SWED Excercise 4

Aufgabe 2

**Definitionen:**

* **Afferente Kopplung (Ca - Afferent Coupling):** Die Anzahl der Klassen *außerhalb* dieses Pakets (oder in diesem Fall, anderer Klassen), die von Klassen *innerhalb* dieses Pakets (oder dieser spezifischen Klasse) abhängen. Es misst die Verantwortung einer Klasse.
* **Efferente Kopplung (Ce - Efferent Coupling):** Die Anzahl der Klassen *außerhalb* dieses Pakets (oder anderer Klassen), von denen Klassen *innerhalb* dieses Pakets (oder dieser spezifischen Klasse) abhängen. Es misst die Abhängigkeit einer Klasse von anderen.
* **Instabilität (I):** Ein Maß dafür, wie anfällig eine Klasse für Änderungen ist. Die Formel lautet: $ I = \frac{Ce}{Ca + Ce} $.
  + Ein Wert von I=0 bedeutet, dass die Klasse maximal stabil ist (viele andere hängen von ihr ab, sie hängt von wenigen ab).
  + Ein Wert von I=1 bedeutet, dass die Klasse maximal instabil ist (wenige hängen von ihr ab, sie hängt von vielen ab).
  + Wenn Ca + Ce = 0, ist I undefiniert oder wird oft als 0 interpretiert (keine externen Kopplungen).

**Analyse der Abhängigkeiten basierend auf dem UML-Diagramm:**

1. **User:**
   * Hängt ab von (Ce):
     + Subscription (Attribut List<Subscription>, Rückgabetyp von registerForUpdates())
     + Website (Parameter von registerForUpdates())
     + NotificationPreferences (Parameter von registerForUpdates())
   * **Ce (User) = 3**
   * Wird verwendet von (Ca):
     + Subscription (Rückgabetyp von getUser(), implizite Assoziation)
   * **Ca (User) = 1**
   * **I (User)** = 3 / (1 + 3) = 3 / 4 = **0.75**
2. **Subscription:**
   * Hängt ab von (Ce):
     + User (Rückgabetyp von getUser(), implizite Assoziation)
     + Website (Rückgabetyp von getWebsite(), implizite Assoziation)
     + NotificationPreferences (Parameter und Rückgabetyp von setPreferences()/getPreferences(), implizite Assoziation)
     + Notification (gerichtete Assoziation von Subscription zu Notification, impliziert, dass Subscription Notifications erstellt oder verwendet)
   * **Ce (Subscription) = 4**
   * Wird verwendet von (Ca):
     + User (Attribut List<Subscription> in User, Rückgabetyp von registerForUpdates() in User)
   * **Ca (Subscription) = 1**
   * **I (Subscription)** = 4 / (1 + 4) = 4 / 5 = **0.8**
3. **Website:**
   * Hängt ab von (Ce):
     + Keine Abhängigkeiten zu *anderen dargestellten Klassen* (DateTime ist ein Basistyp).
   * **Ce (Website) = 0**
   * Wird verwendet von (Ca):
     + User (Parameter von registerForUpdates())
     + Subscription (Rückgabetyp von getWebsite(), implizite Assoziation)
   * **Ca (Website) = 2**
   * **I (Website)** = 0 / (2 + 0) = 0 / 2 = **0**
4. **NotificationPreferences:**
   * Hängt ab von (Ce):
     + Keine Abhängigkeiten zu *anderen dargestellten Klassen*.
   * **Ce (NotificationPreferences) = 0**
   * Wird verwendet von (Ca):
     + User (Parameter von registerForUpdates())
     + Subscription (Parameter und Rückgabetyp von setPreferences()/getPreferences(), implizite Assoziation)
   * **Ca (NotificationPreferences) = 2**
   * **I (NotificationPreferences)** = 0 / (2 + 0) = 0 / 2 = **0**
5. **Notification:**
   * Hängt ab von (Ce):
     + Keine Abhängigkeiten zu *anderen dargestellten Klassen* (DateTime ist ein Basistyp).
   * **Ce (Notification) = 0**
   * Wird verwendet von (Ca):
     + Subscription (gerichtete Assoziation von Subscription zu Notification)
   * **Ca (Notification) = 1**
   * **I (Notification)** = 0 / (1 + 0) = 0 / 1 = **0**

**Zusammenfassung der Metriken:**

|  |  |  |  |
| --- | --- | --- | --- |
| Klasse | Ca (Afferent) | Ce (Efferent) | Instabilität (I) |
| User | 1 | 3 | 0.75 |
| Subscription | 1 | 4 | 0.8 |
| Website | 2 | 0 | 0 |
| NotificationPreferences | 2 | 0 | 0 |
| Notification | 1 | 0 | 0 |

**Interpretation:**

* User und Subscription sind relativ instabil (I > 0.5), was bedeutet, dass sie von mehreren anderen Klassen abhängen und Änderungen in diesen abhängigen Klassen sie wahrscheinlich beeinflussen werden. Subscription ist die instabilste Klasse.
* Website, NotificationPreferences und Notification sind maximal stabil (I = 0). Sie hängen von keiner anderen dargestellten Klasse ab, während andere Klassen von ihnen abhängen. Änderungen in diesen Klassen können sich auf andere auswirken, aber sie selbst sind weniger anfällig für Änderungen in anderen Teilen des Systems (bezogen auf die hier dargestellten Klassen).

⁂

Aufgabe 3

Als Basis-Paketnamen nehmen wir an: com.firmenname.websiteobserver (ersetzen Sie com.firmenname mit Ihrem tatsächlichen Reverse-Domain-Namen).

**Vorgeschlagene Paketstruktur:**

com.firmenname.websiteobserver  
├── model  
│ ├── user  
│ │ └── User.java  
│ ├── subscription  
│ │ ├── Subscription.java  
│ │ └── NotificationPreferences.java  
│ ├── website  
│ │ └── Website.java  
│ └── notification  
│ └── Notification.java  
└── service // (Optional, für zukünftige Service-Klassen)  
└── util // (Optional, für Hilfsklassen)

**Detailliertere Aufschlüsselung und Begründung:**

1. **com.firmenname.websiteobserver.model**: Dieses Hauptpaket enthält alle Domänenmodelle bzw. Entitätsklassen.
   * **model.user**:
     + User.java: Repräsentiert die Benutzer des Systems. Es ist logisch, dies in einem eigenen Unterpaket zu haben, da Benutzer oft eine zentrale Entität in Anwendungen sind.
   * **model.subscription**:
     + Subscription.java: Stellt ein Abonnement dar, das einen Benutzer mit einer Webseite und Benachrichtigungseinstellungen verbindet.
     + NotificationPreferences.java: Definiert die Einstellungen für Benachrichtigungen, die eng mit einem Subscription verbunden sind. Da NotificationPreferences primär im Kontext eines Subscription existiert und konfiguriert wird, ist es sinnvoll, sie zusammen zu gruppieren.
   * **model.website**:
     + Website.java: Repräsentiert die Webseiten, die überwacht werden. Diese Klasse hat eine eigene Identität und kann unabhängig von Abonnements existieren (z.B. im System bekannt sein, bevor jemand sie abonniert).
   * **model.notification**:
     + Notification.java: Repräsentiert eine einzelne Benachrichtigung, die an einen Benutzer gesendet wird. Obwohl sie aus einem Subscription resultiert, ist die Notification selbst ein eigenständiges Konzept.

**Vorteile dieser Struktur:**

* **Hohe Kohäsion innerhalb der Pakete:** Klassen, die eng zusammenarbeiten oder konzeptionell verwandt sind (wie Subscription und NotificationPreferences), befinden sich im selben Paket.
* **Lose Kopplung zwischen den Paketen (angestrebt):** Abhängigkeiten zwischen den Paketen spiegeln die logischen Abhängigkeiten des Systems wider.
* **Klarheit und Verständlichkeit:** Die Paketnamen geben Aufschluss über den Inhalt und die Verantwortlichkeit der darin enthaltenen Klassen.
* **Skalierbarkeit:** Diese Struktur lässt sich gut erweitern. Wenn beispielsweise neue Funktionalitäten rund um Benutzer hinzukommen (z.B. Authentifizierung, Autorisierung), können diese im user-Paket oder in dessen Unterpaketen platziert werden. Service-Klassen, die die Logik orchestrieren, könnten in ein separates service-Paket auf oberster Ebene (com.firmenname.websiteobserver.service) eingefügt werden.

**Import-Anweisungen (Beispiele):**

In User.java (im Paket com.firmenname.websiteobserver.model.user):

import com.firmenname.websiteobserver.model.subscription.Subscription;  
import com.firmenname.websiteobserver.model.subscription.NotificationPreferences;  
import com.firmenname.websiteobserver.model.website.Website;

In Subscription.java (im Paket com.firmenname.websiteobserver.model.subscription):

import com.firmenname.websiteobserver.model.user.User;  
import com.firmenname.websiteobserver.model.website.Website;  
// NotificationPreferences ist im selben Paket, daher kein expliziter Import nötig  
// import com.firmenname.websiteobserver.model.notification.Notification; // Falls benötigt

Um die Kopplung zwischen den vorgeschlagenen Paketen (model.user, model.subscription, model.website, model.notification) zu reduzieren, können verschiedene Designprinzipien und Muster angewendet werden. Eine geringere Kopplung führt zu einem modulareren, flexibleren und einfacher zu wartenden System [[1]](file:///C:\\Users\\omidf\\Downloads\\Name%20options%20to%20reduce%20coupling%20between%20your%20packa.docx" \l "fn1).

Hier sind einige Optionen zur Reduzierung der Kopplung zwischen Ihren Paketen:

1. **Verwendung von Interfaces (Programmierung gegen Abstraktionen):**
   * **Prinzip:** Klassen sollten von Abstraktionen (Interfaces) abhängen, nicht von konkreten Implementierungen [[2]](file:///C:\\Users\\omidf\\Downloads\\Name%20options%20to%20reduce%20coupling%20between%20your%20packa.docx" \l "fn2)[[3]](file:///C:\\Users\\omidf\\Downloads\\Name%20options%20to%20reduce%20coupling%20between%20your%20packa.docx" \l "fn3). Dies ist ein Kernaspekt des Dependency Inversion Principle (DIP), einem Teil der SOLID-Prinzipien [[4]](file:///C:\\Users\\omidf\\Downloads\\Name%20options%20to%20reduce%20coupling%20between%20your%20packa.docx" \l "fn4).
   * **Anwendung:**
     + Das model.user-Paket könnte ein Interface wie ISubscriptionService definieren, das Methoden wie registerForUpdates, modifySubscription und cancelSubscription enthält.
     + Eine Implementierung dieses Interfaces (SubscriptionServiceImpl) würde dann im model.subscription-Paket (oder besser noch in einem übergeordneten service-Paket) liegen.
     + Die User-Klasse würde dann eine Referenz auf ISubscriptionService halten, anstatt direkt mit der konkreten Subscription-Klasse für diese Operationen zu interagieren.
     + Ähnlich könnte das model.subscription-Paket ein INotificationCreator Interface definieren, das vom model.notification-Paket implementiert wird.
   * **Vorteil:** Das model.user-Paket hängt nicht mehr direkt vom model.subscription-Paket für die Erstellung und Verwaltung von Abonnements ab, sondern nur von einem Vertrag (Interface).
2. **Einführung einer Service-Schicht (Service Layer):**
   * **Prinzip:** Eine separate Schicht von Service-Klassen handhabt die Anwendungslogik und orchestriert die Interaktionen zwischen Domänenobjekten.
   * **Anwendung:**
     + Erstellen Sie ein neues Paket, z.B. com.firmenname.websiteobserver.service.
     + Darin könnten Services wie UserService, SubscriptionService, WebsiteService und NotificationService liegen.
     + Beispiel: Wenn ein User sich für Updates registriert, würde die Anforderung an den SubscriptionService gehen. Dieser Service würde dann mit den relevanten Klassen aus den model.user, model.website und model.subscription-Paketen interagieren, um die Operation auszuführen.
   * **Vorteil:** Die Domänenmodelle (User, Website etc.) müssen sich nicht direkt kennen. Die Kopplung wird in die Service-Schicht verlagert, die speziell dafür ausgelegt ist, Abhängigkeiten zu managen. Die Modellpakete selbst werden entkoppelt.
3. **Ereignisgesteuerte Architektur (Event-Driven Architecture):**
   * **Prinzip:** Komponenten kommunizieren über das Senden und Empfangen von Ereignissen, ohne direkte Kenntnis voneinander zu haben [[5]](file:///C:\\Users\\omidf\\Downloads\\Name%20options%20to%20reduce%20coupling%20between%20your%20packa.docx" \l "fn5).
   * **Anwendung:**
     + Wenn die Website-Klasse (im model.website-Paket) eine Änderung feststellt (checkForUpdates() liefert true), könnte sie ein WebsiteUpdateDetectedEvent veröffentlichen.
     + Ein SubscriptionService oder ein dedizierter Listener (z.B. im model.subscription-Paket oder einem service-Paket) würde auf dieses Ereignis lauschen. Bei Empfang würde es die relevanten Subscription-Objekte identifizieren und dann Notification-Objekte (ggf. über einen NotificationService im model.notification-Paket) erstellen und versenden.
   * **Vorteil:** Das model.website-Paket muss nichts über Subscriptions oder Notifications wissen. Es meldet nur ein Ereignis. Ebenso muss das model.subscription-Paket nicht aktiv Websites pollen, sondern reagiert auf Ereignisse. Dies reduziert die Kopplung erheblich.
4. **Mediator-Muster:**
   * **Prinzip:** Ein Mediator-Objekt kapselt, wie eine Reihe von Objekten interagieren. Objekte kommunizieren nicht direkt miteinander, sondern über den Mediator.
   * **Anwendung:** Ein SubscriptionMediator könnte die Interaktionen zwischen User, Website und NotificationPreferences bei der Erstellung oder Änderung eines Subscription-Objekts koordinieren.
   * **Vorteil:** Reduziert direkte Abhängigkeiten zwischen den kollaborierenden Objekten/Paketen, indem die Logik in den Mediator ausgelagert wird.
5. **Tell, Don't Ask-Prinzip:**
   * **Prinzip:** Anstatt den Zustand eines Objekts abzufragen und dann basierend darauf Entscheidungen außerhalb dieses Objekts zu treffen, sollte das Objekt selbst eine Operation anbieten, die die Aktion durchführt [[5]](file:///C:\\Users\\omidf\\Downloads\\Name%20options%20to%20reduce%20coupling%20between%20your%20packa.docx" \l "fn5).
   * **Anwendung:**
     + Anstatt dass eine externe Klasse die List<Subscription> vom User holt, durch diese iteriert und dann eine einzelne Subscription modifiziert, könnte die User-Klasse eine Methode wie modifySpecificSubscription(subscriptionId, newPreferences) anbieten, die die Logik intern kapselt.
   * **Vorteil:** Reduziert die Notwendigkeit für andere Pakete, die interne Struktur (z.B. die List<Subscription>) des User-Objekts zu kennen und davon abhängig zu sein.
6. **Dependency Injection (DI):**
   * **Prinzip:** Abhängigkeiten eines Objekts werden ihm von außen bereitgestellt, anstatt dass das Objekt sie selbst erstellt [[2]](file:///C:\\Users\\omidf\\Downloads\\Name%20options%20to%20reduce%20coupling%20between%20your%20packa.docx" \l "fn2).
   * **Anwendung:** Kombiniert mit Interfaces: Die User-Klasse würde eine Instanz von ISubscriptionService über ihren Konstruktor oder eine Setter-Methode erhalten, anstatt sie selbst zu erzeugen. Ein DI-Framework oder manuelles Wiring in einer übergeordneten Schicht (z.B. main-Methode oder Service-Schicht) würde die konkrete Implementierung bereitstellen.
   * **Vorteil:** Macht Klassen unabhängiger davon, wie ihre Abhängigkeiten erstellt werden, und erleichtert das Austauschen von Implementierungen (z.B. für Tests).
7. **Nutzung von IDs statt vollständigen Objekten für Parameter/Rückgaben zwischen Paketen:**
   * **Prinzip:** Wenn ein Paket nicht den gesamten Zustand eines Objekts aus einem anderen Paket benötigt, übergeben Sie nur die ID oder relevante primitive Daten.
   * **Anwendung:** Wenn das model.notification-Paket eine Benachrichtigung erstellen soll, benötigt es vielleicht nur die userID und websiteUrl, nicht die vollständigen User- und Website-Objekte.
   * **Vorteil:** Reduziert die Datenmenge, die zwischen Paketen ausgetauscht wird, und die Abhängigkeit von der genauen Struktur der Objekte in anderen Paketen.

Durch die Anwendung einer oder mehrerer dieser Techniken kann die Kopplung zwischen den Paketen signifikant reduziert werden, was zu einem robusteren und wartbareren System führt. Oft ist eine Kombination dieser Ansätze am effektivsten [[4]](file:///C:\\Users\\omidf\\Downloads\\Name%20options%20to%20reduce%20coupling%20between%20your%20packa.docx" \l "fn4).