(a) Nennen Sie mindestens zwei Gründe für den Einsatz eines Messaging Systems.

- Asynchrone Kommunikation zwischen Anwendungen
- Möglichkeit zu der zeitlichen und örtlichen Entkopplung zwischen Anwendungen

(b) Erklären Sie kurz und mit eigenen Worten den Unterschied zwischen den Messaging-Modellen „Publisher-Subscriber" und „Sender-Receiver".

Publisher-Subscriber-Model:
Die Subscribers werden in einem Topic registriert. Jeder Subscriber bekommt eine Nachricht, die ein Publisher an diesem Topic sendet(„veröffentlicht") bzw. gesendet hat.

Sender-Receiver-Modell:
Eine Nachricht wird von einem Sender an genau einen Receiver ausgeliefert und dabei in eine Warteschlange(Queue) gestellt, bis die Nachricht abgefragt wird. Bei einer hohen Last können mehrere Receivers zugeschaltet werden.

(c) Beschreiben Sie die beiden Begriffe „(enge und lose) Kopplung" und „Kohäsion" mit eigenen Worten.

Mit „Enge und lose Kopplung" versteht man die Abhängigkeiten zwischen Komponenten. Je mehr solche Abhängigkeiten vorhanden sind, desto enger ist die Kopplung. Die Hauptkategorien von Abhängigkeiten sind:

- Zeitliche Abhängigkeit
- Örtliche Abhängigkeit
- Struktur- und Implementierungsabhängigkeit
- Datenabhängigkeit

Unter Kohäsion versteht man ein Maß für die innere (logische) Abhängigkeit in einer Klasse. Je stärker der logische Zusammenhang/Abhängigkeit der in einer Klasse realisierten Aufgaben ist, desto stärker ist die Kohäsion.

Ein guter Entwurf verfolgt das Ziel eine starke Kohäsion und eine schwache Kopplung zu erreichen.

(d) Beschreiben Sie die fünf SOLID-Prinzipien anhand von selbstgewählten Beispielen. Fertigen Sie hierzu ein kleines Dokument an (max. 1 Seite), das Sie über die Plattform einreichen.

Die SOLID - Prinzipien

Single Responsibility Principle (SRP)

Dieser Grundsatz besagt: Es sollte nie mehr als einen Grund geben, eine Klasse zu wechseln. Jedes Objekt hat eine Verantwortung, die vollständig in der Klasse enthalten ist. Alle Klassenleistungen zielen darauf ab, diese Verantwortung zu gewährleisten. Solche Klassen können bei Bedarf jederzeit leicht geändert werden, da klar ist, wofür die Klasse zuständig ist und nicht. Als Beispiel ein Modul, das Aufträge verarbeitet. Wenn die Bestellung korrekt erstellt wurde, wird sie in der Datenbank gespeichert und es wird ein Brief gesendet, um die Bestellung zu bestätigen.

```
public class MySQLOrderRepository {
    public boolean save(Order order) {
        MySqlConnection connection = new MySqlConnection("database.url");
        return true;
    }
}

public class ConfirmationEmailSender {
    public void sendConfirmationEmail(Order order) {
        String name = order.getCustomerName();
        String email = order.getCustomerEmail();
    }
}

public class OrderProcessor {
    public void process(Order order){

        MySQLOrderRepository repository = new MySQLOrderRepository();
        ConfirmationEmailSender mailSender = new ConfirmationEmailSender();

        if (order.isValid() && repository.save(order)) {
            mailSender.sendConfirmationEmail(order);
        }
    }
}
```

Open Closed Principle (OCP)

Dieses Prinzip wird ausführlich wie folgt beschrieben: Software-Entitäten (Klassen, Module, Funktionen usw.) müssen für die Erweiterung geöffnet, für Änderungen jedoch geschlossen sein.

Dies bedeutet, dass es möglich sein sollte, das externe Verhalten der Klasse zu ändern, ohne physische Änderungen an der Klasse selbst vorzunehmen.

Angenommen man möchte einige Aktionen Vor- und Nach- der Bestellung durchführen. Anstatt die OrderProcessor-Klasse selbst zu ändern, erweitert man sie und findet somit eine Lösung für die Aufgabe, ohne das OCP-Prinzip zu verletzen

```
public class OrderProcessorWithPreAndPostProcessing extends OrderProcessor {

    @Override
    public void process(Order order) {
        beforeProcessing();
        super.process(order);
        afterProcessing();
    }

    private void beforeProcessing() {
        // Einige Maßnahmen vor der Auftragsabwicklung durchführen
    }

    private void afterProcessing() {
        // Einige Aktionen, nach der Auftragsabwicklung, durchführen
    }
}
```

Liskov's Substitution Principle (LSP)

Dies ist eine Variation des zuvor erwähnten Prinzips der Offenheit. Dies kann wie folgt beschrieben werden: Objekte im Programm können durch ihre Erben ersetzt werden, ohne die Eigenschaften des Programms zu ändern. Dies bedeutet, dass eine Klasse, die durch Erweitern basierend auf einer Basisklasse entwickelt wurde, ihre Methoden neu definieren muss, damit die Funktionalität aus Sicht des Clients nicht beeinträchtigt wird. Das heißt, wenn ein Entwickler Ihre Klasse erweitert und in einer Anwendung verwendet, sollte dies das erwartete Verhalten überschriebener Methoden nicht ändern.

Zum Beispiel Sollen einige Bestellungen anders, als immer validiert werden.

```
public class OrderStockValidator {

    public boolean isValid(Order order) {
        for (Item item : order.getItems()) {
            if (! item.isInStock()) {
                return false;
            }
        }
        return true;
    }
}
```

```
public class OrderStockAndPackValidator extends OrderStockValidator {

    @Override
    public boolean isValid(Order order) {
        for (Item item : order.getItems()) {
            if ( !item.isInStock() || !item.isPacked() ){
                throw new IllegalStateException(
                    String.format("Order %d is not valid!", order.getId())
                );
            }
        }
        return true;
    }
}
```

Interface Segregation Principle (ISP)

Es zeichnet sich durch die folgende Aussage aus: Klassen sollten nicht gezwungen werden, Methoden zu implementieren, die sie nicht verwenden werden.

Das Prinzip der Schnittstellentrennung legt nahe, dass zu „dicke" Schnittstellen in kleinere und spezifischere Schnittstellen unterteilt werden sollten, damit Clients kleiner Schnittstellen nur über die in der Arbeit benötigten Methoden Bescheid wissen. Wenn Sie die Schnittstellenmethode ändern, sollten Clients, die diese Methode nicht verwenden, sich daher nicht ändern.

Dependency Inversion Principle (DIP)

Dieses SOLID-Prinzip in Java wird wie folgt beschrieben: Abhängigkeiten innerhalb des Systems werden auf der Basis von Abstraktionen aufgebaut. Module der obersten Ebene sind unabhängig von Modulen der unteren Ebene. Abstraktionen

sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.
Der Code muss so gestaltet sein, dass die verschiedenen Module in sich geschlossen und durch Abstraktion miteinander verbunden sind. Als Beispiel ist die unter gezeigte Implementierung der Schnittstellen und das Constuctor-Injection in OrderProzessor - Klasse.

```java
public interface MailSender {
    void sendConfirmationEmail(Order order);
}

public interface OrderRepository {
    boolean save(Order order);
}

public class ConfirmationEmailSender implements MailSender {

    @Override
    public void sendConfirmationEmail(Order order) {
        String name = order.getCustomerName();
        String email = order.getCustomerEmail();

        // Hier kommt Logik für das Senden von Email
    }

}

public class MySQLOrderRepository implements OrderRepository {

    @Override
    public boolean save(Order order) {
        MySqlConnection connection = new MySqlConnection("database.url");

        // Hier kommt das Logik zum speichern in Datenbank

        return true;
    }
}
```