Részvénypiac Szimuláció - Specifikáció Skeleton Programhoz

Rátki Barnabás

2021.04.13

Tartalomjegyzék

1 Feladat **1**
- 1.1 Rövid összefoglaló a problémáról . . . 1
- 1.2 Bemenetek . . . 1
- 1.3 Beállítások . . . 2
- 1.4 Interaktív mód . . . 2
  - 1.4.1 Szimuláció megállítása/elindítása . . . 2
  - 1.4.2 Statisztikák lekérése . . . 2
  - 1.4.3 Cégek listázása . . . 2
  - 1.4.4 Cégek részletes adatai . . . 3
  - 1.4.5 Kereskedők listázása . . . 3
  - 1.4.6 Kereskedők részletes adatai . . . 3
- 1.5 Kimenetek . . . 3

2 Pontosított feladatspecifikáció **3**

3 Terv **3**
- 3.1 Objektum Terv . . . 3
  - 3.1.1 Könyvtár - Terv . . . 3
  - 3.1.2 Rc<T> - Implementáció . . . 6
- 3.2 Implementációs - Terv . . . 6
- 3.3 Skeleton Program Jelenlegi állása . . . 6

1 Feladat

1.1 Rövid összefoglaló a problémáról

A program célja egy képzeletbeli részvénypiac szimulációja.
Megjegyzés: Nem cél a valós rendszer közeli modellezése, mert ez valószínűleg lehetetlen, a feltételezések, amikre a szimuláció épül nem a valóságon alapulnak és lényeges egyszerűsítéseket tartalmaznak.
Az elképzelt piacunkban két különböző szereplő van, cégek illetve kereskedők. A részvények tulajdonos-cseréjét pedig egy központosított piac biztosítja. Az összes többi piaci körülményekben változást elérő hatást véletlenszerű események fogják generálni. A szimulációban arra vagyunk kíváncsiak, hogy az adott beállításokkal és random *seed*-el milyenek lesznek az árfolyamok grafikonjai a különböző cégeknél és mennyi tőkéjük lesz a kereskedőknek. A kereskedők kezdetben random generált beállításokkal működnek, amik egyedivé teszik az akcióikat és viselkedésüket de összeségében az a céljuk, hogy a maximalizálják profitjaikat. A cégek hasonlóan random generáltak, és a szimuláció elején fix mennyiségű részvényt bocsájtanak ki és igyekszenek maximalizálni az így beszedett pénz mennyiségét. A rendszer ciklusokban működik, minden ciklusban létrejöhetnek események, illetve random aktivált kereskedők kezelhetik befektetéseik. A piac egy úgynevezett "Aukciós Piac" elvén működik, ahol a kereskedők ajánlatainak egyezése esetén történnek eladások és vételek.

1.2 Bemenetek

A program egy parancsoros interfacen keresztül használható aminek a standard bemeneten lehet megadni különböző beállításokat Ezeket egy egyszerű kulcs-érték pár listaként várja a következő formában: *KULCS=ÉRTÉK* a párok pedig valamilyen "whitespace" karakterrel kell, hogy elválasztva legyenek. A felhasználó dupla "whitespace" karakter bevitelével tudja jelezni, hogy több beállítást nem szeretne megadni.

1.3 Beállítások

Kulcs / Tipus	Leírás
INTERACTIVE_MODE (Bool)	A program interaktív módban vagy limit* módban fusson.
CYCLE_LIMITS (Int)	Maximum mennyi ciklusig tarthat a szimuláció.
SEED (Int)	Mi legyen a random *seed***
LOG_LEVEL (Int)	Mennyire legyen a standard kimenetre való naplózás részletes.
TRADER_COUNT (Int)	Hány random generált kereskedők legyen.
COMPANY_COUNT (Int)	Hány random generált cég legyen.
EARNINGS_CYCLES (Int)	Mennyi ciklusonként legyen "earnings" jelentése a cégeknek átlagosan.
DIVIDEND_CYCLES (Int)	Mennyi ciklusonként legyen dividáns fizetés átlagosan.
TRADER_MONEY (Int)	Mennyi legyen a kereskedők átlag kezdő vagyona.
TRADER_INCOME (Int)	Mennyi legyen a kereskedők átlag keresete.
MEDIAN_IPO (Int)	Mennyi legyen a medián kezdetleges részvénykibocsájtás részvényenkénti ára.
PRICE_SAMPLING_RATE (Int)	A szimuláció hűny ciklusonként mintavételzze a cégek árfolyamát. (Kimenet felbontása)

** - Limit módban a program, menü megjelenítése nélkül CYCLE_LIMITS-nyi cikluson keresztül fut aztán kilép.*
*** - A programban ez az egyetlen véletlenszerű forrás, egyébként azonos seedekkel determinisztikusan működik.*

Az összes beállítás opcionális és implementáció függő alapértelmezett értékekkel rendelkezik. Az implementáció támogathat még ezeken kívül más beállításokat is. Illetve a program "–help" parancssori argomentummal történő futtatására kiírja a támogatott beállításait és egy rövid használati útmutatót.

1.4 Interaktív mód

Interaktív módban való futtatáskor a szimulácó aktuális állapota egy menü segítségével megfigyelhető. A továbbiakban az ebből a menüből elérhető funkciókat ismertetem:

1.4.1 Szimuláció megállítása/elindítása

El lehet indítani a szimulációt, a ciklusok addig fognak futni ameddig a felhasználó be nem ír valamit a standard bemenetre, illetve előre is meg lehet adni, hogy hány ciklust menjen mielőtt újra megállna. A programot véglegesen is le lehet állítani.

1.4.2 Statisztikák lekérése

Meg lehet tekinteni aktuális statisztikákat a szimulációról, mennyi pénz kering a rendszerben, melyik a legnagyobb cék illetve, ki a leggazdagabb kereskedő, hányadik ciklusban van.

1.4.3 Cégek listázása

Ki lehet listázni a cégeket, szimbólumaikkal, teljes nevükkel illetve jelenlegi árfolyamaikkal.

1.4.4 Cégek részletes adatai

Le lehet kérni egy cég részletes adatait, rejtett tulajdonságaival és visszamenőleges árfolyam adatokkal szimbólum alapján.

1.4.5 Kereskedők listázása

Ki lehet listázni a kereskedőket, nevükkel, egyenlegeikkel, portfólióik méretével illetve egyedi azonosítójaikkal.

1.4.6 Kereskedők részletes adatai

Le lehet kérni egy kereskedő részletes adatait, beállításait, összes nyitott pozícióját a kereskedő egyedi azonosítója alapján.

1.5 Kimenetek

A szimuláció végeztével, a program az elmentett árfolyam, cég és kereskedő információt kiírja JSON formátumban. A konkrét formátum implementáció függő de alapvetőn hasonló adatokat tartalmaz, mint ami az interaktív menüben elérhető, csak teljesen részletesen.

2 Pontosított feladatspecifikáció

A laborvezetőnek eddig semmilyen változtatási igénye nem merült fel, ezért az munkát az eredeti tervek alapján folytattam. A következőkben az implementáció részleteivel fogok foglalkozni.

A feladatban elkészített osztályok mezői és függvényei már megtervezésre kerültek, de mivel egy szimulációs feladatról beszélünk, az összes algoritmus konkrét implementációja nem tervezhető meg előre. Mivel ez gyakorlatilag a teljes feladat befejezését igényelné, minden componens belső működése előre nem látható hatásokkal lehet a többi komponens működésére. Ez a tulajdonság teszi magát a szimulációt érdekessé.

A feladat megoldásához nem használok **STL** tárolókat a szálkezeléshez szükséges *std::mutex* osztályon, és más ehhez kapcsolódó és szükséges primitíveken kívül. (Illetve még a *std::function* segéd tipust is esedlegesen alkalmazom de ez nem valódi tároló.)

3 Terv

Az implementáció két részből áll, egy futtatható standard bemenetetn keresztül irányítható programrészből, ami magát a szimulációt is futtatja (ez a program készült végfelhasználásra), illetve egy test programból ami a fordításnál definiálandó *TEST_VAR* szimbólum *ETEST*-értékre való beállításával elérhető. A testprogram a szükséges tárolókat, segéd osztályok működését teszteli, nem célja a teljes kódlefedettség. A program nem könyvtárként való használatra készült, ezért minden hiba amit dob az *std::runtime_error* osztály példánya.

3.1 Objektum Terv

3.1.1 Könyvtár - Terv

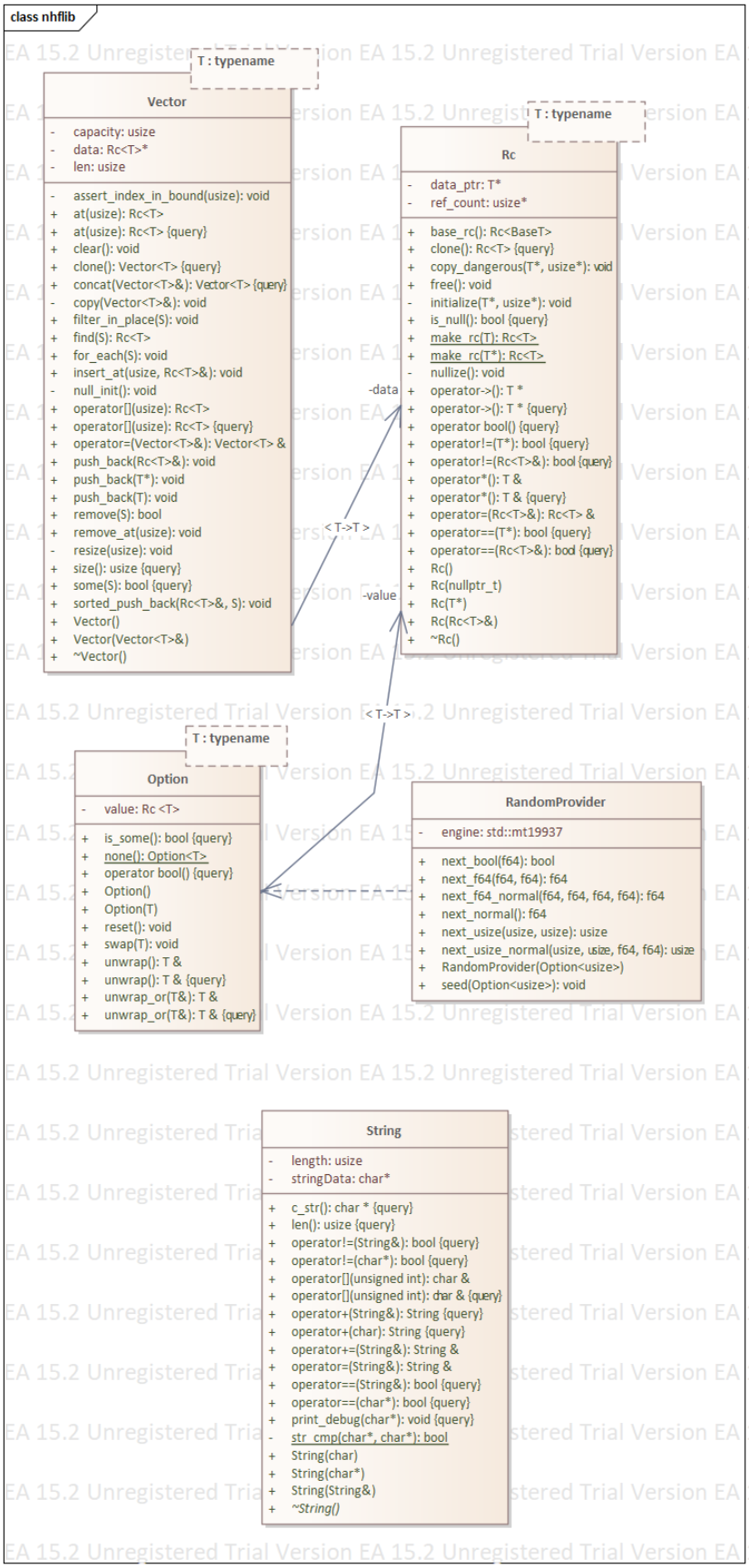
A program egy kisebb sablon könyvtárat tartalmaz, a könyebb fejleszhetőség érdekében. A könyvtár ujradefiniáls numerikus tipusokat is, hogy azok könyebben használhatóak legyenek. Az alábbiak ezek a definíciók:

```
typedef std::size_t usize;
typedef unsigned int u32;
typedef uint8_t u8;
typedef uint16_t u16;
typedef uint32_t u32;
typedef uint64_t u64;
```

```
typedef int8_t i8;
typedef int16_t i16;
typedef int32_t i32;
typedef int64_t i64;

typedef float f32;
typedef double f64;
typedef long double f128;
```

Az itt definiált osztályok definícióját és egymáshoz való függőségeit az alábbi UML diagram ábárzolja:


class nhflib
T : typename
Vector
- capacity: usize
- data: Rc<T>*
- len: usize
- assert_index_in_bound(usize): void
+ at(usize): Rc<T>
+ at(usize): Rc<T> {query}
+ clear(): void
+ clone(): Vector<T> {query}
+ concat(Vector<T>&): Vector<T> {query}
- copy(Vector<T>&): void
+ filter_in_place(S): void
+ find(S): Rc<T>
+ for_each(S): void
+ insert_at(usize, Rc<T>&): void
- null_init(): void
+ operator[](usize): Rc<T>
+ operator[](usize): Rc<T> {query}
+ operator=(Vector<T>&): Vector<T> &
+ push_back(Rc<T>&): void
+ push_back(T*): void
+ push_back(T): void
+ remove(S): bool
+ remove_at(usize): void
- resize(usize): void
+ size(): usize {query}
+ some(S): bool {query}
+ sorted_push_back(Rc<T>&, S): void
+ Vector()
+ Vector(Vector<T>&)
+ ~Vector()
T : typename
Rc
- data_ptr: T*
- ref_count: usize*
+ base_rc(): Rc<BaseT>
+ clone(): Rc<T> {query}
+ copy_dangerous(T*, usize*): void
+ free(): void
- initialize(T*, usize*): void
+ is_null(): bool {query}
+ make_rc(T): Rc<T>
+ make_rc(T*): Rc<T>
- nullize(): void
+ operator->(): T *
+ operator->(): T * {query}
+ operator bool() {query}
+ operator!=(T*): bool {query}
+ operator!=(Rc<T>&): bool {query}
+ operator*(): T &
+ operator*(): T & {query}
+ operator=(Rc<T>&): Rc<T> &
+ operator==(T*): bool {query}
+ operator==(Rc<T>&): bool {query}
+ Rc()
+ Rc(nullptr_t)
+ Rc(T*)
+ Rc(Rc<T>&)
+ ~Rc()
-data
< T->T >
-value
< T->T >
T : typename
Option
- value: Rc <T>
+ is_some(): bool {query}
+ none(): Option<T>
+ operator bool() {query}
+ Option()
+ Option(T)
+ reset(): void
+ swap(T): void
+ unwrap(): T &
+ unwrap(): T & {query}
+ unwrap_or(T&): T &
+ unwrap_or(T&): T & {query}
RandomProvider
- engine: std::mt19937
+ next_bool(f64): bool
+ next_f64(f64, f64): f64
+ next_f64_normal(f64, f64, f64, f64): f64
+ next_normal(): f64
+ next_usize(usize, usize): usize
+ next_usize_normal(usize, usize, f64, f64): usize
+ RandomProvider(Option<usize>)
+ seed(Option<usize>): void
String
- length: usize
- stringData: char*
+ c_str(): char * {query}
+ len(): usize {query}
+ operator!=(String&): bool {query}
+ operator!=(char*): bool {query}
+ operator[](unsigned int): char &
+ operator[](unsigned int): char & {query}
+ operator+(String&): String {query}
+ operator+(char): String {query}
+ operator+=(String&): String &
+ operator=(String&): String &
+ operator==(String&): bool {query}
+ operator==(char*): bool {query}
+ print_debug(char*): void {query}
- str_cmp(char*, char*): bool
+ String(char)
+ String(char*)
+ String(String&)
+ ~String()

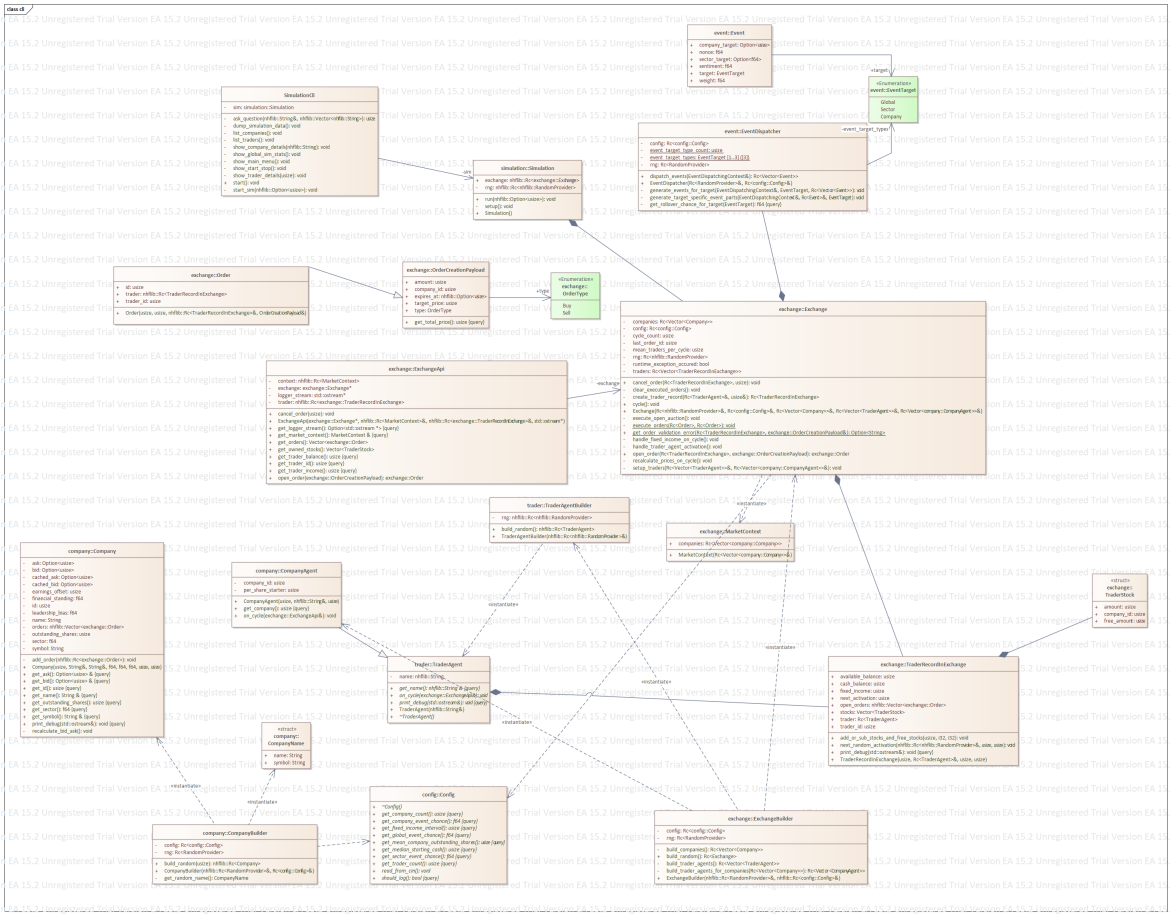
3.1.2 Rc<T> - Implementáció

Egy reference számolt, T tipusu dinamikusan tárolt pointert tartalmaz. A reference számolást egy dinamikusan tárolt *usize* pointerrel valósítja meg. Egy Rc<T> másolásánál, a reference számláló növelve van eggyel illetve a pointerek másolva vannak. Destrukciónál a reference számláló csökkentve van egyel, és a belsőleg tárolt adat csak akkor van felszabadítva ha a reference számláló nulla.

3.2 Implementációs - Terv

A szimuláció főleg kompozícióval épül fel, a proram a *SimulationCli* osztály példányának létrehozásával és a *start* függvényének meghívásával indul el. A szimuláció osztály *run* függvénye egy második szállon fut, és egy mutex segítségével van a futási állapota szinkronizálva a CLI-vel.
Alapvetően a szimulációban lévő kereskedők egy heterogén kollekcióban vannak tárolva, minden osztály ami kompatibilis a *TraderAgent* osztállyal működhet kereskedő "inteligenciaként". A cégek egy speciális *TraderAgent* implementáción keresztül adják el az első részvényeiket (IPO szimulácóió amit brókerek kezelnek).
A kereskedők random időközönként aktiválódnak és az *on_cycle* fügvényük meghívódik, a paraméterként kapott *ExchangeApi*-n keresztül tudnak megnyitni új pozíciókat és információt szerezni a piaci körülményekről a döntéshozáshoz.
Alapvető feltételezés, hogy a *TraderAgent* nem megbízható, tehát ha tud akkor akár csalni is fog, ezért úgy került megtervezésre, hogy a benne futó kódban ne kelljen megbízni. Nem dobhat hibákat, és semmilyen "szenzitív" adathoz nem fér hozzá.

Az osztálydiagram sokkal beszédesebben leírja a különböző kompozíciókat:



3.3 Skeleton Program Jelenlegi állása

A program jelenleg előrehaladott az implementáció terén, ami még hátravan, az a kereskedői logika véglegesítése és a CLI interfész teljes lefejlesztése. A testprogram még hiányosságokkal rendelkezik, a piac nem minden funkciójára van teljese lefedettség.