SWED-Übung -4 Gürbüz Emre Tatar(1455968)

. User

- **Ce (Efferent):** Depends on Subscription
- **Ca (Afferent):** Used by Subscription, possibly MonitorService

Metric	Value
Ca	1
Ce	1
I	1 / (1 + 1) = **0.5**

2. Subscription

- **Ce:** Depends on User, Website, NotificationChannel
- **Ca:** Used by Notification, MonitorService

Metric	Value
Ca	2
Ce	3
I	3 / (2 + 3) = **0.6**

3. Website

- **Ce:** None
- **Ca:** Used by Subscription, MonitorService

Metric	Value
Ca	2
Ce	0
I	0 / (2 + 0) = **0.0** (very stable)

4. Notification

- **Ce:** Depends on Subscription, NotificationChannel (indirectly via Subscription)
- **Ca:** None (no class uses Notification directly — optional depending on MonitorService)

Metric	Value
Ca	0
Ce	1
I	1 / (0 + 1) = **1.0** (unstable)

5. NotificationChannel + subclasses (Email, SMS, Push)

- **Ce (for base class):** None
- **Ca:** Used by Subscription, Notification (via Subscription)

Metric	Value
Ca	2
Ce	0
I	0.0

6. MonitorService

- **Ce:** Depends on Subscription, Website, possibly Notification
- **Ca:** None

Metric	Value
Ca	0
Ce	3
I	1.0

Class	Ca	Ce	Instability (I)
User	1	1	0.5
Subscription	2	3	0.6
Website	2	0	0.0
Notification	0	1	1.0
NotificationChannel	2	0	0.0
MonitorService	0	3	1.0

 Low instability (Website, NotificationChannel) indicates these are stable, foundational classes.

 High instability (Notification, MonitorService) means they are likely to change and rely heavily on others.

 Subscription has moderate instability due to its central role in the system.

Suggested Package Structure

com.yourname.websitemonitoring

|

├── model → Domain classes (User, Subscription, Website, etc.)

├── service → Core business logic (MonitorService, etc.)

├── notification → Notification delivery classes (Notification, NotificationChannel, etc.)

├── util → Utility/helper classes (if needed)

└── main → Entry point (Main class for testing or running the app)

Package	Purpose
model	Contains data classes (POJOs) like User, Subscription, Website
service	Contains logic classes like MonitorService, which run the main logic
notification	Handles how notifications are delivered (e.g., EmailChannel, SMSChannel)
util *(optional)*	Any helper or reusable utility classes (e.g., hashing content, etc.)
main	Contains the main class to run and test the application

Class Name	Suggested Package Path
User.java	com.e.tatar.websitemonitoring.model
Subscription.java	com.e.tatar.websitemonitoring.model
MonitorService.java	com.e.tatar.websitemonitoring.service
Notification.java	com.e.tatar.websitemonitoring.notification
EmailChannel.java	com.e.tatar.websitemonitoring.notification
Main.java	com.e.tatar.websitemonitoring.main

What is coupling?
Coupling refers to the degree of dependency between software modules or packages. High coupling means packages/classes depend heavily on each other, which makes maintenance and scalability difficult. Low coupling is desirable for flexibility and easier updates.

Options to reduce coupling between packages:

1. **Use Interfaces and Abstraction:**
   Define interfaces or abstract classes in one package that other packages can implement or extend. This way, dependencies rely on abstractions, not concrete implementations.
2. **Apply Dependency Injection:**
   Instead of creating dependencies directly inside classes, inject them from outside. This reduces direct package dependencies.
3. **Limit Package Visibility:**
   Use Java's access modifiers (public, protected, default, private) wisely to restrict classes or methods to package-internal use where possible.

4. **Use Facade Pattern:**
   Provide a simplified interface in one package that hides the complex interactions of underlying packages, reducing direct dependencies.
5. **Modular Design and Clear Responsibility:**
   Design packages with clear, focused responsibilities to avoid unnecessary dependencies.
6. **Event-Driven or Observer Patterns:**
   Use events or callbacks to notify other packages rather than calling their methods directly.

For my project, it makes the most sense to use **Interfaces and Abstraction along with Dependency Injection** to reduce coupling. This way, packages depend on interfaces rather than concrete classes, making your code easier to modify and extend.

Since you have a modular package structure in IntelliJ, **limiting package visibility** is also helpful to prevent unnecessary access and reduce coupling.

Using the Facade pattern is a good idea too, as it hides complexity and simplifies communication between packages.