**2. Metriken: Ca​, Ce​ und I**

* **Efferente Kopplung (Ce​)**: Anzahl der Klassen, von denen *diese Klasse* direkt abhängt (durch Attribute, Methodenparameter, Rückgabetypen, Instanziierung, Vererbung – hier keine Vererbung).
* **Afferente Kopplung (Ca​)**: Anzahl der Klassen, die von *dieser Klasse* direkt abhängen.
* **Instabilität (I)**: I=Ca​+Ce​Ce​​. Der Wert liegt zwischen 0 (maximal stabil) und 1 (maximal instabil).

**Abhängigkeitsanalyse:**

* **User**:
  + Hängt ab von: Subscription (Attribut List<Subscription>).
  + Ce​(User)=1
* **Subscription**:
  + Hängt ab von: Website (Attribut website).
  + Ce​(Subscription)=1
* **Website**:
  + Hängt ab von: Notification (instanziiert in checkForUpdates), NotificationService (Attribut und Nutzung in checkForUpdates).
  + Ce​(Website)=2
* **Monitor**:
  + Hängt ab von: Website (Attribut List<Website>, ruft Methoden auf Website auf).
  + Ce​(Monitor)=1
* **Notification**:
  + Hängt ab von: Keiner anderen Klasse im Diagramm.
  + Ce​(Notification)=0
* **NotificationService**:
  + Hängt ab von: Notification (Parameter in sendNotification).
  + Ce​(NotificationService)=1

**Berechnung von Ca​:**

* **User**:
  + Wird verwendet von: Keiner anderen Klasse.
  + Ca​(User)=0
* **Subscription**:
  + Wird verwendet von: User.
  + Ca​(Subscription)=1
* **Website**:
  + Wird verwendet von: Subscription, Monitor.
  + Ca​(Website)=2
* **Monitor**:
  + Wird verwendet von: Keiner anderen Klasse (in diesem Modell; in einer Hauptanwendung würde er instanziiert und genutzt).
  + Ca​(Monitor)=0
* **Notification**:
  + Wird verwendet von: Website, NotificationService.
  + Ca​(Notification)=2
* **NotificationService**:
  + Wird verwendet von: Website.
  + Ca​(NotificationService)=1

**Berechnung der Instabilität (I):**

| **Klasse** | **Ce​** | **Ca​** | **Ca​+Ce​** | **I=Ca​+Ce​Ce​​** | **Interpretation** |
| --- | --- | --- | --- | --- | --- |
| User | 1 | 0 | 1 | 1/1=1 | Maximal instabil |
| Subscription | 1 | 1 | 2 | 1/2=0.5 | Ausgeglichen |
| Website | 2 | 2 | 4 | 2/4=0.5 | Ausgeglichen |
| Monitor | 1 | 0 | 1 | 1/1=1 | Maximal instabil |
| Notification | 0 | 2 | 2 | 0/2=0 | Maximal stabil |
| NotificationService | 1 | 1 | 2 | 1/2=0.5 | Ausgeglichen |

* **User, Monitor (I=1)**: Diese Klassen sind sehr instabil. Sie hängen von anderen ab, aber keine anderen Klassen (innerhalb dieses Modells) hängen von ihnen ab. Änderungen in Subscription oder Website können Änderungen in User bzw. Monitor erfordern.
* **Notification (I=0)**: Diese Klasse ist sehr stabil. Sie hängt von keiner anderen Klasse ab, aber andere Klassen (Website, NotificationService) hängen von ihr ab. Änderungen an Notification würden wahrscheinlich Änderungen in den abhängigen Klassen erfordern.
* **Subscription, Website, NotificationService (I=0.5)**: Diese Klassen haben eine ausgewogene Anzahl an ein- und ausgehenden Abhängigkeiten.

## **3. Vorschlag für eine Paketstruktur**

Eine gängige Methode zur Strukturierung ist die Aufteilung nach Funktionalität oder Schichten. Für dieses System schlage ich eine Kombination vor, die Entitäten von Services trennt:

1. **com.example.model** (oder com.example.entities)
   * User.java
   * Subscription.java
   * Website.java
   * Notification.java
   * Dieses Paket enthält die Kernobjekte des Systems, die hauptsächlich Daten und zugehöriges Verhalten kapseln.
2. **com.example.services**
   * Monitor.java
   * NotificationService.java
   * Dieses Paket enthält Klassen, die primär Geschäftslogik oder Operationen auf den Modellobjekten ausführen.
3. **com.example.main** (optional, für die Ausführung)
   * Main.java (oder eine Klasse mit der main-Methode zum Starten der Anwendung)

**Begründung:**

* **Klare Trennung der Verantwortlichkeiten**: Datenhaltende Entitäten sind von den prozessorientierten Diensten getrennt.
* **Wartbarkeit**: Änderungen an der Geschäftslogik (Services) beeinflussen nicht direkt die Struktur der Datenobjekte (Model) und umgekehrt, solange die Schnittstellen stabil bleiben.
* **Testbarkeit**: Services und Entitäten können oft unabhängiger voneinander getestet werden.

## **4. Optionen zur Reduzierung der Kopplung zwischen Paketen**

Mit der oben vorgeschlagenen Paketstruktur (model und services) haben wir folgende Hauptkopplungen:

* services hängt von model ab (z.B. Monitor verwendet Website, NotificationService verwendet Notification). Dies ist eine natürliche und oft erwünschte Abhängigkeitsrichtung.
* Innerhalb des model-Pakets gibt es Kopplungen (z.B. User -> Subscription -> Website).
* Die Klasse Website im model Paket hängt vom NotificationService im services Paket ab. Dies ist eine Abhängigkeit vom "Model" zum "Service", die man manchmal vermeiden möchte (um das Model unabhängiger zu machen).

Optionen zur Reduzierung der Kopplung, insbesondere der Abhängigkeit von Website zu NotificationService:

1. **Dependency Inversion Principle (DIP) durch Interfaces**:
   * Definiere ein Interface INotificationService im model-Paket (oder einem dedizierten interfaces-Paket).  
     Java  
     // In com.example.model (oder com.example.interfaces)  
     public interface INotificationService {  
      void sendNotification(Notification notification);  
     }
   * Die Klasse Website würde dann von diesem Interface abhängen, nicht von der konkreten Implementierung.  
     Java  
     // In com.example.model.Website  
     private INotificationService notificationService;  
     public Website(String url, INotificationService notificationService) { /\* ... \*/ }
   * Die Klasse NotificationService im services-Paket würde dieses Interface implementieren.  
     Java  
     // In com.example.services.NotificationService  
     public class NotificationService implements INotificationService { /\* ... \*/ }
   * Dies kehrt die Abhängigkeitsrichtung um: Der services-Teil hängt nun von einer Abstraktion ab, die vom model-Teil definiert wird (oder beide hängen von einer gemeinsamen Abstraktion ab). Das model ist nicht mehr direkt vom konkreten services Paket abhängig. Die konkrete Service-Implementierung wird zur Laufzeit injiziert (Dependency Injection).
2. **Event-Driven Architecture / Observer Pattern**:
   * Website könnte ein Event auslösen (z.B. UpdateFoundEvent), wenn ein Update gefunden wird.
   * Ein NotificationEventHandler (der Teil des services-Pakets sein könnte oder in einem eigenen events-Paket liegt) würde auf dieses Event lauschen und dann den NotificationService aufrufen.
   * Dies entkoppelt Website vollständig vom NotificationService. Website weiß nicht einmal, dass Benachrichtigungen gesendet werden.
   * Benötigt einen Event-Bus oder ein Observer-Mechanismus.
3. **Callbacks / Function Pointers (in Java über funktionale Interfaces)**:
   * Die checkForUpdates-Methode könnte eine Funktion (z.B. einen Consumer<Notification>) als Parameter akzeptieren, die aufgerufen wird, wenn eine Benachrichtigung erstellt wird.
   * Der Monitor oder eine andere übergeordnete Logik würde dann die Verbindung zum NotificationService herstellen und die entsprechende Funktion an Website übergeben.
   * Dies macht Website flexibler, aber verlagert die Verantwortung für die Verknüpfung.
4. **Facade Pattern für Services**:
   * Wenn das services-Paket viele detaillierte Operationen anbietet, könnte eine Fassade im services-Paket eine vereinfachte Schnittstelle für das model-Paket bereitstellen. Dies reduziert nicht unbedingt die Abhängigkeit selbst, kann aber die Komplexität der Interaktion verringern.

Für die gegebene Struktur ist die Anwendung des **Dependency Inversion Principle** (Option 1) oft der sauberste und direkteste Weg, um die Kopplung zwischen Website (Model) und NotificationService (Service) zu reduzieren und das Model-Paket unabhängiger zu machen