Tema 3 - Caching System

- Responsabili: Alexandru Ghimisi Cristian-Ionut Nancu Mihnea Andrei Petrescu Razvan Chitu
- Deadline: **17.12.2017**
- Deadline hard: 07.01.2018 - pe perioada vacanței nu se aplică depunctări pe vmchecker.
- Data publicării: 27.11.2017
- Data ultimei modificări: 27.11.2017
- Data tester-ului: 01.12.2017

Obiective

- Integrarea cu un mod arhitectural de a rezolva o problemă reală
- Aprofundarea noțiunilor de moștenire, agregare și interfațare
- Lucrul cu noțiunile din cursul de Structuri de Date (stive, cozi, liste dublu înlănțuite, hashtables) in context OOP
- Aprofundarea noțiunilor de Design Patterns:
  - **Adapter pattern**
  - **Strategy pattern**
  - **Observer pattern**
- Respectarea unui coding-style

Descriere

În cadrul acestei teme, vă propunem să implementați un sistem de caching pentru fișiere text.

Cache

Pe scurt, un **cache** este o arie temporară de stocare unde datele utilizate în mod frecvent pot fi depozitate pentru un acces rapid la acestea.

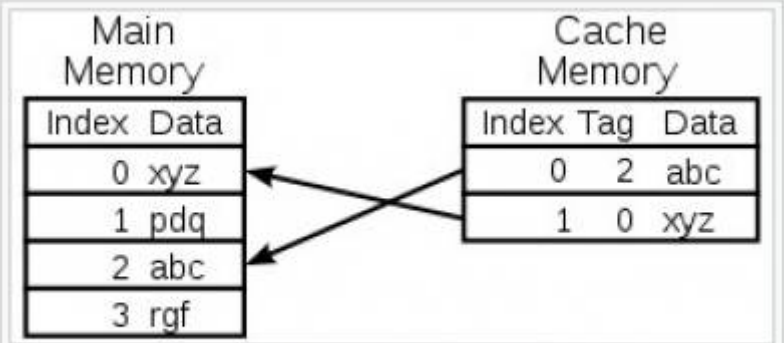

Diagram of a CPU memory cache operation

Default, sistemul de operare cache-uiește datele citite de pe disc și pe cele scrise pe disc. Acest lucru implică faptul că operațiile de citire se fac mai degrabă asupra unei memorii de sistem (system file

cache) decât asupra discului fizic. Analog, și operațiile de scriere se fac asupra memoriei de system cache. Sistemul de caching pe care îl veți implementa va simula reținerea în memorie a datelor unui fișier împreună cu calea către acel fișier pe discul fizic.

Cache replacement policy

Politicile de înlocuire a cache-ului sunt instrucțiuni de optimizare – sau algoritmi – pe care un computer sau o structură hardware le poate urma pentru a manageria memoria cache stocată pe un calculator. Când cache-ul este plin, algoritmul alege ce elemente să decarteze pentru a face loc unor elemente noi.

Variantele politicilor de replacement cache asupra cărora ne vom concentra sunt:

- **FIFO**
- **LRU**
- **TLRU**

Hit & Miss

Pentru evaluarea performanței, cache-ul vostru trebuie să comunice diverse evenimente, precum: hit, miss, put.

Cache hit reprezintă o stare în care datele solicitate de către aplicație se găsesc în memoria cache.

Cache miss reprezintă o stare în care datele solicitate de către aplicație **NU** se găsesc în memoria cache.

Cache put reprezintă o nouă stare introdusă de către noi, în care cache-ul primește date pentru a le reține (sau updata) în memoria sa.

Cerințe și detalii implementare

Pornind de la scheletul de cod, vă propunem să creați propria ierarhie cache. Aceasta trebuie să suporte funcționalitatea de a controla dimensiunea unui cache, de a elimina elementele vechi sau de a evalua performanța pe bază de metrici.

Pentru funcționalitatea de eliminare a elementelor vechi definim conceptul de **CacheStalePolicy**, care reprezintă un modul ce conține strategia pe care un cache trebuie să o urmeze pentru a elimină elementele vechi.

Odată implementate cache-urile `ObservableFIFOCache`, `LRUCache` și `TimeAwareCache`, sistemul vostru de cache va trebui să lanseze evenimente.

Este obligatorie folosirea **Observer Pattern**. Dacă tema nu respectă această cerință, nu va primi punctajul testelor.

Caches

ObservableCache

Completați funcționalitatea clasei abstracte `ObservableCache`. Clasa are nevoie de:

- o metodă pentru setarea unei politici de înlocuire a cache-ului (`setStalePolicy`)
- o metodă pentru setarea unui listener de evenimente (`setCacheListener`)
- o metodă pentru eliminarea intrărilor în cache care sunt considerate "expirate" (`clearStaleEntries`)

ObservableFIFOCache

În scheletul de cod veți găsi o clasă `FIFOCache` care reprezintă o implementare non-observabila a unui cache bazat pe coadă. Nu aveți voie să modificați codul acestei clase, ci doar să vă folosiți de metodele sale publice.

Implementați `ObservableFIFOCache` având la bază clasa `FIFOCache`, pentru ca această din urmă să poată accepta un listener și să semnaleze evenimente.

Relația dintre `ObservableFIFOCache` și `FIFOCache` ar trebui să fie una de compunere, nu de moștenire.

LRUCache

În cazul LRU cache, implementarea operațiilor specifice (get, put) trebuie să respecte o complexitate temporală **O(1)**.

Cheia rezolvării implică o listă dublu înlănțuită care permite mutarea rapidă a nodurilor. Pentru acest task vă recomandăm să implementați o lista dublu înlănțuită generică.

Nu aveţi voie să folosiţi **LinkedHashMap** în implementarea LRUCache. Orice astfel de implementare nu va fi luată în considerare.

TimeAwareCache

Implementați `TimeAwareCache` folosindu-vă de asemănarea dintre acest tip de cache și cel de la task-ul anterior (LRUCache).

`TimeAwareCache` va avea nevoie de o modalitate de a interoga timestamp-ul asociat unei chei (metoda `getTimeStampOfKey`) și de o metodă care să seteze o politică de expirare a elementelor

(`setExpirePolicy`).

Observer Pattern

Cache-urile implementate de voi au nevoie si de un mecanism de semnalare a evenimentelor de hit, miss si put (update). Design pattern-ul **Observer** poate fi aplicat aici, considerand cache-ul drept subiect si mai multe tipuri de **CacheListener** drept observatori.

StatsListener

Implementați StatsListener. Ascultă evenimentele onHit, onMiss și onPut și reține câte astfel de evenimente au fost semnalate la nivel global (trebuie să conțină getteri!).

KeyStatsListener

Implementați KeyStatsListener. Ascultă aceleași evenimente ca StatsListener, ține evidența solicitărilor per cheie și conține metodele publice care întorc liste cu primele **n** cele mai solicitate chei care s-au găsit / care nu s-au găsit în cache, sau care au fost cele mai updatate. De asemenea, va avea metode pentru a interoga aceste statistici și pentru o cheie anume.

BroadcastListener

Implementați BroadcastListener. Acesta poate conține mai mulți listeners și comunică fiecăruia evenimentele semnalate de cache.

Testarea funcționalității

Cele 3 tipuri de cache implementate de voi vor fi testate prin intermediul unui cache de fișiere. Clasa `FileCache` va folosi drept cheie calea către un fișier, iar valoarea asociată va fi chiar conținutul fișierului.

Atunci când un fișier nu este găsit în cache, el trebuie citit din sistemul de fișiere și reținut. Funcționalitatea această trebuie implementată de voi printr-un `CacheListener`, în metoda `createCacheListener` din clasă `FileCache` (hint: event-ul *onMiss*).

Precizări

Structurile de date implementate de voi trebuie să fie **generice**.

Pentru simplitate, clasa `ObservableCache` suportă doar un singur listener. Totuși, pentru testare și evaluare a performanței, am dori să folosiți mai mulți listeneri pentru același cache.

Pentru a nu schimba implementarea din `ObservableCache`, trebuie implementat un `BroadcastListener` (care să conțină mai mulți listeners și să comunice fiecăruia evenimentele semnalate de cache).

README-ul nu trebuie sa cuprinda explicatia fiecarei metode. Aceste explicatii trebuie puse in javadoc

Date de intrare. Date de ieșire.

Clasa **Main** conține logica de parsare a testelor și de afișare corespunzătoare a rezultatelor. Corectitudinea rezultatelor depinde de implementarea corecta a celor trei tipuri de Cache si a observatorilor (listeners).

Pentru partea de debug, puteți inspecta fișierele de test, care conțin și comentarii ce va pot ajuta în a înțelege rezultatele. Formatul testelor este descris în fișierul `README.md`.

Punctaj

Punctaj (100p)

- **80p** teste publice
- **10p** coding style
- **5p** README, aspect cod, comentarii (dacă e cazul)
- **5p** Javadoc

La evaluare mai pot apărea depunctări pentru nerespectarea design-ului impus sau alte situații specifice temei, neprevăzute în barem.

Pentru a primi punctele pentru coding style, tema trebuie să treacă testul executat de Checkstyle [4]. Dacă este picat și numărul de erori depistate depășește 30 (o treime din punctajul maxim, fără bonus), atunci punctele pentru coding-style nu vor fi acordate iar dacă punctajul este negativ, *acesta se trunchiază la 0*.

Exemple:

- `punctaj_total = 100` și `nr_erori = 200` ⇒ `nota_finala = 90`
- `punctaj_total = 100` și `nr_erori = 29` ⇒ `nota_finala = 100`
- `punctaj_total = 80` și `nr_erori = 30` ⇒ `nota_finala = 80`
- `punctaj_total = 80` și `nr_erori = 31` ⇒ `nota_finala = 70`

Temele vor fi testate împotriva plagiatului. Orice tentativă de copiere va duce la **anularea punctajului** de pe parcursul semestrului şi **repetarea materiei** atât pentru sursă(e) cât şi pentru destinație(ii), fără excepție.

You shall indeed not pass !!

Structura arhivei

Arhiva pe care o veţi urca pe **Vmchecker** va trebui să conţină în directorul rădăcină:

- README în care să explicaţi toate functionalitatile implementate și alte detalii relevante pentru implementare.
- directorul `src` ce conţine fişierele sursă. **Nu modificaţi denumirea directorului src. Checker-ul se bazează pe această denumire.**

Checker-ul

Checkerul de pe **VMChecker** va fi cel public din repository-ului de Github al Temei 3. Pentru testare locală puteţi rula scriptul `test.sh`.

Împreună cu checker-ul aveţi disponibil pentru testare locală şi un Makefile denumit `CachingMakefile`. Makefile-ul poate fi folosit cu următoarea comandă:

```
make -f CachingMakefile
```

Resurse

Implementarea temei va porni de la scheletul de cod disponibil în repository-ului de Github al Temei 3.

Referințe

- Indicații pentru teme
- Despre cod și IDE
- Design patterns - Singleton, Factory, Observer
- Design patterns - Command, Strategy