()
+save()
+update()
+delete()
+findAll()

Question 7 [20 points] {40 minutes}

Your boss asks you to implement **Car Insurance Offer Application** as a **Service Oriented Architecture** with the following business process:

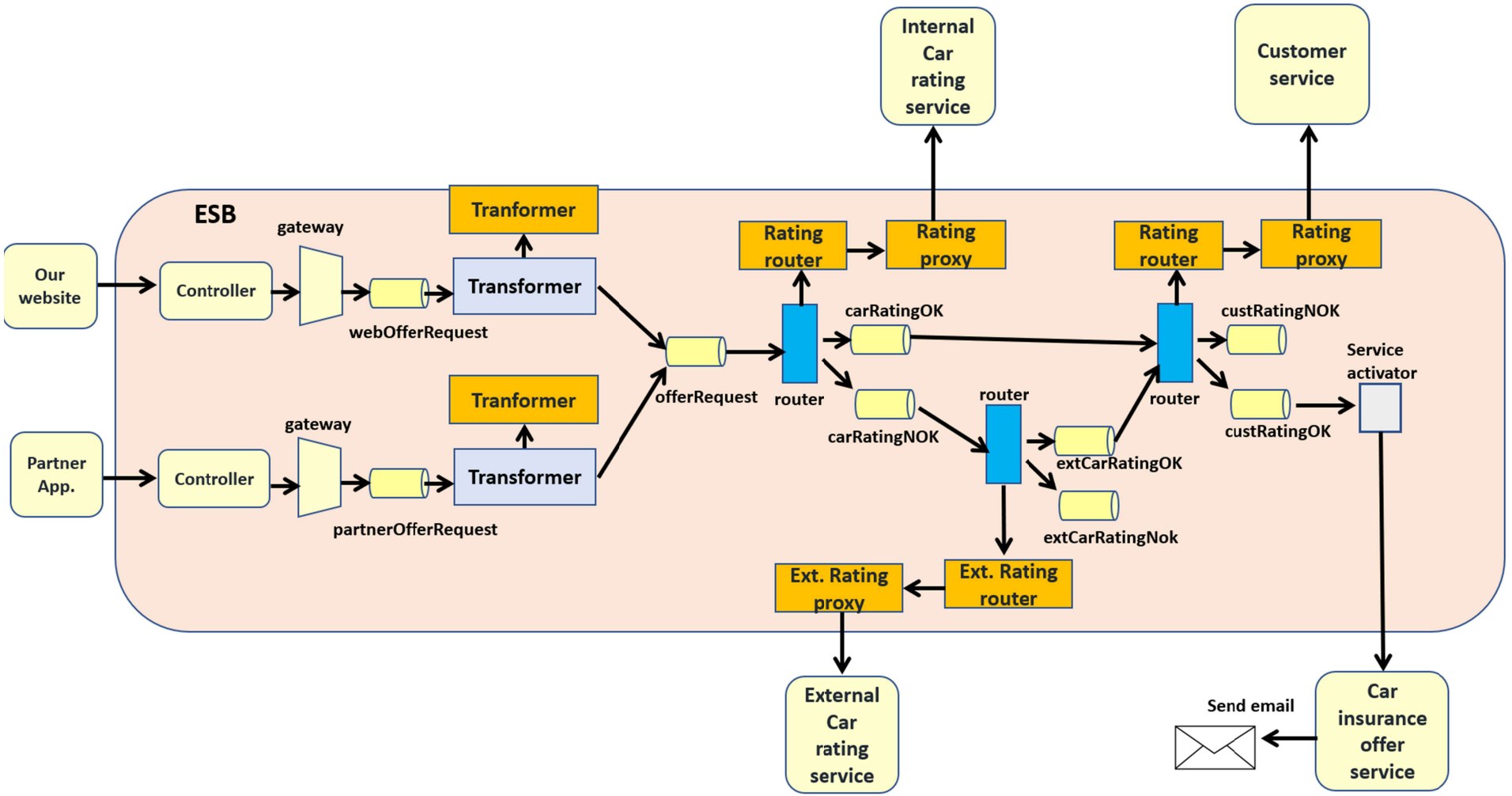
- We receive car insurance requests from our website through a REST interface.
- We also have a business partner that forwards car insurance offer requests to us through a different REST endpoint than the endpoint used by our own website.
- The data structure we receive from our website is also different from the data structure we receive from our business partner.
- After we receive a car insurance requests, we go to our own car rating functionality to get a rating for this particular car. A car rating a number between 0 and 100. Zero means that our car rating software cannot create a rating for this car.
- If the car rating is 0, we will call an external car rating service.
- If the external car rating service also returns 0, our process stops.
- After we get a car rating, we calculate a customer rating. A customer rating a number between 0 and 100. Zero means that our customer rating software cannot create a rating for this customer and our process stops.
- With both a car rating and a customer rating, we can calculate an car insurance offer that we send in an email to the customer.



Architectural requirements:

- It should be easy to add new functionality with minimal impact on the other parts of the system
- Because this will eventually a very large system, we need to be able to work with different agile scrum teams on this application.
- We have to use an ESB and we choose to use Spring Integration to implement the ESB.

Draw your solution in one diagram.
Show the integration design using the integration patterns we studied.


Internal Car rating service
Customer service
ESB
Tranformer
gateway
Rating router
Rating proxy
Rating router
Rating proxy
Our website
Controller
Transformer
webOfferRequest
carRatingOK
custRatingNOK
Service activator
offerRequest
router
router
Tranformer
gateway
carRatingNOK
router
Partner App.
Controller
Transformer
partnerOfferRequest
extCarRatingOK
custRatingOK
extCarRatingNok
Ext. Rating proxy
Ext. Rating router
External Car rating service
Send email
Car insurance offer service

Question 8 [5 points] {10 minutes}

Describe how an Enterprise Service Bus (ESB) relates to one or more of the SCI principles you know. Your answer should be about half a page, but should not exceed one page (handwritten). The number of points you get for this question depends on how well you explain the relationship between an Enterprise Service Bus (ESB) and the principles of SCI. Write clearly, I cannot give points if I cannot read it.

Your answer:

[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 non-functional 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

[10 minutes]

What are the 4 responsibilities of the integration broker in a hub and spoke architecture?

[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.

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.

a.

```
public class CustomerRouter {
  public String route(Customer customer) {
    String destinationChannel = null;
    if (customer.getAge()< 50)
      destinationChannel = "youngcustomers";
    else
      destinationChannel = "oldcustomers";
    return destinationChannel;
  }
}
```

b.

```
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 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>>
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]

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.

[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]

Select all statements that are correct

- [ ] A. It is a general best practice that layers of a software application are closed
- [ ] B. DTO classes are immutable
- [ ] C. In Domain Driven Design, a domain service class can be stored in the database
- [ ] D. In Domain Driven Design, entity classes are immutable
- [ ] E. An entity object in domain driven design is stateless
- [ ] F. A value object in domain driven design is stateless
- [ ] G. A domain service object in domain driven design is stateless
- [ ] H. The domain service object in domain driven design resides in the service layer
- [ ] I. A custom router in spring integration returns a boolean
- [ ] J. A custom router in spring integration returns a String

Suppose you are hired by a startup company that want to revolutionize the world of travel. They will offer travel deals worldwide and they want to become the world leader in the domain of travel. They have no software systems yet, and it is your task to define the architecture of their IT system(s). The architecture has the following requirements:

1. There is a lot of functionality to write (customers, hotels, flights, car rentals, packages, insurances, etc.)
2. You need at least 50 developers to build all this functionality.

Some people advice you to use a Service Oriented Architecture style while other people advice you to choose the microservice architecture.

a. Give the advantages and disadvantages of both architectural styles.

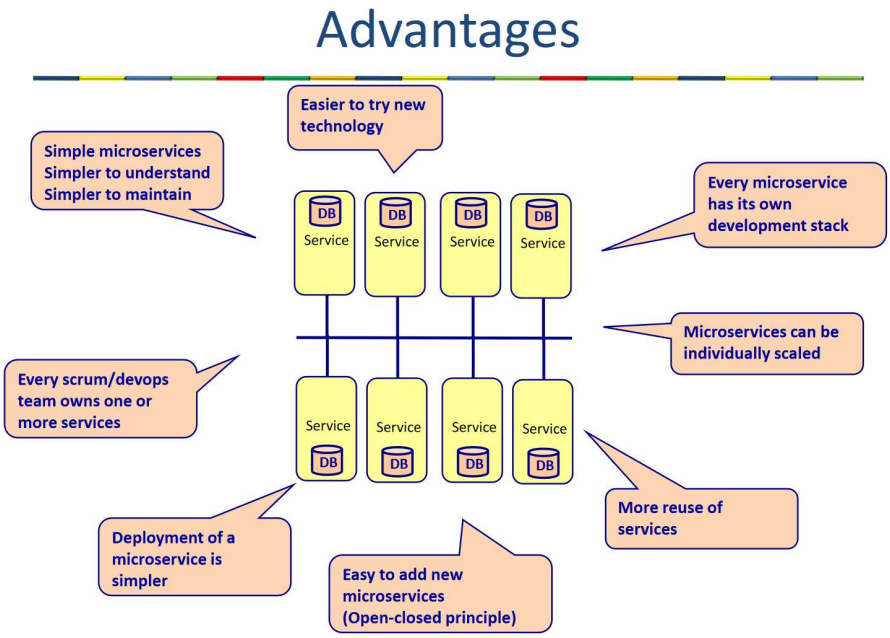
SOA advantages

- Support agile development. Every scrum team works on one service
- The business process is easy to understand and is easy to change.
- Every service can be build, deployed and scaled in isolation

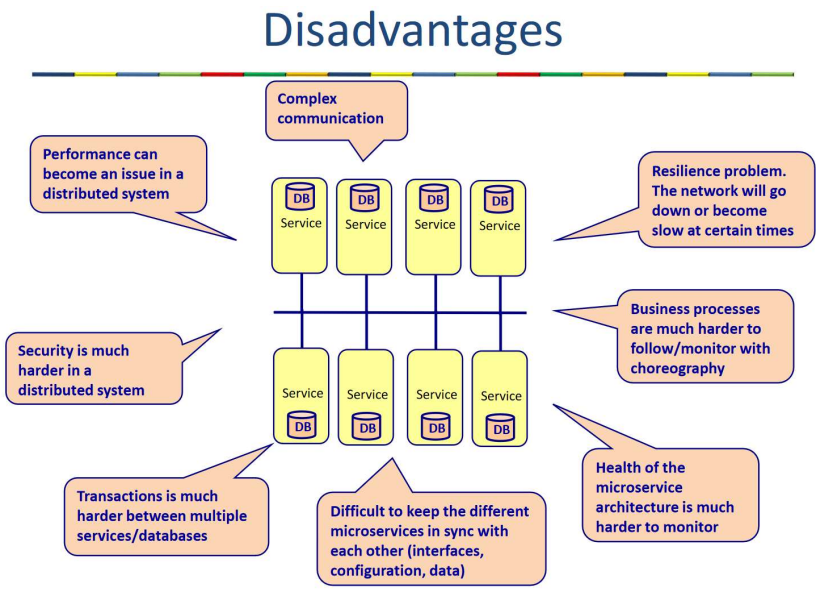
SOA disadvantages

- Complexity of a distributed system (performance, security, etc)
- In a distributed system, remote call will go wrong.
- Orchestration: ESB gets complex

Microservice advantages



Microservice disadvantages



b. What architectural style would you use and why? The number of points you get depends on how well you explain why you would choose this style.

Because there is a lot of functionality and this is a world wide travel system, you want to use choreography (and not orchestration). This is the main reason to use microservices and not SOA.

a. (3 points) We studied 4 different types of NoSQL databases. Give the name of all 4 different types of NoSQL databases.

- Document databases

- Graph databases

- Column family databases

- Key-value pair databases

b. (4 points) Explain clearly the Brewers CAP theorem

From the 3 qualities Consistency, Availability and Partition tolerance we can have only 2 for 100%, not 3.

c. (8 points) In a distributed MongoDB (or Cassandra database) we typically have data duplication. Explain clearly why we have data duplication in these databases.

If we have to distribute the data over multiple nodes, and we want fast queries, then every query should only have to access one node. This can only work if we have data duplication.

[15 minutes]

a. Give the 3 main problems of a relational database that are solved by the NoSQL databases.

1. Hard to scale
2. Hard to change (fixed schema)
3. Does not handle unstructured or semi structured data very well

Suppose you use a relational database for your application, but the database get too much load.

b. Explain what technique you can use to solve this problem

Master-slave replication

c. What is/are the main disadvantage(s) of this technique

Writes need to go to the same node. You cannot scale writes without losing strict consistency

Suppose you use a relational database for your application, but the database contains too much data so that it does not fit on one node anymore.

d. Explain what technique you can use to solve this problem

Sharding

e. What is/are the main disadvantage(s) of this technique

Data duplication
eventual consistency

[15 minutes]

Suppose you need to design a large application with the following main requirements:

- The application is large and we need 4 scrum teams that will work on this application
- Performance is crucial for this application. Almost all user actions are request-response type of actions and should be very fast.
- The system does not need to be scalable, there are only 400 users.
- We do not need fast deployment for this system.
- We need to avoid architectural complexity as much as possible

Describe clearly the architecture style(s) you use for this system based on the given requirements.

The number of points you get depends on how well you explain the architecture relevant choices you make and how these choices help to achieve the required requirements.

The correct answer is a monolith using component based design. A monolith for fast performance and architectural simplicity. Components for the ability to have every scrum team to work independently from each other.

A microservice architecture introduces a lot of architectural complexity.

Suppose we need to design an application for a car rental company with the following requirements:

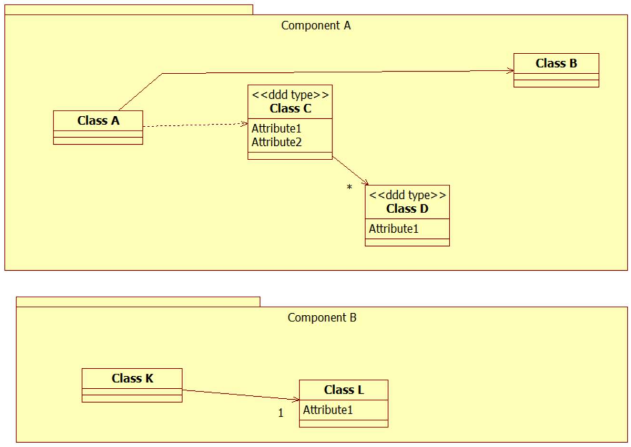
- Customers can reserve car types. For example you can reserve a Toyota Camry for a certain time.
- Customers can rent a certain car for a certain time. For example you can rent the red Toyota Camry with license plate NNG344 from 04-12-2021 till 03-01-2022.
- The system keeps track of all car reservations and rentals. Every reservation has an unique reservation number, and the system stores the start date and end date of the reservation.
- We can search and browse through the catalog of cars. For every car type we can see the brand and make of the car, its price per day, price per week and price per month and we can also see how many cars we have available for this car type. The system also knows the license plate number, year and color of every car.
- When a customer reserves a car or rents a car the system keeps track of the following customer information: first name, last name, gender, phone, email, street, city, zip, country, drivers license number, driver license expiration date, driver license country of issue.

a. (10 points) Suppose we would design this system as one monolithic system. Draw the class diagram of the domain classes of this system. In domain driven design (DDD) we learned that there are different types of domain classes. For every class in your diagram indicate

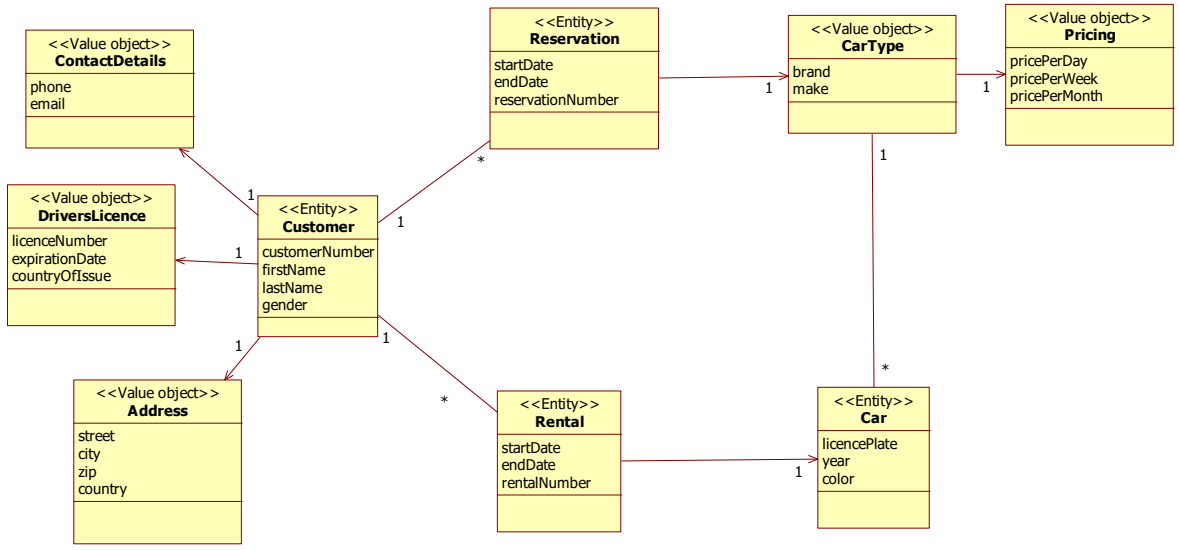
what DDD type of class this is. Only show the **domain classes, its DDD type, its attributes and its relationships with cardinality**. Do **NOT** show methods.

b. (25 points) Suppose we would design this system as a component based design. Draw in one class diagram the different components and **ALL** classes in these components (not only the domain classes). Show **ALL classes, attributes and relationships with cardinality**. Do **NOT** show methods. For every **domain** class show clearly its **DDD type**. Again, it is important to show **ALL** classes in every component.

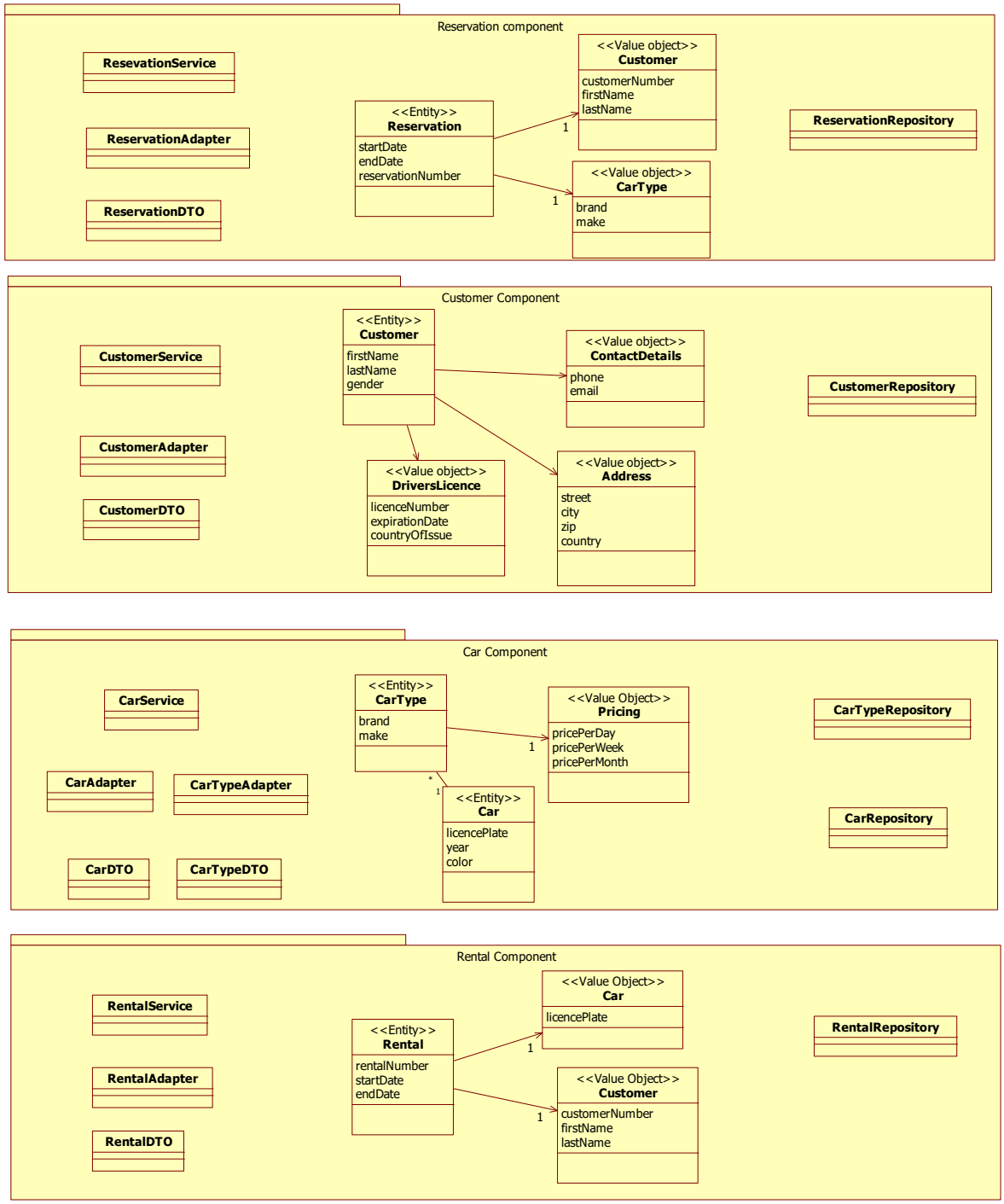
Your diagram should look similar like this:



a.



b.


Reservation component
ResevationService
ReservationAdapter
ReservationDTO
<<Entity>>
Reservation
startDate
endDate
reservationNumber
1
<<Value object>>
Customer
customerNumber
firstName
lastName
1
<<Value object>>
CarType
brand
make
ReservationRepository
Customer Component
CustomerService
CustomerAdapter
CustomerDTO
<<Entity>>
Customer
firstName
lastName
gender
<<Value object>>
ContactDetails
phone
email
<<Value object>>
DriversLicence
licenceNumber
expirationDate
countryOfIssue
<<Value object>>
Address
street
city
zip
country
CustomerRepository
Car Component
CarService
CarAdapter
CarTypeAdapter
CarDTO
CarTypeDTO
<<Entity>>
CarType
brand
make
1
*
1
<<Entity>>
Car
licencePlate
year
color
<<Value Object>>
Pricing
pricePerDay
pricePerWeek
pricePerMonth
CarTypeRepository
CarRepository
Rental Component
RentalService
RentalAdapter
RentalDTO
<<Entity>>
Rental
rentalNumber
startDate
endDate
1
<<Value Object>>
Car
licencePlate
1
<<Value Object>>
Customer
customerNumber
firstName
lastName
RentalRepository

[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

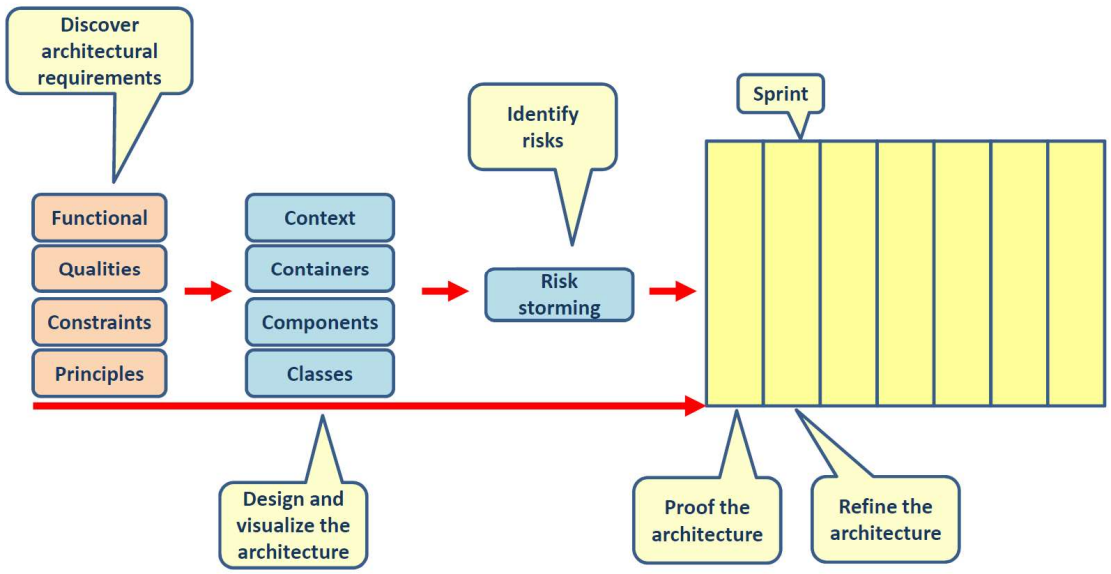
- On which part of the application should I put the focus
- Where to put your most experienced developers
- What code should be of the highest quality
- Where to apply DDD

[15 minutes]

The question is given in the attachment. **Open the PDF file in the browser.**

Type your answer in the editor on this page

Attachments
question 1.pdf



[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

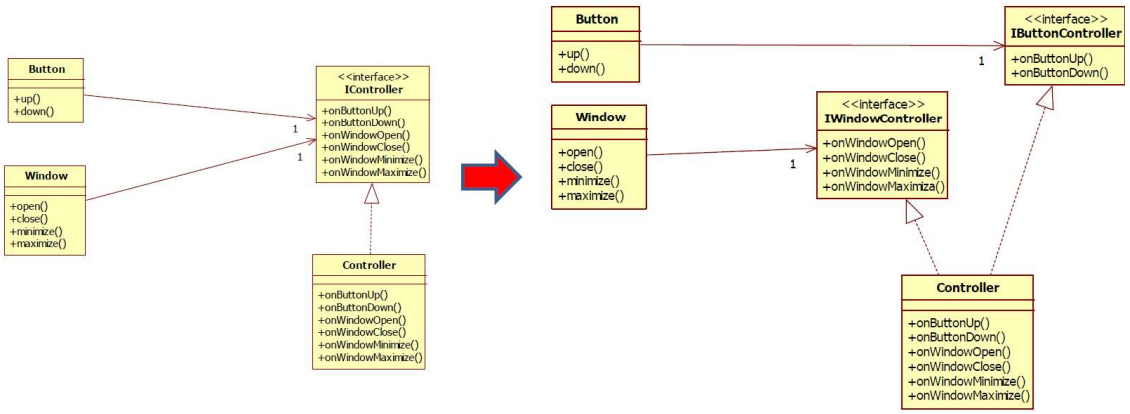
[15 minutes]

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



[10 minutes]

Give the 4 main features offered by the integration broker in a hub and spoke architecture.

1. Transport
2. Transformation
3. Routing
4. Orchestration

[15 minutes]

In this course we learned many enterprise integration patterns. We grouped these patterns into different groups like message channel, message construction, message endpoint, message routing, message transformation and message management.

a. give 2 different **message channel** integration patterns.

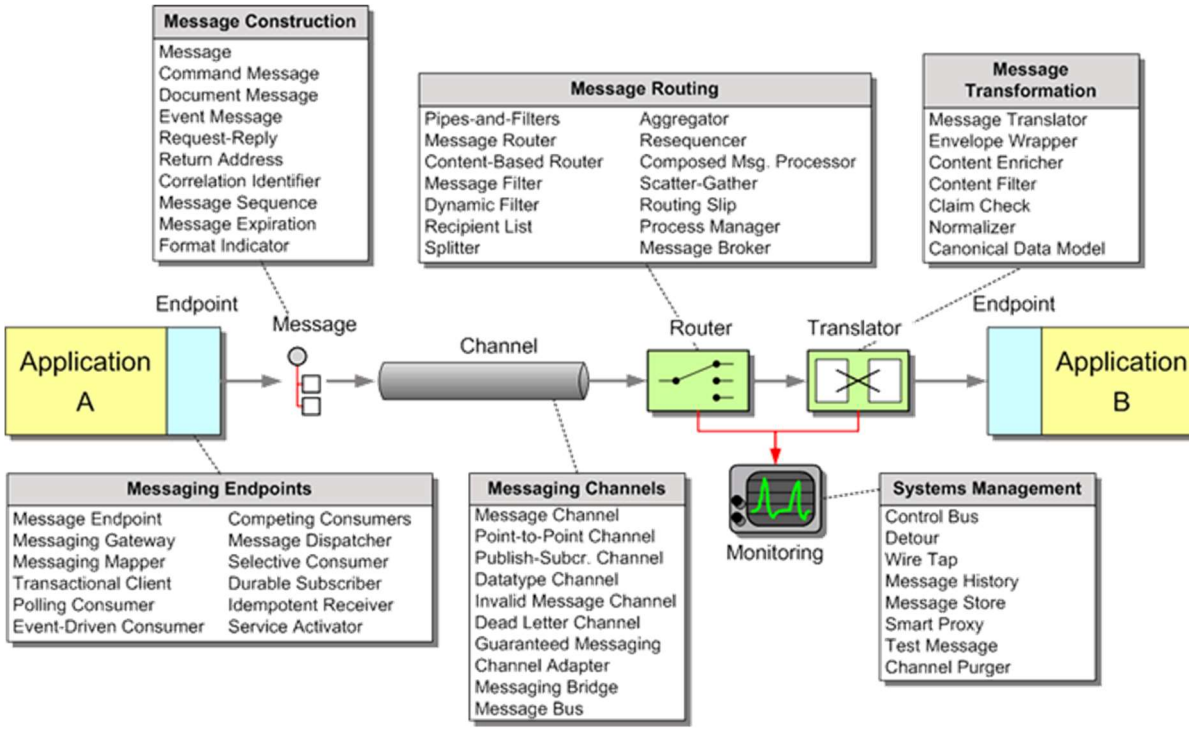
b. give 2 different **message construction** integration patterns .

c. give 2 different **message routing** integration patterns.

d. give 2 different **message endpoint** integration patterns.

e. give 2 different **message transformation** integration patterns.

f. give 2 different **message management** integration patterns.



[10 minutes]

Explain clearly the main difference(s) between a **domain service class** from domain driven design and a **service class** in the service layer.

Domain service lives in the domain layer. It contains only business logic
Service class lives in the service layer. It does not contain business logic. It acts as a façade that connects classes in the other layers.

[10 minutes]

Select all statements that are true

☐ A.

An entity object in domain driven design is stateless

☐ B.
A value object in domain driven design is stateless

☐ C.
A domain service object in domain driven design is stateless

☐ D.
An event object object in domain driven design is stateless

☐ E. An entity object in domain driven design is immutable
☐ F. A value object in domain driven design is immutable
☐ G. Non aggregate roots can hold a reference to other aggregate roots
☐ H. The aggregate root class resides in the service layer
☐ I. The domain service object in domain driven design resides in the service layer
☐ J. An aggregate root class in domain driven design is immutable

[30 minutes]

Suppose we have a 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 List<Tool> tools;
   private Customer customer;
   private Date startDate;
   private Date endDate;
...
}
```

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 (and their type) of every domain class

Your answer should look like the following example:

StudentComponent:

class Student:
int studentNumber
String firstname
String lastName
String email
Address address

class Address:
String street
String city
String zip

b. For every component, write the interface of every service class (show the return value, name of the method, attributes and its type)

Your answer should look like the following example:

interface StudentComponent:

void addStudent(int studentNumber, String firstName, String lastName, String email);
Student getStudentByNumber(int studentNumber);

a.

Customer component:

```
public class Customer{
   private int customerNumber;
   private String name;
   private String email;
   private String phone;
...
}
```

Tool component:

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

Rental component:

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

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

```
public class Customer{
   private int customerNumber;
   private String name;
   private String email;
   private String phone;
...
}
```

b.

Customer component:

```
public interface CustomerService {
   public void addCustomer(int customerNumber, String name, String
email, String phone);
   public CustomerDTO getCustomer(int customerNumber);
   public void removeCustomer(int customerNumber);

}
```

Tool component:

```
public interface RentalService {
   public void addTool(int toolNumber, String name, double price);
   public ToolDTO getTool(int toolNumber);
   public void removeTool(int toolNumber);

}
```

Rental component:

```
public interface RentalService {
   public void addRental(int rentalNumber, ToolDTO tool, CustomerDTO
customer, Date startDate, Date endDate);
   public RentalDTO getRental(int rentalNumber);
   public void removeRental(int rentalNumber);
}
```

[15 minutes]

a. Explain the difference between horizontal and vertical scaling

- horizontal scaling: add more servers
- vertical scaling: use a larger server with more resources

b. Explain the 3 ways of horizontal scaling. For every of these 3 options explain shortly how they work.

- Sharding
- Partitioning
- Replication

[15 minutes]

a. We studied the 4C model to communicate architecture. Give the names of the 4 diagrams of the 4C model.

- Context
- Container
- Component
- Class

b. We learned to quantify risk in 2 parts. In what 2 parts do we quantify risk?

Impact and probability

c. Explain the 4 steps of risk storming

Step 1. Draw the architecture diagrams

Step 2. Identify the risks individually

Step 3. Converge the risks on the diagrams

Step 4. Prioritize the risks

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);
}
```