شبکه‌ی زیبا

فرض کنید برنامه‌ای برای مدیریت انتقال داده‌ها در شبکه داریم. بخشی از این برنامه مختص اعمال تغییرات روی داده‌ها است. این تغییرات در حال حاضر شامل موارد زیر هستند:

- فشرده‌سازی داده
- رمزنگاری داده
- تبدیل داده به چند تکه (*chunk*)

قواعد مختلفی برای برای شبکه‌های مختلف تعریف شده که برخی از آن‌ها به شرح زیر هستند:

- در شبکه‌ی *A* ، داده‌ها فقط باید فشرده‌سازی شوند.
- در شبکه‌ی *B* ، داده‌ها هم باید فشرده‌سازی شوند و هم باید رمزنگاری شوند.
- در شبکه‌ی *C* ، داده‌ها هم باید رمزنگاری شوند و هم باید به چند تکه تقسیم شوند.

از آن‌جایی که به‌زودی قرار است از این برنامه در شبکه‌های دیگری نیز استفاده شود، ممکن است همهٔ حالت‌های ممکن به ازای اعمال یا عدم اعمال تغییرات مختلف روی داده‌ها موردنیاز باشد. به همین خاطر، کلاس‌های زیر برای اعمال انواع تغییرات روی داده‌ها تعریف شده‌اند:

- `FileCompressor`
- `FileEncryptor`
- `FileChunker`
- `FileCompressorAndEncryptor`
- `FileCompressorAndChunker`
- `FileEncryptorAndChunker`
- `FileCompressorAndChunker`
- `FileCompressorAndEncryptorAndChunker`

منطق فشرده‌سازی، رمزنگاری و تکه‌تکه‌سازی داده‌ها در سه کلاس مجزا تعریف شده است، اما برای هر یک از حالت‌های مختلف اعمال تغییرات نیز یک کلاس مجزا در نظر گرفته شده است. در این روش طراحی، اگر n تغییر مختلف روی داده‌ها قابل اعمال باشد، در نهایت $2^n - 1$ کلاس برای حالت‌های مختلف خواهیم داشت.

با استفاده از الگوهای طراحی‌ای که با آن‌ها آشنا هستید، این روش طراحی را به‌گونه‌ای اصلاح کنید که به ازای هر یک از حالت‌های مختلف، نیازمند تعریف کلاس جداگانه نباشیم. در صورت نیاز، می‌توانید کلاس‌های جدیدی در نظر بگیرید.

برای این سؤال نیازمند نوشتن کد نخواهید بود. ذکر نام و وظیفهٔ کلاس‌هایی که در نظر گرفته‌اید کافی است.

آن‌چه باید آپلود کنید

پاسخ خود را در قالب یک فایل *PDF* آپلود کنید.

جست‌وجوگر خسته

آیدا که از سرچ کردن برای پروژه‌هایش و باز کردن تب‌های متعدد خسته شده، تصمیم گرفته به سبک دیگری جست‌وجو انجام بدهد.

او می‌خواهد سامانه‌ای راه‌اندازی کند تا کاربران بتوانند چیزهایی که می‌خواهند در مورد آن بدانند را به ترتیب در یک فایل بنویسند و موتور جست‌وجو آن فایل را خوانده، محتوای تمام فایل‌های مربوط به آن‌ها را جمع‌آوری کرده و در یک فایل `result` بنویسد.

توضیحات فایل پروژه

فایل اولیه پروژه را از این لینک دریافت کنید. ساختار فایل‌ها به‌صورت زیر است:

```
file-search-engine
├── FileSearchEngine.java
├── Test.java
└── files
    ├── java1.txt
    ├── java3.txt
    ├── subjects.txt
    └── weather2.txt
```

تمامی فایل‌های متنی برنامه در یک فولدر قرار می‌گیرند که مسیر آن به کانستراکتور کلاس `FileSearchEngine` داده می‌شود (چه فایل‌های خواندنی و چه فایلی که در نهایت تولید می‌شود).

در ادامه به توضیح متدها خواهیم پرداخت:

```java
public String[] readSubjects()
```

این متد از فولدر موردنظر، فایل `subjects.txt` را می‌خواند. در فایل `subjects.txt` موضوعات قابل سرچ هر کدام به صورت جداگانه در یک خط نوشته شده‌اند. متد در نهایت در صورتی که اکسپشنی در طول اجرایش رخ ندهد، یک آرایه‌ی رشته‌ای متشکل از تمام موضوعات نوشته شده در `subjects.txt` برمی‌گرداند. در صورت مواجهه با اکسپشن، باید یک آرایه استرینگی خالی برگردانده شود. تضمین

می‌شود که تعداد موضوعات قابل سرچ کمتر از ۱۰۰ تا است. توجه داشته باشید که شما نمی‌توانید امضای این متد را با اضافه کردن `throws Exception` تغییر دهید.

```
1 | public boolean arrangeSearches(String[] subjects)
```

وظیفه‌ی این متد، جمع‌آوری نتایج و ریختن آن‌ها در یک فایل واحد است. ورودی آن آرایه `subjects` است.

در متد `getFileNames(String[] subjects)` ، سابجکت‌ها به صورت ورودی پاس داده می‌شوند. این متد تمام فایل‌هایی که مربوط به یک سابجکت خاص هستند را به یک آرایه اضافه می‌کند. در نهایت به شما آرایه‌ای متشکل از نام تمام فایل‌هایی که برای خواندن به آن‌ها احتیاج دارید می‌دهد. این متد قبلاً پیاده‌سازی شده و نیاز به تغییر در آن نیست.

در متد `arrangeSearches` شما با دریافت `subjects` باید تمام فایل‌های مربوط به این موضوعات را جمع‌آوری کرده و محتویات آن‌ها را در فایل `result.txt` کپی کنید. بین محتویات هر دو فایل که در `result.txt` نوشته می‌شود، باید یک کاراکتر `\n` قرار دهید. همچنین، رشته‌ای که در فایل `result.txt` نوشته می‌شود را `trim` کنید. توجه کنید که `result.txt` هم باید در پوشه‌ی موردنظر قرار داشته باشد. ممکن است در آرایه‌ای که متد `getFileNames` برمی‌گرداند، نام فایلی باشد که در واقعیت وجود ندارد. در صورت برخورد با این مسئله، روند اجرای برنامه نباید متوقف شود و برنامه باید به روال عادی خود ادامه بدهد.

در نهایت، اگر تمام عملیات‌ها موفقیت‌آمیز پیش رفت، مقدار `true` و در غیر این‌صورت مقدار `false` برگردانده شود. توجه کنید که وجود نداشتن یک یا چند تا از فایل‌هایی که توسط `getFileNames` برگردانده می‌شود، نباید موجب `false` برگرداندن توسط `arrangeSearches` شود.

مثال

برای تست کد خود، فایل اولیه پروژه را دانلود کنید. در صورت اجرای `Test.java` ، باید فایل `result.txt` درون پوشه‌ی `files` با این محتوا تولید شده باشد:

```
java is cool
java is cool1
java is cool
```

```
java is cool
java is cool
java is cool
java is cool
java is cool2222222222222222222222
java 18 is now available!
java 18 is now available!
java 18 is now available!
java 18 is now available!
weather is rainy
weather is sunny
```

آن‌چه باید آپلود کنید

فایل `FileSearchEngine.java` را آپلود کنید.

دیجی‌والِت

محمدرضا و تیمش مشغول طراحی سامانه‌ای برای مدیریت کیف پول دیجیتال به‌نام *دیجی‌والِت* هستند.

در نسخه‌ی اولیه‌ی این سامانه قرار است قابلیت‌های زیر وجود داشته باشد:

- امکان ایجاد تراکنش و تغییر وضعیت آن
- امکان مشاهده‌ی لیست تراکنش‌ها
- امکان تسویه‌حساب
- امکان مشاهده‌ی موجودی کیف پول

تیم محمدرضا ساختار برنامه را طراحی کرده‌اند و از شما می‌خواهند تا پیاده‌سازی آن را انجام دهید.

جزئیات پروژه

پروژه‌ی اولیه را از این لینک دانلود کنید.

▼ ساختار فایل‌ها

```
src
└── ir
    └── digipay
        └── digiwallet
            ├── model
            │   ├── AdminWallet.java
            │   ├── Transaction.java
            │   ├── TransactionStatus.java
            │   ├── TransactionType.java
            │   └── Wallet.java
            ├── repository
            │   ├── CrudRepository.java
            │   ├── TransactionRepository.java
            │   └── TransactionRepositoryImpl.java
            ├── service
            │   ├── AdminWalletService.java
            │   └── WalletService.java
            ├── Application.java
            └── DigiWalletSampleTest.java
```

مدل‌ها

- کیف پول (Wallet)
  - این مدل تنها شامل یک شناسه از نوع String است.
- کیف پول مربوط به ادمین‌های برنامه (AdminWallet)
  - این مدل از کلاس Wallet ارث‌بری می‌کند و هیچ پیاده‌سازی اضافه‌ای نسبت به مدل Wallet ندارد.
- نوع تراکنش (TransactionType): این کلاس یک enum است که به‌ترتیب شامل دو مقدار DEPOSIT و WITHDRAWAL است.
  - تراکنش‌هایی که از نوع DEPOSIT هستند، بیانگر واریزهایی هستند که مقصد آن‌ها، کیف پول فعلی است.
  - تراکنش‌هایی که از نوع WITHDRAWAL هستند، بیانگر برداشت‌هایی هستند که از کیف پول به حساب بانکی صاحب کیف پول منتقل می‌شوند.
- وضعیت تراکنش (TransactionStatus): این کلاس یک enum است که به‌ترتیب شامل سه مقدار CANCELED ، PENDING و ACCEPTED است.
- تراکنش (Transaction)
  - این مدل به‌ترتیب شامل پراپرتی‌های زیر است:
    - long id : شناسه‌ی تراکنش
    - Wallet wallet : کیف پول مربوط به تراکنش
    - TransactionType type : نوع تراکنش
    - BigDecimal amount : مبلغ تراکنش
    - Date createdAt : زمان ایجاد تراکنش
    - TransactionStatus status : وضعیت تراکنش با مقدار اولیه‌ی PENDING
    - Date updatedAt : زمان تغییر وضعیت تراکنش
  - همه‌ی پراپرتی‌ها به‌جز پراپرتی updatedAt در کانستراکتور مقداردهی می‌شوند.
  - متد setStatus را طوری پیاده‌سازی کنید که با دریافت یک TransactionStatus ، وضعیت تراکنش را به وضعیت واردشده تغییر داده و مقدار پراپرتی updatedAt را برابر با یک آبجکت جدید از نوع Date قرار دهد.

مخزن‌ها

- اینترفیس `CrudRepository<T, ID>`
  - این اینترفیس شامل دو پارامتر جنریک `T` (نوع مدل) و `ID` (نوع شناسه‌ی مدل) است.
  - این اینترفیس شامل تعریف متدهای زیر است:
    - `boolean add(T t)` : این متد، مدل را به مخزن اضافه می‌کند؛ به شرط آن که مدل از قبل در مخزن وجود نداشته باشد. اگر مدل از قبل در مخزن موجود باشد، مقدار `false` و در غیر این‌صورت، مقدار `true` را برمی‌گرداند.
    - `List<T> getAll()` : این متد، لیست همه‌ی مدل‌های ذخیره‌شده را به‌ترتیب درج برمی‌گرداند.
    - `T get(ID id)` : این متد، مدلی که شناسه‌ی آن برابر با `id` است را برمی‌گرداند. اگر چنین مدلی یافت نشود، مقدار `null` را برمی‌گرداند.
    - `List<T> get(Predicate<T> predicate)` : این متد، لیست مدل‌هایی که شرایط داده‌شده در `predicate` را دارند به‌ترتیب درج برمی‌گرداند.
- اینترفیس `TransactionRepository`
  - این اینترفیس از اینترفیس `CrudRepository` ارث‌بری می‌کند و صرفاً نوع تراکنش‌ها (که `Transaction` است) و نوع شناسه‌ی آن‌ها (که `Long` است) را مشخص می‌کند.
- کلاس `TransactionRepositoryImpl`
  - این کلاس، اینترفیس `TransactionRepository` را پیاده‌سازی می‌کند و تراکنش‌های مربوط به همه‌ی کیف پول‌ها در آن ذخیره می‌شود.
  - متدهای این کلاس را مطابق توضیحات اینترفیس `CrudRepository` پیاده‌سازی کنید.
  - در متد `add` اگر مبلغ تراکنش کوچک‌تر یا مساوی صفر باشد، یک `IllegalArgumentException` باید پرتاب شود.

سرویس‌ها

- سرویس `WalletService`
  - از این کلاس برای مدیریت کیف پول‌ها استفاده می‌شود.
  - متد `addTransaction` را طوری پیاده‌سازی کنید که با دریافت یک تراکنش، با فراخوانی متد `add` از `transactionRepository` ، آن را به لیست تراکنش‌ها اضافه کند. این متد

در صورتی که تراکنش از قبل موجود باشد، باید مقدار false و در غیر این‌صورت، باید مقدار true را برگرداند.

- متد getTransactions(Wallet wallet) را طوری پیاده‌سازی کنید که لیست همه‌ی تراکنش‌هایی که مربوط به کیف پول ورودی هستند را به‌ترتیب درج برگرداند.
- متد getTransactions(Wallet wallet, Predicate<Transaction> predicate) را طوری پیاده‌سازی کنید که لیست همه‌ی تراکنش‌هایی که مربوط به کیف پول ورودی هستند و شرایط داده‌شده در predicate را دارند به‌ترتیب درج برگرداند.
- متد getBalance را طوری پیاده‌سازی کنید که با دریافت یک کیف پول، موجودی حساب کیف پول را در قالب یک BigDecimal برگرداند. موجودی حساب برابر با مجموع مبلغ DEPOSIT های ACCEPTED منهای مجموع مبلغ WITHDRAWAL های ACCEPTED است.
- متد setTransactionStatus را طوری پیاده‌سازی کنید که با دریافت یک تراکنش و وضعیت جدید، در صورتی که وضعیت تراکنش PENDING نبود یا وضعیت جدید (وضعیت پاس‌داده‌شده به همین متد) برابر با PENDING بود، مقدار false را برگرداند. در غیر این‌صورت، اگر تراکنش از نوع WITHDRAWAL بود و موجودی کیف پول به اندازه‌ی مبلغ تراکنش نبود، یک IllegalArgumentException پرتاب شود. در غیر این‌صورت، وضعیت تراکنش به وضعیت جدید تغییر کند، مقدار پراپرتی updatedAt تراکنش به‌روز شود و مقدار true برگردانده شود.

- سرویس AdminWalletService
  - این کلاس از کلاس WalletService ارث‌بری می‌کند.
  - متد getTransactions(Predicate<Transaction> predicate) را طوری پیاده‌سازی کنید که لیست همه‌ی تراکنش‌های مربوط به همه‌ی کیف پول‌ها که شرایط داده‌شده در predicate را دارند به‌ترتیب درج برگرداند.
  - متد getAllTransactions را طوری پیاده‌سازی کنید که لیست همه‌ی تراکنش‌های مربوط به همه‌ی کیف پول‌ها (همه‌ی تراکنش‌های موجود در مخزن) را به‌ترتیب درج برگرداند.

نکات

- لازم نیست که ذخیره‌سازی مقدار مدل‌ها *persistent* باشد (تنها یک بار اجرای برنامه را در نظر بگیرید).
- هر نمونه از مخزن باید داده‌ها را به‌صورت جداگانه در خودش ذخیره کند.
- در صورت نیاز، می‌توانید پراپرتی‌ها و متدهای جدیدی به کلاس‌ها (به‌جز کلاس `Application`) اضافه کنید.

مثال

با اجرای متد `main` موجود در کلاس `Application` ، خروجی زیر مورد انتظار است:

```
true
1000
[[Id: 1, Wallet Id: x-y-z, Type: DEPOSIT, Amount: 1000, Status: ACCEPTED]]
[[Id: 1, Wallet Id: x-y-z, Type: DEPOSIT, Amount: 1000, Status: ACCEPTED]]
```

آن‌چه باید آپلود کنید

پس از پیاده‌سازی موارد خواسته‌شده، یک فایل زیر آپلود کنید که وقتی آن را باز می‌کنیم، با ساختار زیر مواجه شویم (از سایر فایل‌ها صرف‌نظر می‌شود):

```
.
└── ir
    └── digipay
        └── digiwallet
            ├── model
            │   └── Transaction.java
            ├── repository
            │   └── TransactionRepositoryImpl.java
            ├── service
            │   ├── AdminWalletService.java
            └── └── WalletService.java
```