СУ „Св. Климент Охридски“, ФМИ

Специалност „Софтуерно Инженерство“

Обектно-ориентирано програмиране, 2020-2021 г.

Задача за домашно № 4

Срок: 06.06.2021 г. 23:59

Важна информация

Инструкции

1. *Позволено е използването на всички библиотеки от STL*
2. *За да компилира кодът ви трябва всички методи да имат имплементация, дори да връщат грешен отговор.*
3. *Позволено е да добавяте други методи/класове, за да реализирате задачата.*
4. *Не е позволено използването на външни библиотеки (които не са част от STL)*
5. *На някои от задачите, може да се очаква да се допълнят някои класове, или пък да се създадат нови.*

Оценяване на домашното

- Част от точките за това домашно ще бъдат давани след покриването на автоматични тестове - за **коректно реализирана функционалност**
- За да получите тези точки, предадените от вас решения трябва да отговарят на следните критерии
    - Да съдържат указаните методи и имена на класове (ще ви бъде даден шаблон, върху който да работите) - позволено е да добавяте нови методи и класове.
    - Предавайте единствено файлове съдържащи код - архиви съдържащи `.sln` файлове или каквито и да е други файлове, които не са `.cpp` или `.hpp` ще получават 0 точки на автоматичните тестове.
    - **Не предавайте** архиви от тип `.rar` - **ще се приемат архиви от тип `.zip`.** При получен архив от тип `.rar` (или друг тип, които не може да бъде разархивиран от системата за тестване), отново получавате 0 точки на автоматичните тестове.
    - Именувайте архива си по следния начин - `SI_R_HW4_<курс>_<група>_<факултетен номер>`. Архиви, които не

спазват тази конвенция ще получат 0 точки на автоматичните тестове. (Пример: `SI_R_HW4_1_1_12345.zip`)
- Не променяйте имената на файлове, които получавате
- След разархивиране на архива, трябва да се получат 2 папки, с имена ‘1’ и ‘2’
- Може да тествате архивите си тук: [линк]
- Спазвайте следната структура на архива:

```
SI_R_HW4_<курс>_<група>_<факултетен номер>.zip
├── 1
│   ├── Averager.hpp
│   ├── Averager.cpp
│   ├── BacklogPublisher.hpp
│   ├── BacklogPublisher.cpp
│   ├── Message.hpp
│   ├── MovingAverager.hpp
│   ├── MovingAverager.cpp
│   ├── PeriodicSampler.hpp
│   ├── PeriodicSampler.cpp
│   ├── Repository.hpp
│   ├── Repository.cpp
│   ├── SimplePublisher.hpp
│   ├── SimplePublisher.cpp
├── 2
│   ├── Comparable.hpp
│   ├── Debug.hpp
│   ├── Document.hpp
│   ├── Document.cpp
│   ├── KeyValueDatabase.hpp
│   ├── KeyValueDatbse.cpp
│   ├── Object.hpp
│   ├── Object.cpp
│   ├── Serializable.cpp
│   └── Serializable.hpp
```

- Ако решението на някоя задача ви не се компилира, получавате 0 точки на автоматичните тестове за съответната задача
- Спазвайте практиките за обектно-ориентирано програмиране, коментирани на упражнения и лекции.

Задача 1 (6 точки - 3.6 точки от автоматични тестове)

Линк към header файлове

В програмите, които един софтуерен разработчик създава, често се налага да се наблюдава настъпването на различни събития или промяната на някакви стойности.
Като прост пример може да си представим един сензор, който измерва и излъчва различни стойности през интервали от време. Кой обаче наблюдава тези стойности? На теория може **неограничен** брой "наблюдатели" да се "закачат" към сензора с цел получаване на всяка нова измерена стойност. Могат да съществуват обаче различни начини както за наблюдаване, така и за изпращане на стойности. В тази задача ще направим няколко такива, с помощта на предоставените хедъри и условията към тях, описани долу.

Данните

`Message`

Класът с данни се казва `Message` (съобщение) и съдържа конструктор с параметър от тип `int`, който запазва стойността му в публично поле от същия тип с името `data`. Стойността на това поле, веднъж инициализирана, не трябва да може да бъде променяна.

Класове, които излъчват данни (Publishers)

`SimplePublisher`

- методът `subscribe` позволява закачането на още един **конкретен** наблюдател към инстанцията
- методът `unsubscribe` позволява разкачането на **конкретен** наблюдател от инстанцията
- методът `signal` изпраща параметъра от тип `Message` на всички наблюдатели, закачени към инстанцията, чрез извикване на техния метод `signal`

`BacklogPublisher`

- методът `subscribe` позволява закачането на още един **конкретен** наблюдател към инстанцията, както и веднага му изпраща абсолютно всички предишни ("пропуснати") съобщения чрез `signal` (в реда, в който са били получени)

- методът `unsubscribe` позволява разкачането на **конкретен** наблюдател от инстанцията
- методът `signal` изпраща параметъра от тип `Message` на всички наблюдатели, закачени към инстанцията, чрез извикване на техния метод `signal`

Hint 1: Думата "конкретен" не е случайна и подсказва нещо, което ще ви улесни в имплементацията на методите - Този обект не трябва да е силно обвързан с класа Publisher.

Класове, които наблюдават данни (Subscribers)

Всеки един от тях трябва да съдържа публично поле с име `id` от тип `std::string` с цел идентификация (два наблюдетеля с едно и също `id` да се смятат за равни). Стойността на полето, веднъж инициализирана, не трябва да може да бъде променяна.

Получаването на съобщения става чрез `signal`, а четенето им след това - чрез `read`. Разликата между трите класа се проявява при четенето и обработването на получените данни.

При никакви получени данни и трите класа връщат 0 при четене.

`Averager`

- При четене връща средната стойност на всички данни, получени досега.

`MovingAverager`

- Съдържа публично поле с име `windowSize` от тип `size_t`. Стойността на това поле, веднъж инициализирана, не трябва да може да бъде променяна.
- При четене връща средната стойност само на последните `windowSize` на брой съобщения.

`PeriodicSampler`

- Приема в конструктор `period` от тип `size_t`
- При четене връща стойността на последното получено неигнорирано съобщение. За игнорирано съобщение смятаме всяко `n`-то съобщение след първото получено, за което `n % period != 0`

Пример:

```
Averager* avg = new Averager("id1");
```

```
MovingAverager* movAvg = new MovingAverager("id2", 5);

PeriodicSampler* perSam = new PeriodicSampler("id3", 3);



SimplePublisher pub;

pub.subscribe(avg);

pub.subscribe(movAvg);

pub.subscribe(perSam);



pub.signal(1);

pub.signal(2);

pub.signal(3);

pub.signal(4);

pub.signal(5);

pub.signal(6);

pub.signal(7);

pub.signal(8);

pub.signal(9);



avg.read(); // трябва да връща 5 (ср. аритм. на 1...9)

movAvg.read(); // трябва да връща 7 (ср. аритм. на 5...9)

perSam.read(); // трябва да връща 7 (стойността на шестото съобщение след
първото; игнорира стойности 8 и 9)
```

Клас, който управлява наблюдатели

Клас `Repository` служи като хранилище за различни наблюдатели, независимо от това от къде те получават своите данни.

Поддържа операциите добавяне (създаване) на нов наблюдател и достъп до наблюдател чрез неговото `id`.

Примери за употребата му разгледайте в приложения `main.cpp` файл.

Hint 2: Тук думата "конкретен" нарочно липсва.

Hint 3: Позволено е решаването на задачата по такъв начин, че вместо `static_cast` да е необходимо използването на `dynamic_cast` в приложените примери.

Hint 4: Във всички хедъри има коментари с уточнения и подсказки, разгледайте ги внимателно в случай, че нещо не ви е ясно.

Hint 5: Позволено е (и винаги е било) да променяте всеки един ред, който сме ви дали в хедър файловете. За да компилира решението с тестовете е необходимо просто да не се създават т.нар. "breaking changes", т.е. да не се променят имената на дадените функции, полета и класове и т.н.

Hint 6: Ако не ви харесва употребата на `void*` нямате проблеми да използвате нещо друго - той просто служи като пойнтър, който може да бъде кастнат към всичко на теория.

Задача 2: (6 точки - 3.6 точки от автоматични тестове)

Линк към header файловете

Един от най-използваните елементи от ООП-то, са т.нар. **интерфейси** и **абстрактни класове**.
В езика C++ те не съществуват в същия вид, както например в езиците C#/Java, целта на тази задача е да се създадат такива подобни.

Comparable

Ще играе ролята на интерфейс, чрез който ще показваме, че даден клас може да се сравнява.

- Предефинирайте операторите `==` и `!=` като абстрактни функции, приемащи указател от тип `Comparable`

Debug

Ще играе ролята на интерфейс, чрез който ще даваме възможността, да принтираме допълнителна информация за
състоянието на даден клас, с цел откриване на грешки.

- Създайте абстрактна функция `debug_print`, която да връща `string` обект.

Serializable

Ще играе ролята на интерфейс, показващ че даден клас може да се сериализира (да се превърне до низ), както и
десериализира (че може да се създаде обект, по подаден низ).

- Създайте абстрактна функция `to_string`, която връща `string` обект.
- Създайте абстрактна функция `from_string`, която приема `string` обект, и модифицира текущия обект, спрямо подадения низ.

Пример:

Ако имаме дадения клас Foo:

```
class Foo{

public:

   int value;

};
```

`to_string` може да връща стойността на `value`, а `from_string` ще приема един такъв низ, и ще променя
стойността на `value` спрямо съдържанието на низа.

Бележка: За тези класове са предоставени header файлове, които съдържат празен клас

Всеки един от тези класове, ще бъдат наследени от един специален клас **Object**

Object

Този абстрактен клас ще играе ролята на база, върху която ще стъпят всички други класове в задачата.
За целите на конкретната задача, той ще съдържа и някои член-данни.

- Наследява `Comparable`, `Debug` и `Serializable`
- Конструктор с параметри
- Метод `clone`, който прави дълбоко копие на текущия обект. Връща се `Object*`
- Име на файл (name)
- Локация на файл (location)
- Разширение на файл (extension)
- Метод за създаване на пълен път на файла
    - Формат: `<location>/<name>.<extension>`

Бележка: За този клас е предоставен непълен header файл, който трябва да се допълни/разшири

С помощта на `Object` ще създадем още два класа - **Document** и **KeyValueDatabase**

Document

Играе ролята на текстов документ, където можем да пишем и четем ред по ред

- Наследява `Object`
- Пази в себе си колекция от низове (един низ е равен на един ред)
- Метод за писане във документа `write_line`
- Методи за четене в документа. Документа се чете ред по ред, като се "помни" до къде сме прочели документа
    - Метод `read_line`, който връща следващия ред, който трябва да се "прочете"
    - Метод `read_line`, приемащ аргумент кой ред да се прочете. След извикването на този метод, четенето на документа продължава от подадения ред
    - Ако не може да се прочете ред (например сме прочели документа до край), да се хвърля `std::out_of_range` изключение
    - Броенето на редовете започва от 1
- Два документа са равни, когато техните редове са едни и същи *(бележка: ако редовете на документ 1 е подмножество на редовете на документ 2, документ 1 НЕ Е равен на документ 2)*
- Сериализирането на един документ се извършва по следния начин:
    - На първите три реда се записва името, локацията и разширението на документа
    - На следващите реда, се записва съдържанието на документа
    - След последния ред от документа, при сериализиране се очаква да има нов ред също
- Десериализирането на един документ се извърша по същия начин

- Принтирането на допълнителната информация се извършва по следния начин:
    - За всеки ред от съдържанието, се записва нов ред `Line <номер_на_реда>:<ред>`
    - След последния ред се очаква да има нов ред

Примерно извикване:

```
Document temp("temp", "/tmp", "doc");



temp.write_line("This is an example doc");

temp.write_line("This is another example line");

temp.write_line("This is a third example line");

temp.write_line("This is a forth example line");



std::cout << temp.read_line() << std::endl; //This is an example doc

std::cout << temp.read_line() << std::endl; //This is another example line



std::cout << temp.read_line(3) << std::endl; //This is a third example line

std::cout << temp.read_line() << std::endl; //This is a forth example line



std::cout << temp.read_line(1) << std::endl; //This is an example doc

std::cout << temp.read_line() << std::endl; //This is another example line
```

Бележка: За този клас е предоставен непълен header файл, който трябва да се допълни/разшири

KeyValueDatabase

Играе ролята на база от данни, пазеща в себе си двойки от ключ и стойност.

- Наследява `Object`
- Пази в себе си двойки ключ и стойност, където ключът е низ, а стойността - цяло число.

Съдържа метод за добавяне на нова двойка ключ и стойност. Ако ключът вече се съдържа в колекцията,

- се хвърля `std::invalid_argument`

Съдържа метод за взимане на стойност, по подаден ключ. Ако ключът не се съдържа в колекцията,

- се хвърля `std::invalid_argument`
- Две бази са равни, когато техните двойки от ключове и стойности са равни *(бележка: ако двойките ключ/стойност на база1 са подмножество на тези от база2, база 1 НЕ Е равна на база 2)*
- Сериализирането на една база се извършва по следния начин:
    - На първите три реда се записва името, локацията и разширението на документа
    - На следващите редове, се записва ред със следния формат `<ключ>:<стойност>`
    - Очаква се низа да завършва с нов ред
- Десериализирането на една база се случва по същия начин
- Принтирането на допълнителната информация се извършва по следния начин:
    - За всяка двойка, се записва нов ред `{<ключ>:<стойност>}`
    - Очаква се низа да завършва с нов ред

За този клас е забранено използването на `std::map`, `std::unordered_map` и всички други колекции,

- **които предоставят търсена функционалност "наготово" (`std::vector` и други подобни са позволени)**

Бележка: За този клас е предоставен непълен header файл, който трябва да се допълни/разшири