TRƯỜNG ĐẠI HỌC VĂN LANG

**KHOA CÔNG NGHỆ THÔNG TIN**

A red logo with white text

AI-generated content may be incorrect.

**NHẬP MÔN CÔNG NGHỆ PHẦN MỀM**

NGÀNH: CÔNG NGHỆ THÔNG TIN

**Chuẩn hóa code theo nguyên tắc KISS/SOLID**

**SVTH: Nguyễn Văn Long – 2374802010282**

**GVHD: Nguyễn Minh Tân**

**TP. Hồ Chí Minh – Năm 2025**

**Mục lục**

[**Phần 1: Giải thích Refactor Code** 2](#_Toc204150276)

[**1.** **Mô tả mục tiêu đoạn code** 2](#_Toc204150277)

[**2.** **Các vấn đề của đoạn code ( ban đầu )** 2](#_Toc204150278)

[**3.** **Giải thích các nguyên tắc đã áp dụng** 4](#_Toc204150279)

[**4.** **Các bước refactor cụ thể và lý do cho các thay đổi** 5](#_Toc204150280)

[**5.** **So sánh code trước và sau khi refactor** 7](#_Toc204150281)

[**Phần 2: Hướng dẫn các bước đưa code lên GitHub và quản lý nhánh** 11](#_Toc204150282)

[**Trích xuất tài liệu tham khảo theo chuẩn APA** 14](#_Toc204150283)

# **Phần 1: Giải thích Refactor Code**

1. **Mô tả mục tiêu đoạn code**

Đoạn code được cung cấp xây dựng một hệ thống quản lý nhiệm vụ cá nhân đơn giản. Mục tiêu chính của nó là cho phép người dùng thêm một nhiệm vụ mới vào danh sách công việc.

Cụ thể, chức năng **addNewTaskWithViolations** nhận các thông tin đầu vào bao gồm:

* Tiêu đề (title)
* Mô tả (description)
* Ngày hết hạn (due date)
* Mức độ ưu tiên (priority)

Sau khi nhận thông tin, chương trình sẽ thực hiện các bước kiểm tra dữ liệu cơ bản (validation) như kiểm tra tiêu đề và ngày tháng có hợp lệ không. Nếu dữ liệu hợp lệ và nhiệm vụ chưa tồn tại, nó sẽ tạo một bản ghi nhiệm vụ mới và lưu trữ vào một tệp tin cơ sở dữ liệu có định dạng JSON (**tasks\_database.json**). Việc lưu trữ này giúp dữ liệu được duy trì ngay cả khi chương trình kết thúc.

1. **Các vấn đề của đoạn code ( ban đầu )**

Đoạn code ban đầu, mặc dù hoạt động, nhưng vi phạm nhiều nguyên tắc thiết kế phần mềm quan trọng, khiến nó khó bảo trì, khó mở rộng và khó kiểm thử.

* **Vi phạm Nguyên tắc KISS (Keep It Simple, Stupid):**
  + Phương thức **addNewTaskWithViolations** quá dài và phức tạp. Nó chứa nhiều mức lồng nhau (if-else, try-catch, for loop), khiến luồng hoạt động khó theo dõi.
  + Việc sử dụng **JSONObject** để thao tác dữ liệu rất dễ gây lỗi (gõ sai key, ép kiểu sai) và làm code khó đọc hơn so với việc sử dụng một đối tượng Java có kiểu dữ liệu rõ ràng.
* **Vi phạm Nguyên tắc YAGNI (You Ain't Gonna Need It):**
  + Như trong chú thích của code, thuộc tính **is\_recurring** và **recurrence\_pattern** được thêm vào mà không có bất kỳ logic nào xử lý chúng. Điều này làm tăng độ phức tạp không cần thiết cho đối tượng nhiệm vụ.
  + Hậu quả: Làm tăng tốc độ phức tạp không cần thiết cho cấu trúc dữ liệu và logic mà không mang lại giá trị tức thời.
* **Vi phạm các nguyên tắc SOLID**
* **S - Single Responsibility Principle (Nguyên tắc Trách nhiệm Đơn): VI PHẠM NẶNG**
  + **Vấn đề:** Lớp **PersonalTaskManagerViolations** và đặc biệt là phương thức **addNewTaskWithViolations** đang đảm nhiệm quá nhiều trách nhiệm khác nhau:
    1. **Xử lý Logic nghiệp vụ (Business Logic):** Kiểm tra tính hợp lệ của dữ liệu đầu vào (validation), kiểm tra một nhiệm vụ có bị trùng lặp hay không.
    2. **Truy cập Dữ liệu (Data Persistence):** Chứa logic để đọc (**loadTasksFromDb**) và ghi (**saveTasksToDb**) dữ liệu trực tiếp từ một tệp JSON.
    3. **Mô hình hóa Dữ liệu (Data Modeling):** Sử dụng **JSONObject**, một cấu trúc dữ liệu chung chung, để đại diện cho một "nhiệm vụ". Cấu trúc của một nhiệm vụ (các trường **id**, **title**,...) bị định nghĩa rời rạc bên trong phương thức.
    4. **Tương tác Người dùng (Presentation):** In các thông báo lỗi (**Lỗi: Tiêu đề không được để trống.**) và thông báo thành công trực tiếp ra giao diện dòng lệnh (**System.out**, **System.err**).
  + **Hậu quả:** Bất kỳ thay đổi nào, dù là thay đổi quy tắc nghiệp vụ, thay đổi cách lưu trữ, hay thay đổi cách hiển thị thông báo, đều đòi hỏi phải sửa đổi lớp này. Điều này làm cho lớp rất dễ bị lỗi khi thay đổi.
* **O - Open/Closed Principle (Nguyên tắc Mở/Đóng): VI PHẠM**
  + **Vấn đề:** Lớp này không "mở" cho việc mở rộng nhưng lại "mở" cho việc sửa đổi.
    1. **Ví dụ 1:** Nếu chúng ta muốn thay đổi nguồn lưu trữ từ file JSON sang một cơ sở dữ liệu như SQLite hoặc một API web, chúng ta phải vào bên trong lớp **PersonalTaskManagerViolations** và **sửa đổi trực tiếp** các phương thức **loadTasksFromDb**, **saveTasksToDb**.
    2. **Ví dụ 2:** Nếu muốn thêm một mức độ ưu tiên mới ("Khẩn cấp"), ta phải sửa đổi mảng **validPriorities** cứng bên trong phương thức.
  + **Hậu quả:** Việc mở rộng chức năng đòi hỏi phải thay đổi code đã tồn tại và đã được kiểm thử, làm tăng nguy cơ phát sinh lỗi mới.
* **D - Dependency Inversion Principle (Nguyên tắc Đảo ngược Phụ thuộc): VI PHẠM**
  + **Vấn đề:** Các module cấp cao (chứa logic nghiệp vụ) đang phụ thuộc trực tiếp vào các module cấp thấp (chi tiết triển khai).
    1. Ở đây, **PersonalTaskManagerViolations** (module cấp cao) phụ thuộc trực tiếp vào các lớp triển khai cụ thể như **org.json.simple.JSONArray**, **FileReader**, **FileWriter**.
  + **Lẽ ra phải:** Module cấp cao nên phụ thuộc vào các *abstraction* (giao diện - interface), và các module cấp thấp sẽ triển khai các interface đó.
  + **Hậu quả:** Code bị ràng buộc chặt chẽ (tight coupling) với thư viện và công nghệ lưu trữ cụ thể (JSON). Việc thay thế các thành phần cấp thấp trở nên rất khó khăn.

1. **Giải thích các nguyên tắc đã áp dụng**

Để giải quyết các vấn đề trên, nhóm đã áp dụng các nguyên tắc sau:

* **KISS (Keep It Simple, Stupid):**
  + **Mục tiêu:** Giữ cho mỗi thành phần (lớp, phương thức) đơn giản và tập trung vào một việc duy nhất.
  + **Áp dụng:** Tách phương thức lớn thành nhiều phương thức nhỏ, chuyên biệt. Sử dụng lớp **Task** (POJO) thay cho **JSONObject** để làm cho cấu trúc dữ liệu rõ ràng và an toàn hơn.
* **DRY (Don't Repeat Yourself):**
  + **Mục tiêu:** Tránh lặp lại code. Mỗi mẩu kiến thức hoặc logic chỉ nên tồn tại ở một nơi duy nhất.
  + **Áp dụng:** Tách logic truy cập dữ liệu (đọc/ghi file) vào một lớp riêng biệt (**TaskRepository**) để các thành phần khác có thể tái sử dụng mà không cần biết chi tiết về cách lưu trữ.
* **YAGNI (You Ain't Gonna Need It):**
  + **Mục tiêu:** Chỉ triển khai những gì thực sự cần thiết tại thời điểm hiện tại.
  + **Áp dụng:** Loại bỏ các thuộc tính và logic liên quan đến nhiệm vụ lặp lại (**is\_recurring**, **recurrence\_pattern**) vì chúng chưa được sử dụng. Chúng ta có thể thêm lại sau khi có yêu cầu cụ thể.
* **SOLID (cụ thể là SRP - Single Responsibility Principle):**
  + **Mục tiêu:** Mỗi lớp hoặc module chỉ nên có một lý do duy nhất để thay đổi, tức là chỉ chịu trách nhiệm cho một chức năng duy nhất.
  + **Áp dụng:** Phân tách code thành 3 lớp chính với các trách nhiệm rõ ràng:
    1. **Task**: Đại diện cho dữ liệu (Model).
    2. **TaskRepository**: Chịu trách nhiệm truy cập dữ liệu (Data Access Layer).
    3. **TaskService**: Chịu trách nhiệm về logic nghiệp vụ (Business Logic Layer).

1. **Các bước refactor cụ thể và lý do cho các thay đổi**

**Bước 1: Tách Lớp Mô hình Dữ liệu (Task.java)**

* **Mục tiêu:** Tạo một lớp POJO (Plain Old Java Object) chỉ để đại diện cho một đối tượng "Nhiệm vụ".
* **Hành động:**
  1. Tạo file **Task.java**.
  2. Định nghĩa các trường dữ liệu cần thiết (**id**, **title**, **description**, **dueDate**, **priority**, **status**, **createdAt**, **lastUpdatedAt**) với các kiểu dữ liệu phù hợp (**String**, **LocalDate**, **LocalDateTime**).
  3. **Giải quyết "Magic Strings":** Thay thế các trường **String priority** và **String status** bằng kiểu **Enum**. Điều này đảm bảo an toàn kiểu dữ liệu, tránh lỗi chính tả và giúp code tự diễn giải (self-documenting).
  4. **Cải thiện logic tạo đối tượng:** Tạo ra hai constructor riêng biệt: một để **tạo mới** một Task (tự động gán ID, status mặc định) và một để **tái tạo** một Task từ dữ liệu đã có (nhận đầy đủ các tham số).
  5. **Giải quyết YAGNI:** Loại bỏ các trường không sử dụng là **isRecurring** và **recurrence\_pattern**.
* **Lý do & Lợi ích:**
  1. **Tách biệt (Decoupling):** Lớp **Task** giờ đây hoàn toàn độc lập, không biết gì về JSON hay cách nó được lưu trữ.
  2. **An toàn & Rõ ràng:** Sử dụng **Enum** giúp code an toàn hơn và dễ đọc hơn nhiều.
  3. **Tuân thủ SRP:** Lớp này chỉ có một lý do để thay đổi: khi định nghĩa của một Task thay đổi.

**Bước 2: Tách Lớp Truy cập Dữ liệu (TaskRepository.java)**

* **Mục tiêu:** Tập trung toàn bộ logic liên quan đến việc đọc/ghi dữ liệu vào một nơi duy nhất.
* **Hành động:**
  1. Tạo file **TaskRepository.java**.
  2. Di chuyển các phương thức **loadTasksFromDb** và **saveTasksToDb** từ lớp cũ vào đây và đặt chúng ở chế độ **private**.
  3. Tạo các phương thức **public** để cung cấp một API rõ ràng cho các lớp khác sử dụng, ví dụ: **findAll()**, **save()**, **deleteById()**, **findById()**.
  4. Tạo các phương thức chuyển đổi **private** là **toJsonObject(Task task)** và **fromJsonObject(JSONObject jsonObject)**. Đây là cầu nối duy nhất giữa đối tượng **Task** thuần túy và đối tượng **JSONObject** của thư viện.
* **Lý do & Lợi ích:**
  1. **Tuân thủ SRP:** Lớp này chỉ chịu trách nhiệm về việc lưu trữ và truy xuất dữ liệu.
  2. **Che giấu chi tiết (Abstraction):** Các lớp khác không cần biết dữ liệu được lưu trong file JSON, file XML, hay database. Chúng chỉ cần gọi **repository.findAll()**.
  3. **Dễ thay đổi:** Nếu muốn chuyển sang dùng database SQL, chúng ta chỉ cần tạo một lớp **TaskRepositoryDBImpl** mới mà không cần sửa các lớp nghiệp vụ.
  4. **Chống lặp code (DRY):** Logic đọc/ghi file được viết một lần và tái sử dụng.

1. **So sánh code trước và sau khi refactor**

| **Tiêu chí** | **Trước Khi Refactor (PersonalTaskManagerViolations.java)** | **Sau Khi Refactor (Task, TaskRepository, TaskService, TaskValidator, TaskDuplicateChecker)** | **Lợi ích** |
| --- | --- | --- | --- |
| **Cấu trúc & Phân lớp** | Toàn bộ logic (validate, đọc ghi file, xử lý nghiệp vụ) nằm chung trong 1 lớp và 1 hàm lớn addNewTaskWithViolations. Không có sự phân chia rõ ràng. | Áp dụng mô hình phân lớp rõ ràng: - Task: Dữ liệu (Model) - TaskRepository: Lưu trữ, thao tác file - TaskService: Xử lý nghiệp vụ - TaskValidator: Kiểm tra hợp lệ - TaskDuplicateChecker: Kiểm tra trùng lặp | Giúp code rõ ràng, dễ tìm, dễ sửa. Mỗi phần đảm nhận một nhiệm vụ riêng biệt. Dễ phối hợp khi làm nhóm. |
| **Nguyên tắc KISS**(Keep It Simple, Stupid) | Phương thức quá dài, lồng ghép nhiều chức năng: validate, đọc file, kiểm tra trùng lặp, ghi file… Xử lý bằng JSONObject trực tiếp, gây rối. | Mỗi hàm thực hiện 1 chức năng cụ thể. Task là POJO giúp thao tác với dữ liệu đơn giản, an toàn. | Code dễ hiểu, dễ đọc, dễ debug. Người mới vào dự án dễ tiếp cận nhanh. |
| **Nguyên tắc DRY**(Don’t Repeat Yourself) | Lặp lại nhiều đoạn code đọc ghi file ở nhiều nơi khác nhau. Nếu sau này thêm chức năng update, delete, sẽ tiếp tục bị lặp lại. | Đọc/ghi file gom vào TaskRepository. Các chức năng khác chỉ gọi lại, không cần viết lại code. | Giảm trùng lặp, tiết kiệm thời gian khi bảo trì. Nếu cần sửa thao tác file, chỉ cần sửa 1 chỗ duy nhất. |
| **Nguyên tắc YAGNI**(You Ain’t Gonna Need It) | Thêm các thuộc tính chưa sử dụng như is\_recurring, recurrence\_pattern, làm phức tạp hóa mô hình dữ liệu. | Loại bỏ các thuộc tính chưa cần thiết, chỉ giữ những gì phục vụ chức năng hiện tại. | Giữ code gọn gàng, tập trung đúng yêu cầu. Tránh ôm đồm những thứ "có thể cần" gây rối. |
| **Xử lý lỗi** | In lỗi trực tiếp ra console bằng System.out, trả về null. Client không biết chính xác lỗi gì xảy ra. | Sử dụng Exception (IllegalArgumentException) với thông báo rõ ràng. Client có thể bắt và xử lý lỗi tùy ý. | Giúp chương trình ổn định hơn, dễ kiểm soát luồng lỗi. Dễ tích hợp vào các hệ thống lớn hơn. |
| **Kiểm thử (Testability)** | Rất khó kiểm thử, phải thao tác với file thật. Khó test riêng biệt các trường hợp lỗi hoặc nghiệp vụ. | Có thể viết Unit Test cho TaskService độc lập với file nhờ mock TaskRepository. | Viết test dễ dàng hơn, phát hiện lỗi sớm, tự tin khi mở rộng hoặc refactor tiếp. |
| **Mở rộng (Scalability)** | Khi muốn thêm chức năng mới (ví dụ: updateTask, searchTask), phải sửa code cũ, dễ gây lỗi dây chuyền, khó tái sử dụng. | Chỉ cần thêm method vào TaskService hoặc TaskRepository. Không ảnh hưởng đến phần khác. | Hệ thống dễ mở rộng, có thể phát triển thêm tính năng mà không làm rối code cũ. |
| **Áp dụng SOLID** | - **S (Single Responsibility)**: Vi phạm, 1 lớp làm quá nhiều việc.- **O (Open/Closed)**: Khó mở rộng, thêm tính năng phải sửa code cũ.- **D (Dependency Inversion)**: Phụ thuộc chi tiết vào file, JSONObject, không tách tầng nghiệp vụ và lưu trữ. | - **S**: Mỗi class 1 nhiệm vụ rõ ràng.- **O**: Thêm chức năng mới không cần sửa code cũ.- **D**: TaskService gọi qua TaskRepository, dễ mock, dễ test. | Code sạch, dễ mở rộng, giảm rủi ro lỗi khi sửa đổi hoặc thêm tính năng. Hỗ trợ tốt phát triển lâu dài. |

# **Phần 2: Hướng dẫn các bước đưa code lên GitHub và quản lý nhánh**

**1. Tạo repository trên GitHub:**

- Truy cập GitHub, đăng nhập tài khoản nhóm.

- Nhấn nút “New” để tạo repository mới.

- Nhập tên repo (ví dụ: HK243\_NM-CNPM\_DAMH\_Nhom2).

- Chọn “Public” hoặc “Private” theo yêu cầu nhóm.

- Tick chọn Add a README file và .gitignore (chọn Java).

- Nhấn Create repository.

**2. Khởi tạo Git trong máy và kết nối với GitHub:**

- Mở terminal tại thư mục chứa mã nguồn.

- Khởi tạo Git: git init

- Thêm remote repo từ GitHub: git remote add origin https://github.com/yuuki2206/HK243\_NM-CNPM\_DAMH\_Nhom2.git

**3. Tạo nhánh feature/refactor-code:**

- Tạo nhánh mới: git checkout -b feature/refactor-code

- Mục đích: Tách biệt quá trình refactor khỏi nhánh chính main, đảm bảo tính ổn định cho dự án.

**4. Thêm và commit code đã refactor:**

- Cấu trúc thư mục:

PersonalTaskManager/

└── src/

└── person/

├── Main.java

├── PersonalTaskManagerViolations.java

├── Task.java

├── TaskRepository.java

├── TaskService.java

├── TaskValidator.java

- Thêm tất cả file vào Git: git add .

- Commit lần 1: git commit -m "Khởi tạo cấu trúc dự án ban đầu"

- Commit lần 2: git commit -m "Refactor code: áp dụng KISS/DRY/YAGNI/SOLID "

**5. Đẩy code lên GitHub:**

- Đẩy nhánh feature/refactor-code lên GitHub: git push origin feature/refactor-code

**6. Tạo Pull Request (PR):**

- Truy cập GitHub → repository dự án.

- Vào tab Pull requests, nhấn New pull request.

- So sánh feature/refactor-code với main.

- Viết mô tả: “...”

- Nhấn Create pull request.

**7. Hợp nhất Pull Request:**

- Nhấn Merge pull request → Confirm merge.

- (Tùy chọn) Xóa nhánh feature/refactor-code sau khi merge.

- Cập nhật nhánh main trên máy local: git checkout main && git pull

**8. Thêm tài liệu Word vào GitHub:**

- Tạo file Word từ nội dung báo cáo này.

- Đặt tên: HK243\_NM-CNPM\_DAMH\_Nhom2\_CleanCode\_Report.docx

- Thêm file vào Git:

git add HK243\_NM-CNPM\_DAMH\_Nhom2\_CleanCode\_Report.docx

git commit -m "Thêm tài liệu báo cáo refactor"

git push origin main

**Các nhánh đã tạo:**

- main: Nhánh chính, chứa code ổn định

- feature/refactor-code: Refactor code, cải tiến theo nguyên tắc clean code

**Lưu ý quan trọng:**

- Tất cả thành viên nên có quyền truy cập vào repository.

- Ghi chú commit rõ ràng, cụ thể: ví dụ "Tách hàm kiểm tra trùng lặp thành TaskDuplicateChecker.java".

- Thường xuyên đồng bộ nhánh main: git pull origin main

- Đảm bảo file tasks\_database.json tồn tại để các class đọc/ghi dữ liệu hoạt động đúng.

# 

# **Trích xuất tài liệu tham khảo theo chuẩn APA**

1. Martin, R. C. (2003). *Agile software development, principles, patterns, and practices*. Prentice Hall.
2. Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.
3. Martin, R. C. (2018). *Clean architecture: A craftsman's guide to software structure and design*. Prentice Hall.
4. Fowler, M. (2018). *Refactoring: Improving the design of existing code* (2nd ed.). Addison-Wesley Professional.
5. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional.
6. Hunt, A., & Thomas, D. (1999). *The pragmatic programmer: From journeyman to master*. Addison-Wesley Professional.
7. Pressman, R. S., & Maxim, B. R. (2019). *Software engineering: A practitioner's approach* (9th ed.). McGraw-Hill Education.