Analyse des eigenen Codes

Identifizieren Sie mindestens 3 Stellen i ihrem Code, die von funktionalen Konzepten profitieren würden.

1. Validierungsmethoden

Aktueller Code

Die Validierungsmethoden (*validateEmail*, *validatePhone*, *validatePassword*) sind als separate, imperative Methoden implementiert, die direkt Ausnahmen werfen. Jede Methode arbeitet unabhängig, was zu duplizierter Logik führen könnte, falls weitere Validierungen hinzugefügt werden.

```
private void validateEmail(String email) throws InvalidInputException {  Orock237
    if (userRepository.findByEmail(email) != null) {
        throw new InvalidInputException("E-Mail bereits in Gebrauch");
    }
    if (!email.matches( regex: "^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]{2,}$")) {
        throw new InvalidInputException("Ungültige E-Mail Adresse");
    }
}

private void validatePhone(String phone) throws InvalidInputException {  Orock237
    if (!phone.matches( regex: "^(\\+\\d{1,3}[- ]?)?([\\d, /-]*\\d){7,15}$")) {
        throw new InvalidInputException("Ungültige Telefonnummer");
    }
}

private void validatePassword(String password) throws  InvalidInputException {  Orock237
    if (!password.matches( regex: "^(?=.*[A-Za-z])(?=.*\\d)[A-Za-z\\d]{8,20}$")) {
        throw new InvalidInputException("Ungültiges Passwort");
    }
}
```

Geplante Überarbeitung

- Anwendung funktionaler Pipelines, um die Validierungen zu kombinieren.
- Implementierung einer generischen Methode, die Validierungsregeln als Lambda-Funktionen akzeptiert und sequenziell ausführt.

Erwartete Verbesserungen

- Wiederverwendbarkeit: Validierungslogik ist generisch und modular.
- Wartbarkeit: Neue Regeln können leicht hinzugefügt werden.
- Lesbarkeit: Validierungen sind deklarativer und kompakter.

2. Mapping von Rollen zu Autoritäten

Aktueller Code

Die Methode *mapRolesToAuthorities* verwendet imperatives Mapping mit Streams, um Rollen in GrantedAuthority-Objekte zu transformieren. Die Funktionalität ist einfach, aber das Muster könnte optimiert werden.

```
private Collection<? extends GrantedAuthority> mapRolesToAuthorities(Collection<Role> roles){  1 usage  Orock237
    return roles.stream().map( Role role -> new SimpleGrantedAuthority(role.getName())).collect(Collectors.toList());
}
```

Geplante Überarbeitung

- Anwendung von methodenreferenzbasierten Transformationen oder einen reinen funktionalen Ansatz, um die Zuordnung deklarativer zu gestalten.

Erwartete Verbesserungen

- Klarheit: Die Transformation ist deklarativer.
- Robustheit: Reduziert potenzielle Fehlerquellen.

3. Validierungsmethoden für einzelne Felder (Complaint-Location)

Aktueller Code

Die Validierungsmethoden sind wiederholend und spezifisch für jedes Feld implementiert. Dies führt zu Boilerplate-Code und erschwert die Wartung.

```
private boolean isValidStreet(String street) {  cveti
    return street != null && STREET_PATTERN.matcher(street.trim()).matches();
}

private boolean isValidHouseNumber(String houseNumber) {  cveti
    return houseNumber != null && HOUSE_NUMBER_PATTERN.matcher(houseNumber.trim()).matches();
}

private boolean isValidPostalCode(String postalCode) {  cveti
    return postalCode != null && POSTAL_CODE_PATTERN.matcher(postalCode.trim()).matches();
}

private boolean isValidCity(String city) {  cveti
    return city != null && CITY_PATTERN.matcher(city.trim()).matches();
}
```

Geplante Überarbeitung

- Implementierung einer generischen, funktionalen Methode, die ein Feld und das entsprechende Validierungsmuster als Parameter akzeptiert.
- Nutzung von Lambda-Ausdrücken oder Method-References, um die Validierungslogik zu kapseln.

Erwartete Verbesserungen

- Wiederverwendbarkeit: Die Methode isValidField kann für beliebige Felder und Muster verwendet werden.
- Kürzerer Code: Reduziert redundanten Code und macht die Klasse leichter lesbar.
- Flexibilität: Änderungen an der Validierungslogik sind einfacher umzusetzen.

4. Konstruktorlogik für Validierung und Initialisierung

Aktueller Code

Der Konstruktor in der *Location*-Klasse enthält die Validierungslogik für alle Felder und führt die Initialisierung direkt durch. Dies könnte modularisiert werden, um die Lesbarkeit zu verbessern.

```
public Location(String street, String houseNumber, String postalCode, String city) {
    if (!isValidStreet(street)) {
        throw new IllegalArgumentException("Invalid street name.");
    }
    if (!isValidHouseNumber(houseNumber)) {
        throw new IllegalArgumentException("Invalid house number.");
    }
    if (!isValidPostalCode(postalCode)) {
        throw new IllegalArgumentException("Invalid postal code.");
    }
    if (!isValidCity(city)) {
        throw new IllegalArgumentException("Invalid city name.");
    }

    this.street = street.trim();
    this.houseNumber = houseNumber.trim();
    this.postalCode = postalCode.trim();
    this.city = city.trim();
}
```

Geplante Überarbeitung

- Extrahiere die Validierungslogik in eine funktionale Pipeline.
- Nutze eine Hilfsfunktion, um die Validierung und Initialisierung zu kombinieren.

Erwartete Verbesserungen

- Konsistenz: Die Validierungslogik wird in einer einzigen Methode zentralisiert.
- Wiederverwendbarkeit: Die Methode *validateAndTrim* kann für alle Felder genutzt werden.
- Lesbarkeit: Der Konstruktor ist kompakter und besser verständlich.

Implementierung funktionaler Konzepte

Setzen Sie die identifizierten Verbesserungen um.

1. Validierungsmethode

Optimierte Validierungsmethode:

```
 * @param input The input string to be validated.
 * @param rules A list of functions representing the validation rules. Each function returns an Optional<String> with an error message if the rule fails.
 * @throws InvalidInputException if any rule fails, with the first error message found.
 */
private void validateInput(String input, List<Function<String, Optional<String>>> rules) throws InvalidInputException {  3 usages   Orock237 *
    Optional<String> errorMessage = rules.stream() Stream<Function<...>>
            .map( Function<String, Optional<...>> rule -> rule.apply(input)) Stream<Optional<...>>
            .filter(Optional::isPresent)
            .map(Optional::get) Stream<String>
            .findFirst();

    if (errorMessage.isPresent()) {
        throw new InvalidInputException(errorMessage.get());
    }
}
```

Verwendung für die Validierung von E-Mail Adressen

```
this.validateInput(signUpDto.getEmail(), List.of(
    String e -> e.matches( regex: "^[\\w.-]+@[\\w.-]+\\.[a-zA-Z]{2,}$")
            ? Optional.empty()
            : Optional.of( value: "Ungültige E-Mail Adresse"),
    String e -> userRepository.findByEmail(e) == null
            ? Optional.empty()
            : Optional.of( value: "E-Mail bereits in Gebrauch")
));
```

2. Mapping von Rollen zu Autoritäten

Optimierte Version der *mapRolesToAuthorities* Methode:

```
 * @param roles the collection of Role objects to be mapped
 * @return a collection of GrantedAuthority objects corresponding to the roles
 */
private Collection<? extends GrantedAuthority> mapRolesToAuthorities(Collection<Role> roles) {
    return roles.stream() Stream<Role>
            .map(Role::getName) Stream<String>
            .map(SimpleGrantedAuthority::new) Stream<SimpleGrantedAuthority>
            .collect(Collectors.toList());
}
```

3. Validierungsmethoden für einzelne Felder (Complaint-Location)

Optimierte Validierungsmethoden

```java
private boolean isValidField(String value, Pattern pattern) {  new *
    return value != null && pattern.matcher(value.trim()).matches();
}

private boolean isValidStreet(String street) {  cveti *
    return isValidField(street, STREET_PATTERN);
}

private boolean isValidHouseNumber(String houseNumber) {  cveti *
    return isValidField(houseNumber, HOUSE_NUMBER_PATTERN);
}

private boolean isValidPostalCode(String postalCode) {  cveti *
    return isValidField(postalCode, POSTAL_CODE_PATTERN);
}

private boolean isValidCity(String city) {  cveti *
    return isValidField(city, CITY_PATTERN);
}
```

4. Konstruktorlogik für Validierung und Initialisierung

Optimierter Konstruktor:

```java
public Location(String street, String houseNumber, String postalCode, String city) {  13 usages  cveti *
    this.street = validateAndTrim(street, STREET_PATTERN,  errorMessage: "Invalid street name.");
    this.houseNumber = validateAndTrim(houseNumber, HOUSE_NUMBER_PATTERN,  errorMessage: "Invalid house number.");
    this.postalCode = validateAndTrim(postalCode, POSTAL_CODE_PATTERN,  errorMessage: "Invalid postal code.");
    this.city = validateAndTrim(city, CITY_PATTERN,  errorMessage: "Invalid city name.");
}
```

```java
private String validateAndTrim(String value, Pattern pattern, String errorMessage) {  4 usages  new *
    if (!isValidField(value, pattern)) {
        throw new IllegalArgumentException(errorMessage);
    }
    return value.trim();
}
```

Collection Processing

Implementierte Operationen:

Aggregation/Reduktion - FIlterung

```
 * @param input The input string to be validated.
 * @param rules A list of functions representing the validation rules. Each function returns an Optional<String> with an
 * @throws InvalidInputException if any rule fails, with the first error message found.
 */
private void validateInput(String input, List<Function<String, Optional<String>>> rules) throws InvalidInputException {
    Optional<String> errorMessage = rules.stream() Stream<Function<...>>
            .map( Function<String, Optional<...>> rule -> rule.apply(input)) Stream<Optional<...>>
            .filter(Optional::isPresent)
            .map(Optional::get) Stream<String>
            .findFirst();

    if (errorMessage.isPresent()) {
        throw new InvalidInputException(errorMessage.get());
    }
}
```

Transformation/Mapping

```
 * @return a list of {@code ComplaintDto} objects representing all complaints
 */
public List<ComplaintDto> findAllComplaints() {  4 usages   cveti
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm");
    return complaintRepository.findAll().stream() Stream<Complaint>
            .map( Complaint complaint -> new ComplaintDto(
                    complaint.getTitle(),
                    complaint.getDescription(),
                    complaint.getLocation().toString(),
                    complaint.getCreatedAt().format(formatter))) Stream<ComplaintDto>
            .collect(Collectors.toList());
}
```

Reflexion

Vorteile der funktionalen Implementierung

- **Klarheit und Modularität:** Die Verwendung von funktionalen Interfaces wie *Function* und *Optional* ermöglicht eine klare Trennung von Validierungslogik. Jede Regel wird als eigenständige Funktion definiert, wodurch die Wiederverwendbarkeit erhöht wird.
- **Fehlervermeidung**: Durch den Einsatz von *Optional* werden NullPointerExceptions minimiert, da explizit signalisiert wird, ob ein Wert vorhanden ist.
- **Testbarkeit**: Funktionale Implementierungen sind leichter zu testen, da jede Regel unabhängig getestet werden kann.
- **Lesbarkeit**: Die Stream-API reduziert Boilerplate-Code, was den Code kompakter und leichter verständlich macht.

Nachteile der funktionalen Implementierung

- **Komplexität für Einsteiger:** Für Entwickler, die nicht mit funktionalen Konzepten vertraut sind, können Streams und Lambdas anfangs schwer zu verstehen sein.
- **Performance**: Die intensive Nutzung von Streams und Optional kann in hochperformanten Anwendungen leichte Overheads verursachen.

Eingesetzte Technologien

Copilot unterstützte bei der Refaktorierung von Code, insbesondere bei der Implementierung von Streams und Optional. Es schlug effiziente Lösungen vor, sparte Zeit und reduzierte Fehler.

Auswirkungen auf Codequalität und Lesbarkeit

Die Codequalität wurde durch die funktionale Implementierung erheblich verbessert. Der Code wurde modularer, prägnanter und leichter wartbar. Die Lesbarkeit profitierte durch die klare Struktur der Validierungsregeln.

Lessons Learned

- Funktionale Konzepte sind ein mächtiges Werkzeug, sollten jedoch schrittweise eingeführt werden.
- Tools wie Copilot können die Effizienz und Qualität erheblich steigern.