Развој софтвера

Збирка задатака за вежбу

У припреми

Никола Ајзенхамер
Момир Аџемовић

Математички факултет
Београд, 2021.

Електронска верзија у припреми.

Садржај

Глава 1

Управљање динамичким ресурсима

Ово поглавље је у припреми!

Током свог рада, у већини апликација се у неком тренутку јави потреба за управљањем некаквим ресурсима. Пре него што се они могу користити, потребно их је *иницијализовати*. У неком другом тренутку такви ресурси нису више неопходни, те се они морају *деиницијализовати*. Овакве ресурсе називамо *динамичким* у смислу да њихов животни век често није предефинисан. Примери оваквих ресурса су: објекти на динамичкој меморији, датотеке, сокети, конекције на базе података и сл. Приликом имплементирања апликација које користе динамичке ресурсе, програмери су у обавези да прописно иницијализују и деиницијализују те ресурсе, како би онемогућили појаву различитих проблема. У овој глави разматрамо неколико техника за управљање динамичким ресурсима, са акцентом на динамичку меморију у програмском језику C++.

1.1 Динамичка меморија

Задатак 1.1.1. Написати апликацију која међу подацима о правоугаоницима проналази онај са најмањом површином.

- Написати структуру *Тачка* која садржи две координате које описују тачку у равни.
- Написати структуру *Правоугаоник* која садржи две инстанце структуре *Тачка*: једну за горње лево теме, а другу за доње десно теме правоугаоника. Структура има и метод којим се израчунава површина датог правоугаоника.
- Написати апликацију која из датотеке, чија се путања прослеђује као аргумент командне линије, учитава податке о правоугаоницима. Подаци о једном правоугаонику су задати као четири броја (прва два представљају координате горњег левог темена, а друга два координате доњег десног темена) и налазе се у засебној линији. Затим, апликација проналази правогаоник који има најмању површину и исписује координате темена и површину тог правоугаоника. Користити динамичку меморију за смештање објеката структура *Тачка* и *Правоугаоник*. Користити колекције стандардне библиотеке за складиштење објеката.

Решење 1.1.1.

Задатак 1.1.2. Написати апликацију која управља информацијама о студентима и курсевима на које су уписани.

- Написати структуру *Курс* која садржи идентификатор, назив и број ЕСПБ.
- Написати структуру *Студент* која садржи индекс, име, презиме и колекцију курсева на које је тај студент уписан.
- Написати апликацију која са стандардног улаза прво учитава број курсева N, а затим учитава податке за сваки од N курсева. Затим, апликација учитава број студената M и, потом, податке за сваког од M студената. Након учитавања података о једном студенту, апликација захтева да корисник унесе један или више идентификатора курсева, након чега се управо унети студент уписује на одабране курсеве. На крају уноса, исписати на стандардни излазни ток за сваког студента списак уписаних курсева. Користити динамичку меморију за смештање објеката структура *Курс* и *Студент*. Користити колекције стандардне библиотеке за складиштење динамичких објеката.

Решење 1.1.2.

Задатак 1.1.3. Решити задатак 1.1.2 коришћењем динамичких низова уместо колекција стандардне библиотеке.

Решење 1.1.3.

Задатак 1.1.4. Написати апликацију која управља информацијама о студијским програмима и студентима који су уписани на тим студијским програмима.

- Написати структуру *Студент* која садржи индекс, име и презиме.
- Написати структуру *СтудијскиПрограм* која садржи идентификатор, назив, обим ЕСПБ и колекцију студената који су уписани на тај студијски програм.
- Написати апликацију која са стандардног улаза прво учитава број студијских програма N, а затим учитава податке за сваки од N курсева. Затим, апликација учитава број студената M и, потом, податке за сваког од M студената. Након учитавања података о једном студенту, апликација захтева да корисник унесе један идентификатор студијског програма, након чега се управо унети студент уписује на одабрани студијски програм. На крају уноса, исписати на стандардни излазни ток за сваки студијски списак уписаних студената. Користити динамичку меморију за смештање објеката структура *Студент* и *СтудијскиПрограм*. Користити колекције стандардне библиотеке за складиштење динамичких објеката.

Решење 1.1.4.

Задатак 1.1.5. Решити задатак 1.1.4 коришћењем динамичких низова уместо колекција стандардне библиотеке.

Решење 1.1.5.

Задатак 1.1.6. Написати апликацију која симулира игру са турнирским биткама.

- Написати структуру *Борац* која представља једног борца одређеног именом (ниска), типом (карактер који може да буде `'M'` (маг), `'W'` (ратник) ili `'A'` (стрелац)) и снагом (позитиван цео број).

- Написати структуру *Турнир* која чува низ бораца и имплементира методу којом се иницијализује борба између парних и непарних бораца у низу. Борбу побеђује борац који који има већу снагу. Маг има дуплу снагу над ратником, ратник има дуплу снагу над стрелчем, и стрелац има дуплу снагу над магом.

- Написати апликацију која из датотеке, чије се име наводи као аргумент командне линије, учитава број бораца N, а затим N редова који се састоје од имена, карактера (типа) и снаге бораца. На стандардни излаз исписати име победника турнира коришћењем структуре *Турнир*. Користити динамичку меморију за смештање објеката структура *Борац* и *Турнир*.

Решење 1.1.6.

Задатак 1.1.7. Написати апликацију за сабирање великих бројева.

- Написати структуру *Број* која се састоји из низа цифара произвољне димензије.

- Написати структуру *Сабирач* која садржи низ великих бројева (структура *Број*) и може да рачуна збир свих бројева које садржи.

- Написати апликацију која са стандардног улазног тока учитава број бораца N, а затим N ниски које садрже само цифре. Коришћењем структуре *Сабирач*, на стандрадни излаз исписати резултат сабирања свих бројева. Користити динамичку меморију за смештање објеката структура *Број* и *Сабирач*.

Решење 1.1.7.

Задатак 1.1.8. Написати апликацију која броји речи у неком тексту.

- Написати структуру *Чвор* која представља елемент дрволике структуре података. Објекти ове структуре садрже ниску која представља реч и ненегативан цели број који представља број појављивања те речи у тексту.

- Написати структуру *Речник* која имплементира дрволику структуру података за претрагу речи, чији су елементи објекти структуре *Чвор*. Структура треба да подржава операције додавања појављивања неке речи, дохватања броја појављивања неке речи, као и израчунавање свих речи које су се најмање и највише пута појављивале у тексту.

- Написати апликацију која са стандардног улазног тока учитава текст, а затим, коришћењем структуре *Речник* проналази речи које су се најмање и највише пута појављивале у тексту. Користити динамичку меморију за смештање објеката структура *Чвор* и *Речник*.

Решење 1.1.8.

Задатак 1.1.9. Написати апликацију која броји речи у неком тексту.

- Написати структуру *Чвор* која представља елемент листе (корен листе у специјалном случају). Објекти ове структуре садрже ниску која представља реч, ненегативан цели број који представља број појављивања те речи у тексту и следећи елемент у листи.
- Написати структуру *Кофа* која представља листу елемената *Чвор*. Ова структура подржава операције додавања у кофу, где се ажурира одговарајући чвор ако постоји реч у листи, а иначе додаје нови елемент за ту реч у листу. Такође је подржава операција проналаска речи која се појављује најмањи број пута у листи.
- Написати структуру *ХешРечник* која имплементира хеш структуру података за претрагу речи, чији су елементи објекти структуре *Кофа*. Приликом иницијализације структуре *ХешРечник*, бира се број елемената M. Структура треба да подржава операције додавања неке речи, где се реч додаје у одговарајућу листу (кофу) помоћу хеш функције $S \ modul \ M$, где је S сума карактера речи.
- Написати апликацију која са стандардног улазног тока учитава текст, а затим, коришћењем структуре *ХешРечник* проналази речи које су се најмање пута појављивале у тексту. Користити динамичку меморију за смештање објеката структура *Чвор*, *Кофа* и *ХешРечник*.

Решење 1.1.9.

1.2 Дељени показивачи

Задатак 1.2.1. Решити задатак 1.1.1 коришћењем дељених показивача.

Решење 1.2.1.

Задатак 1.2.2. Решити задатак 1.1.2 коришћењем дељених показивача.

Решење 1.2.2.

Задатак 1.2.3. Решити задатак 1.1.4 коришћењем дељених показивача.

Решење 1.2.3.

1.3 Јединствени показивачи

Задатак 1.3.1. Решити задатак 1.1.6 коришћењем јединствених показивача.

Решење 1.3.1.

Задатак 1.3.2. Решити задатак 1.1.7 коришћењем јединствених показивача.

Решење 1.3.2.

Задатак 1.3.3. Решити задатак 1.1.8 коришћењем јединствених показивача.

Решење 1.3.3.

Задатак 1.3.4. Решити задатак 1.1.9 коришћењем јединствених показивача.

Решење 1.3.4.

1.4 Идиом RAII

Задатак 1.4.1. Решити задатак 1.1.1 коришћењем идиома RAII.

Решење 1.4.1.

Задатак 1.4.2. Решити задатак 1.1.2 коришћењем идиома RAII. Допунити имплементацију тако да се, пре учитавања података са стандардног улаза, учитају подаци из датотека *kursevi.txt* и *studenti.txt*. Након учитавања нових података и исписивања свих података, сачувати све податке у исте датотеке.

Решење 1.4.2.

Задатак 1.4.3. Решити задатак 1.1.4 коришћењем идиома RAII. Допунити имплементацију тако да се, пре учитавања података са стандардног улаза, учитају подаци из датотека *studijskiPrograms.txt* и *studenti.txt*. Након учитавања нових података и исписивања свих података, сачувати све податке у исте датотеке.

Решење 1.4.3.

Задатак 1.4.4. Нека је дата структура *Тачка* која представља дводимензионалну тачку у равни. Написати структуру *Раван* која имплементира:

- Метод за инстанцирање насумичних тачака у равни на динамичкој меморији.
- Метод за проналажење тачке која је најближа координатама које се прослеђују методу.

Осигурати се да структура *Раван* буде безбедна за управљање динамичким ресурсима.

```
struct Tacka
{
    double _x;
    double _y;
};
```

Решење 1.4.4.

Задатак 1.4.5. Нека је дата функција *обрадиБројеве* која врши учитавање целих бројева из датотеке која се наводи као аргумент функције, а затим исписује збир учитаних бројева. Учитавање негативног целог броја се сматра грешком. Написати структуру *ЗбирБројева* која имплементира функционалност дате функције тако да буде безбедна за управљање динамичким ресурсима.

```
void obradiBrojeve(const std::string &putanjaDoDatoteke)
{
    int broj, suma = 0;
    std::ifstream datoteka(putanjaDoDatoteke);
    while (datoteka >> broj)
    {
        if (broj < 0)
        {
            throw std::exception("Datoteka ne sme da sadrzi negativan
                ↪ broj");
        }
        suma += broj;
```

```
    }
    std::cout << "Zbir brojeva je: " << suma << std::endl;
    datoteka.close();
};
```

Решење 1.4.5.

Задатак 1.4.6. Нека су дате функције за креирање и раскидање конекције на неки систем за управљање базама података. Функције враћају показивач на структуру *Конекција* која садржи метод за извршавање упита. Приликом извршавања наредбе је могуће доћи до грешке у систему за управљање базама података. Написати структуру *БазаПодатака* која замењује употребу датих функција и структура тако да буде безбедна за управљање динамичким ресурсима.

```
struct Konekcija
{
    void IzvrsiNaredbu(const std::string &naredba) const
    {
        std::mt19937 rng(std::random_device{}());
        auto rand_bool = static_cast<bool>(std::
            ↪ uniform_int_distribution<>{0, 1}(rng));
        if (rand_bool)
        {
            throw std::exception("SQL greska");
        }
        std::cout << "Naredba" << naredba << " je uspesno izvrsena!"
            ↪ << std::endl;
    }
};

Konekcija *KreirajKonekciju()
{
    return new Konekcija;
}

void KreirajKonekciju(Konekcija *konekcija)
{
    delete konekcija;
}
```

Решење 1.4.6.

1.5 Решења задатака

Динамичка меморија

Решење 1.1.1.

Задатак 1.1.1.

Решење 1.1.2.

Задатак 1.1.2.

Решење 1.1.3.

Задатак 1.1.3.

Решење 1.1.4.

Задатак 1.1.4.

Решење 1.1.5.

Задатак 1.1.5.

Решење 1.1.6.

Задатак 1.1.6.

Решење 1.1.7.

Задатак 1.1.7.

Решење 1.1.8.

Задатак 1.1.8.

Решење 1.1.9.

Задатак 1.1.9.

Дељени показивачи

Решење 1.2.1.

Задатак 1.2.1.

Решење 1.2.2.

Задатак 1.2.2.

Решење 1.2.3.

Задатак 1.2.3.

Јединствени показивачи

Решење 1.3.1.

Задатак 1.3.1.

Решење 1.3.2.

Задатак 1.3.2.

Решење 1.3.3.

Задатак 1.3.3.

Решење 1.3.4.

Задатак 1.3.4.

Идиом RAII

Решење 1.4.1.

Задатак 1.4.1.

Решење 1.4.2.

Задатак 1.4.2.

Решење 1.4.3.

Задатак 1.4.3.

Решење 1.4.4.

Задатак 1.4.4.

Решење 1.4.5.

Задатак 1.4.5.

Решење 1.4.6.

Задатак 1.4.6.

Глава 2

Објектно-оријентисане технике

Ово поглавље је у припреми!

У овој глави разматрамо неколико објектно-оријентисаних техника. Једна од популарних техника су UML дијаграми који нам омогућују једноставност комуникације између чланова тима путем графичког језика, односно, дијаграма. Овим дијаграмима можемо једноставно записати захтеве апликација, а затим их искористити за њихову имплементацију. Стандардизованост значења UML дијаграма води ка униформном раду на пројекту. Након тога, прелазимо на имплементацију класа у програмском језику C++, осврћући се на специфичности овог језика. Затим прелазимо на имплементацију односа између класа, са посебним акцентом на хијерархије класа и вишеврсност у понашању које оне омогућавају (тзв. хијерархијски полиморфизам). Коначно, главу завршавамо темом о вишеструком наслеђивању, предностима и манама његовог коришћења.

2.1 UML

У игри „Брзо куцање” корисници имају за циљ да откуцају што већи број речи у унапред дефинисаном временском интервалу. Речи се појављују са врха екрана и падају на доле. Ако корисници успеју да откуцају реч пре него што она дотакне дно екрана, они освајају поене који зависе од дужине те речи. У супротном, они губе поене који зависе од дужине речи. Речи су подељене по категоријама, као што су: воће, поврће, животиње, технички уређаји и др. Потребно је имплементирати апликацију која имплементира описану игру. Након играња једне партије игре, корисници имају могућност да упишу своје име и апликација памти освојен резултат. Корисници могу видети десет најбољих одиграних партија, за свако од предефинисаних времена трајања партије игре. Такође, апликација омогућава кориснику да одабере категорије речи које ће се појављивати у игри (нека од предефинисаних или да направи своју категорију), као и једно од предефинисаних времена трајања. Додатно, пре започињања нове партије игре, корисници имају могућност да позову друге кориснике који ће пратити прогрес корисника током текуће партије.

Задатак 2.1.1. Препознати случајеве употребе и актере у имплементацији игре „Брзо куцање” и нацртати дијаграм случајева употребе.

Решење 2.1.1.

Задатак 2.1.2. Написати опис случаја употребе „Играње једне партије игре”.

Решење 2.1.2.

Задатак 2.1.3. Нацртати дијаграм секвенци за случај употребе „Играње једне партије игре” (без навођења подслучајева употребе).

Решење 2.1.3.

Задатак 2.1.4. Препознати класе, њихове чланице и везе између класа, па нацртати дијаграм класа за имплементацију игре „Брзо куцање”.

Решење 2.1.4.

Задатак 2.1.5. Написати описе преосталих случајева употребе и нацртати одговарајуће дијаграме секвенци.

Решење 2.1.5.

2.2 Објекти и класе

Задатак 2.2.1. Потребно је омогућити рад са разломцима. Подржати конструкцију разломака, основне аритметичке операције, конверзију у број у покретном зарезу, као и учитавање из улазних токова и исписивање на излазне токове. Нацртати UML дијаграм класа и имплементирати неопходне класе.

Решење 2.2.1.

Задатак 2.2.2. Потребно је омогућити рад са једноструко-повезаном листом. Елементи листе су цели бројеви који се чувају на динамичкој меморији. Подржати конструкцију празне листе, додавање елемената на почетак и крај листе, брисање елемената са почетка и краја листе, израчунавање броја елемената, приступање елементима листе на основу индекса, као и учитавање из улазних токова и исписивање на излазне токове. Нацртати UML дијаграм класа и имплементирати неопходне класе.

Решење 2.2.2.

2.3 Хијерархије класа и хијерархијски полиморфизам

Задатак 2.3.1. Потребно је омогућити рад са подацима о студентима на Математичком факултету. Студенти имају своје име и презиме, ознаку студијског програма и списак оцена. Студенти мастер студија, поред ових информација, имају и назив мастер тезе и име и презиме ментора. Постоје три студијска програма: „Математика”, „Информатика” и „Астрономија”. Подржати израчунавање просечне оцене студената, додавање нове оцене и дохватање свих доступних информација о студентима као ниску. Нацртати UML дијаграм класа и имплементирати неопходне класе.

Решење 2.3.1.

Задатак 2.3.2. Потребно је омогућити рад са подацима о возном парку и возилима која могу бити аутомобили, камиони и бицикла. Сва возила у возном парку имају број точкова и број седишта. Аутомобили и камиони имају и број прозора. Корисници имају могућност добијања информација о овим подацима, као и о називу типа возила. Возни парк може имати произвољан број возила, а могу му се накнадно додавати нова возила. Сва возила припадају једном возном парку. Нацртати UML дијаграм класа и имплементирати неопходне класе.

Решење 2.3.2.

2.4 Вишеструко наслеђивање

Задатак 2.4.1. Потребно је омогућити рад са подацима о поклонима у једној антикварници и њиховим паковањима. Производи имају своје шифре и цену, док су кутије бесплатне. Производи се могу спаковати у кутије које могу бити картонске или стаклене, а такође, могу се спаковати у украсни папир. Украсни папир има шифру шаре. Кутије могу бити отворене на врху или затворене. Картонске кутије могу бити обојене у неку посебну боју или у стандардну неутралну (браон) боју. Стаклене кутије се добијају независно од поклона, док картонске кутије и украсни папир купци добијају са производима који су упаковани у њих. Омогућити израчунавање цене појединачних производа и групе производа које су упаковане у картонску кутију или украсни папир. Нацртати UML дијаграм класа и имплементирати неопходне класе.

Решење 2.4.1.

Задатак 2.4.2. Допунити UML дијаграм класа и имплементацију класа из задатка 2.4.1 тако да се омогући паковање производа у украсну картонску кутију.

Решење 2.4.2.

2.5 Решења задатака

UML

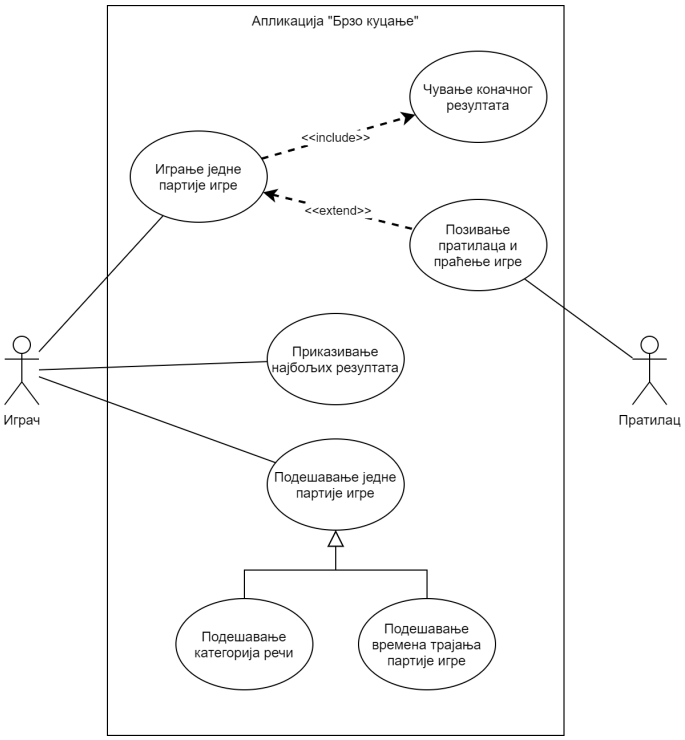
Решење 2.1.1. Случајеви употребе:

- Играње једне партије игре
- Чување коначног резултата
- Приказивање најбољих резултата
- Подешавање једне партије игре
- Подешавање категорија речи
- Подешавање времена трајања партије игре
- Позивање пратилаца и праћење игре

Актери:

- Играч
- Пратилац

Дијаграм случајева употребе:



Задатак 2.1.1.

Решење 2.1.2. Опис случаја употребе „Играње једне партије игре”:

- Кратак опис: Играч започиње једну партију игре из главног менија апликације. Апликација учитава подешавања партије и започиње партију. Након одигране партије, апликација складишти информације о одиграној партији.
- Актери:
  - Играч – игра једну партију игре
  - Пратилац – прати једну партију игре играча
- Предуслови: Апликација је покренута и приказује главни мени.
- Постуслови: Информације о одиграној партији су трајно сачувани.
- Основни ток:
  1. Играч бира дугме „Започни игру” из главног менија.
  2. Апликација приказује дијалог за позивање пратилаца.
     2.1. Ако је Играч одабрао да позове пратиоце:
        2.1.1. Прелази се на случај употребе „Позивање пратилаца”. По завршетку, прелази се на корак 3.
  3. Апликација дохвата информације о подешавањима партије.
  4. Апликација конструише нови тајмер и започиње одбројавање.
  5. Апликација конструише нову партију и започиње је.
  6. Све док не истекне време, понављају се наредни кораци:
     6.1. Апликација проверава да ли има довољно речи на екрану.

6.1.1. Уколико нема довољно речи на екрану, потребно је да прикаже нову реч.

6.1.1.1. Апликација бира једну од преосталих речи.

6.1.1.2. Апликација приказује одабрану реч.

6.1.1.3. Прелази се на корак 6.2.

6.2. Апликација проверава да ли је нека реч достигла дно екрана.

6.2.1. Уколико постоји реч која је достигла дно екрана, потребно је умањити поене.

6.2.1.1. Апликација израчунава казнене поене на основу дужине речи.

6.2.1.2. Апликација умањује укупне поене за израчунати број казнених поена.

6.2.1.3. Прелази се на корак 6.3.

6.3. Играч уноси реч у поље за унос.

6.4. Апликација проверава да ли је играч унео исправну реч међу речима које су приказане на екрану.

6.4.1. Уколико је играч унео исправну реч, потребно је увећати поене.

6.4.1.1. Апликација израчунава добитне поене на основу дужине речи.

6.4.1.2. Апликација увећава укупне поене за израчунати број добитних поена.

6.4.1.3. Прелази се на корак 6.5.

6.4.2. Уколико је играч унето неисправну реч, потребно је обавестити играча.

6.4.2.1. Апликација приказује поруку о неуспешно унетој речи.

6.4.2.2. Прелази се на корак 6.5.

6.5. Уколико има пратилаца партије, апликација обавештава пратиоце о текућем стању.

7. По истеку времена, апликација приказује дијалог са коначним резултатима.

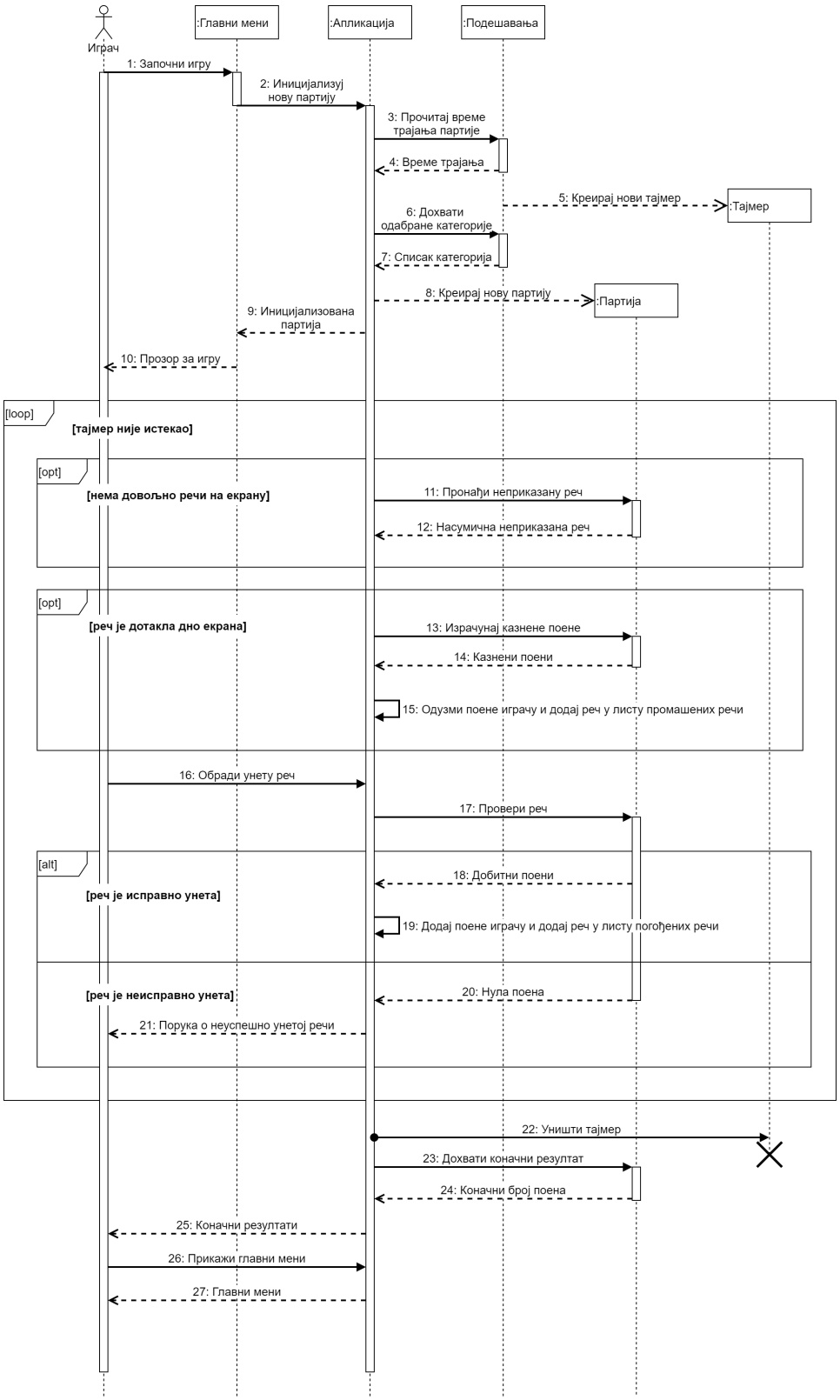
8. Прелази се на случај употребе „Чување коначног резултата”. По завршетку, прелази се на корак 9.

9. Апликација приказује главни мени.

- Алтернативни токови:
  - A1: **Неочекивани излаз из апликације**. Уколико у било којем кораку корисник искључи апликацију, све евентуално запамћене информације о тренутној партији игре се поништавају и апликација завршава рад. Случај употребе се завршава.
- Подтокови: /
- Специјални захтеви: Апликација је повезана на интернет ради комуникације са пратиоцима.
- Додатне информације: Током трајања партије, апликација памти информације о свим речима које је корисник успешно и неуспешно унео.

Задатак 2.1.2.

Решење 2.1.3. Дијаграм секвенци за случај употребе „Играње једне партије игре”:



Задатак 2.1.3.

Решење 2.1.4. Из захтева задатка можемо препознати наредне класе и њихове чланице[1]:

- Играч
  - Име
  - Текући број поена
  - Додај добитне поене
  - Додај казнене поене
- Партија
  - Играч који игра партију
  - Списак пратилаца
  - Списак погођених речи
  - Списак промашених речи
  - Тајмер
  - Додај пратиоца
  - Започни партију
  - Обавести пратиоце о играчевом напретку
  - Заврши партију
  - Прикажи резултат
- Реч
  - Реч
  - Позиција на екрану
  - Израчунај број добитних поена
  - Израчунај број казнених поена
  - Ажурирај позицију речи
- Категорија
  - Назив
  - Списак речи који припадају категорији
  - Индикатор одабраности категорије
  - Додај нову реч
- Складиште
  - Време трајања партије
  - Списак времена трајања партија

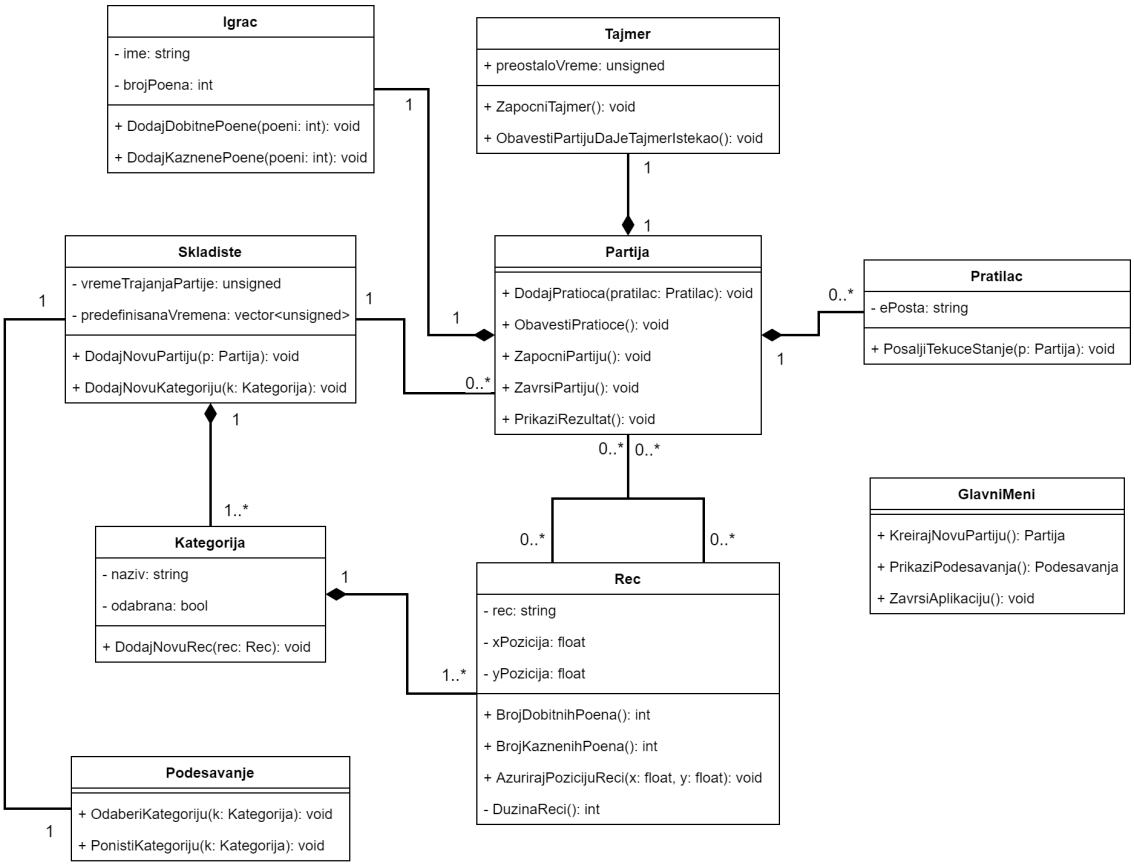
[1] Ово је једна могућа интерпретација захтева задатка. Сва је прилика да би иоле вешт програмер, приликом имплементације класа приметио нека „заударања” у коду (на пример, класа *Партија* је кандидат за „велику класу”), те би приступио измени кода, при чему би имплементација остала функционално-еквивалентна (тзв. *рефакторисање кода*; више о овоме у Глави 4). Додатно, неке од функционалности које су неопходне како би систем функционисао (на пример, централни апликациони подсистем, подсистем за исцртавање, подсистем за управљање датотекама и др.) намерно нису наведене како би се читалац фокусирао на главне класе и њихова задужења у апликацији.

  - Списак одиграних партија
  - Списак категорија речи
  - Додај нову партију
  - Додај нову категорију
- Главни мени
  - Креирај нову партију
  - Прикажи подешавања
  - Заврши апликацију
- Подешавање
  - Складиште
  - Одабери категорију
  - Поништи категорију
- Пратилац
  - Адреса електронске поште
  - Пошаљи податке кроз мрежу
- Тајмер
  - Партија у којој је тајмер активан
  - Преостало време
  - Започни тајмер
  - Обавести партију да је тајмер истекао

Везе између класа:

- Партија има 1 играча. Играч припада 1 партији. Играч не постоји без партије.
- Партија има 0 или више пратилаца. Пратилац припада 1 партији. Пратилац не постоји без партије.
- Партија има 0 или више погођених речи. Погођена реч припада 0 или више партија.
- Партија има 0 или више непогођених речи. Непогођена реч припада 0 или више партија.
- Партија има 1 тајмер. Тајмер припада 1 партији. Тајмер не постоји ван партије.
- Категорија има 1 или више речи. Реч припада 1 категорији. Реч не постоји без категорије.
- Складиште има 0 или више партија. Партија припада 1 складишту.
- Складиште има 1 или више категорија. Категорија припада 1 складишту. Категорија не постоји без складишта.

Дијаграм класа:



Задатак 2.1.4.

Решење 2.1.5.

Задатак 2.1.5.

Објекти и класе

Решење 2.2.1.

Задатак 2.2.1.

Решење 2.2.2.

Задатак 2.2.2.

Хијерархије класа и хијерархијски полиморфизам

Решење 2.3.1.

Задатак 2.3.1.

Решење 2.3.2.

Задатак 2.3.2.

Вишеструко наслеђивање

Решење 2.4.1.

Задатак 2.4.1.

Решење 2.4.2.

Задатак 2.4.2.

Глава 3

Тестирање софтвера и развој вођен тестовима

Ово поглавље је у припреми!

3.1 Писање јединичних тестова

Задатак 3.1.1. Дата је наредна функција:

```
#include <vector>

using namespace std;

int prebrajanje(const vector<int> &xs)
{
    if (xs.empty()) return -1;

    int rezultat = 0;
    int tmp = xs[0];

    for (auto x: xs) {
        if (x < tmp) {
            rezultat++;
        }
        tmp = x;
    }

    return rezultat;
}
```

Користећи библиотеку „Catch2” написати:

- Барем 4 јединична тест случаја који тестирају специјалне случајеве функције.
- Барем 2 јединична тест случаја који тестирају уобичајене случајеве функције.

Решење 3.1.1.

Задатак 3.1.2. Дата је наредна функција:

```
#include <vector>
#include <map>
#include <string>
#include <iterator>
#include <stdexcept>

using namespace std;

string json(const vector<map<string, string>> &objekti)
{
    if (objekti.empty())
    {
        throw invalid_argument("objekti");
    }

    string rezultat = "[";

    for (auto i = 0u; i < objekti.size(); ++i)
    {
        if (objekti[i].empty())
        {
            rezultat += "{},";
            continue;
        }

        rezultat += '{';
        for (auto iter = cbegin(objekti[i]); iter != cend(objekti[i]);
            ↪  ++iter)
        {
            rezultat += '"';
            rezultat += iter->first;
            rezultat += '"';
            rezultat += ':';
            rezultat += '"';
            rezultat += iter->second;
            rezultat += '"';
            rezultat += ',';
        }
        rezultat.back() = '}';
        rezultat += ',';
    }

    rezultat.back() = ']';

    return rezultat;
}
```

Користећи библиотеку „Catch2” написати:

- Барем 4 јединична тест случаја који тестирају специјалне случајеве функције.
- Барем 2 јединична тест случаја који тестирају уобичајене случајеве функције.

Решење 3.1.2.

Задатак 3.1.3. Дата је наредна функција:

```
#include <vector>
#include <cmath>

using namespace std;

using tacka = pair<double, double>;
using linija = pair<tacka, tacka>;

vector<linija> izdvojLinije(const vector<linija> &linije, const double
    ↪  najmanjaDuzina)
{
    vector<line> idzvojene;

    for (auto i = 0u; i < linije.size(); ++i)
    {
        const auto l = linije[i];
        const auto duzina = sqrt(pow(l.first.first - l.second.first,
            ↪ 2) + pow(l.first.second - l.second.second, 2));

        if (duzina >= najmanjaDuzina)
        {
            idzvojene.push_back(l);
        }
    }

    return idzvojene;
}
```

Користећи библиотеку „Catch2” написати:

- Барем 4 јединична тест случаја који тестирају специјалне случајеве функције.
- Барем 2 јединична тест случаја који тестирају уобичајене случајеве функције.

Решење 3.1.3.

Задатак 3.1.4. Дата је наредна функција:

```
using namespace std;

struct racer_data
{
    racer_data(double speed, double time)
        : speed(speed)
        , time(time)
    {}

    double speed; // m/s
    double time;  // s

    bool operator== (const racer_data & other) const
    {
        return speed == other.speed && time == other.time;
    }
};

void split_racer_data(vector<racer_data> &data, const double
    ↪ border_road)
```

```
{
    vector<racer_data> tmp;

    for (auto i = 0u; i < data.size(); ++i)
    {
        const double racer_road = data[i].speed * data[i].time;
        if (racer_road < border_road)
        {
            tmp.push_back(data[i]);
        }
    }
    for (auto i = 0u; i < data.size(); ++i)
    {
        const double racer_road = data[i].speed * data[i].time;
        if (racer_road >= border_road)
        {
            tmp.push_back(data[i]);
        }
    }

    data = tmp;
}
```

Користећи библиотеку „Catch2” написати:

- Барем 4 јединична тест случаја који тестирају специјалне случајеве функције.
- Барем 2 јединична тест случаја који тестирају уобичајене случајеве функције.

Решење 3.1.4.

Задатак 3.1.5. Дата је наредна функција:

```
using namespace std;

struct point
{
    int x;
    int y;

    bool operator<(const point& other) const
    {
        return x < other.x || (x == other.x && y < other.y);
    }

    bool operator==(const point& other) const
    {
        return x == other.x && y == other.y;
    }
};

void arrange_data(vector<point> &data, const int axis_limit)
{
    vector<point> tmp;

    for (auto i = 0u; i < data.size(); ++i)
    {
        if (data[i].x < axis_limit && data[i].y < axis_limit)
```

```
        {
            tmp.push_back(data[i]);
        }
    }
    for (auto i = 0u; i < tmp.size(); ++i)
    {
        for (auto j = 0u; j < tmp.size(); ++j)
        {
            if (tmp[i] < tmp[j])
            {
                auto t = tmp[i];
                tmp[i] = tmp[j];
                tmp[j] = t;
            }
        }
    }
    data = tmp;
}
```

Користећи библиотеку „Catch2” написати:

- Барем 4 јединична тест случаја који тестирају специјалне случајеве функције.
- Барем 2 јединична тест случаја који тестирају уобичајене случајеве функције.

Решење 3.1.5.

3.2 Развој вођен тестовима

Задатак 3.2.1. Коришћењем технике развоја вођеног тестовима, имплементирати класу *ФилтерСтудената* која из колекције објеката класе *Студент* издваја оне које задовољавају одређени критеријум. Објекти класе *Студент* представљају студенте који имају име, презиме и просечну оцену. Издвајање се врши према просечној оцени. Обезбедити наредни јавни интерфејс класе *ФилтерСтудената*:

- Конструктор класе прихвата наредна два аргумента који дефинишу критеријум издвајања:
  1. Карактер који представља начин на који се врши издвајање и може имати вредности l, g или e, којима се издвајају студенти који имају просек који је мањи, већи или једнак вредности другог аргумента, редом. Све остале карактере сматрати грешком.
  2. Вредност у покретном зарезу која ће се користити у упоређивању са просечном оценом.
- Написати оператор позива функцијског објекта који прихвата колекцију објеката класе *Студент* и враћа колекцију која садржи имена и презимена само оних студената који задовољавају критеријум који је дефинисан прослеђеним вредностима приликом конструкције.

Решење 3.2.1.

Задатак 3.2.2. Коришћењем технике развоја вођеног тестовима, имплементирати класу *ЛокаторСупермаркета* која израчунава удаљеност супермаркета од породица у неком граду. Објекти класе *Породица* представљају локације породица у дводимензионалној равни и имају x-координату и y-координату, док се супермаркет налази у координатном почетку. Обезбедити наредни јавни интерфејс класе *ЛокаторСупермаркета*:

- Конструктор класе прихвата x-координату и y-координату која представља координате супермаркета.
- Метод *додајПородицу* служи за чување информације о локацији једне породице. Обезбедити да се не чувају дупликати (тј. да на једној координати живи највише једна породица). Претпоставити да породице неће живети на истој локацији као и супермаркет.
- Написати оператор позива функцијског објекта који на основу аргумента који представља редни број квадранта, израчунава и враћа суму растојања до супермаркета од оних тачака из запамћене колекције објеката класе *Породица* који се налазе у квадранту који је прослеђен као аргумент оператора. Прослеђивање квадранта који није 1, 2, 3 или 4 сматрати за грешку.

Решење 3.2.2.

Задатак 3.2.3. Коришћењем технике развоја вођеног тестовима, имплементирати класу *НГрамКалкулатор* која израчунава колекцију n-грама за дати текст. Један n-грам представља подниску дужине n за дати текст. Потребно је пронаћи све n-граме за дати текст као и број појављивања тих n-грама у датом тексту. Користити инстанце класе *НГрам* као репрезентацију једног n-грама. Објекти ове класе имају стринг који представља сам n-грам као и број појављивања тог n-грама у тексту. Обезбедити наредни јавни интерфејс класе *НГрамКалкулатор*:

- Конструктор који прихвата стринг који представља текст на основу којег се рачунају n-грами.
- Написати оператор позива функцијског објекта који прихвата неозначени цео број n који представља дужину n-грама које треба израчунати. Повратна вредност метода је колекција објеката класе *НГрам*.

Пример: Триграми текста $abbbabbaabba$ и њихова појављивања су дати секвенцом $(abb, 3), (bbb, 1), (bba, 3), (bab, 1), (baa, 1), (aab, 1)$.

Решење 3.2.3.

Задатак 3.2.4. Коришћењем технике развоја вођеног тестовима, имплементирати класу *ТрговачкиПутник* која израчунава најбржи пут за обилазак градова грамзивом претрагом. Објекти класе *Град* представљају локације градова у дводимензионалној равни и имају x-координату, y-координату и индикатор да ли је град посећен током претраге. Обезбедити наредни јавни интерфејс класе *ТрговачкиПутник*:

- Метод *додајГрад* служи за додавање информације о локацији једног града. Претпоставити да неће постојати дупликати.

- Метод *пронађиНаредниНајближиГрад* проналази индекс једног од запамћених градова који није до сада посећен у претрази, при чему важи да је пронађен град најближи (у смислу Еуклидског растојања) последњем посећеном граду у претрази. Претрага почиње од координатног почетка.

- Написати оператор позива функцијског објекта који извршава обилазак запамћених градова. Претрага почиње од координатног почетка, а затим посећује један по један град. Редослед обилажења диктирају индекси који се добијају методом *пронађиНаредниНајближиГрад*, описаним изнад. Оператор враћа укупан пређени пут током претраге, при чему се пређени пут рачуна помоћу Менхетн растојања.

Решење 3.2.4.

Задатак 3.2.5. Коришћењем технике развоја вођеног тестовима, имплементирати класу *Лавиринт* која омогућава израчунавање успеха пролазака кроз лавиринт. Обезбедити наредни јавни интерфејс класе *Лавиринт*:

- Конструктор који на основу прослеђеног стринга иницијализује један пут кроз лавиринт. Пут кроз лавиринт се сматра валидним само ако испуњава наредне услове:

  – Пут не сме бити празан.
  – Пут се мора састојати само од карактера w, a, s, d и $.$ (тачка). Карактери w, a, s и d представљају један корак у смеру напред, лево, назад и десно, редом. Карактер тачке представља одморište.
  – Пут се мора завршавати тачком.

  Уколико пут не успуњава неки од наведених услова, пријавити одговарајућу грешку.

- Метод *дужинаПута* служи за израчунавање информације о дужини пута (број корака који је потребно остварити на путу од почетка до краја лавиринта, без одморишта).

- Написати оператор позива функцијског објекта који на основу аргумента типа стринг, који представља пут неке особе кроз лавиринт (претпоставити да ће бити прослеђен пут који се састоји од карактера w, a, s и d), израчунава награду коју је та особа добила. Награда се израчунава тако што се посматра да ли је особа прешла валидан пут између свака два суседна одморишта. Уколико јесте, онда се тој особи додељује број бодова који је једнак дужини пута између текућа два одморишта и даље се посматра пут до наредног одморишта. Уколико је особа направила погрешан корак у било ком тренутку између два одморишта, онда зауставити њено даље кретање кроз лавиринт и пријавити награду до претходног одморишта.

Пример: Нека је дат наредни исправан пут кроз лавиринт

$$wwdsdw.waaw.wddssddww.aawddd.$$

- Особа која прелази пут waw добија 0 поена (није стигла до 1. одморишта).

- Особа која прелази пут $wwdsdwwaawwddaaaaaaaaaaaa$ добија $6 + 4 = 10$ поена (стигла је до 2. одморишта, али је погрешила на путу до 3. одморишта).
- Особа која прелази пут $wwdsdwwaawwddssddwwaawddd$ добија $6 + 4 + 9 + 6 = 25$ поена (стигла је по последњег одморишта без грешке).

Графички приказ прелажења ових путева дат је у наредној табели:

w	w	d	s	d	w	.	w	a	a	w	.	w	d	d	s	s	d	d	w	w	.	a	a	w	d	d	d	.
w	a	w				×																						0
w	w	d	s	d	w	✓	w	a	a	w	✓	w	d	d	a	a	a	a	a	a	×	a	a	a	a	a	a	10
w	w	d	s	d	w	✓	w	a	a	w	✓	w	d	d	s	s	d	d	w	w	✓	a	a	w	d	d	d	25

Решење 3.2.5.

3.3 Решења задатака

Писање јединичних тестова

Решење 3.1.1.

Задатак 3.1.1.

Решење 3.1.2.

Задатак 3.1.2.

Решење 3.1.3.

Задатак 3.1.3.

Решење 3.1.4.

Задатак 3.1.4.

Решење 3.1.5.

Задатак 3.1.5.

Развој вођен тестовима

Решење 3.2.1. Развој започињемо имплементацијом класе *Студент* која се развија праволинијски, с обзиром да је у питању класа-податак. Тестове коректности ове класе остављамо читаоцима за вежбу, с обзиром да је акценат овог задатка на класи *ФилтерСтудената*, коју развијамо у наставку решења.

```
#pragma once

#include <string>

using namespace std;

class Student
{
public:
    Student(string ime, string prezime, double prosecnaOcena)
        : _ime(ime)
```

```
        , _prezime(prezime)
        , _prosecnaOcena(prosecnaOcena)
    {}

    inline string imePrezime() const
    {
        return _ime + " " + _prezime;
    }

    inline double prosecnaOcena() const
    {
        return _prosecnaOcena;
    }

private:
    string _ime;
    string _prezime;
    double _prosecnaOcena;
};
```

Пређимо сада на развој конструктора класе *ФилтерСтудената*. Пре него ли се конструише објекат класе, потребно је написати тест који проверава коректност прослеђених аргумената. Уколико корисник за први аргумент проследи карактер који није l, g или e, очекујемо да буде испаљен изузетак `std::invalid_argument`.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include "Student.hpp"
#include "FilterStudenata.hpp"

TEST_CASE("FilterStudenata", "[class]")
{
    SECTION("Konstruktor prijavljuje invalid_argument izuzetak ako se
        ↪ prosledi karakter koji nije l, g ili e")
    {
        // Arrange
        const auto ulaz = 'x';

        // Act + Assert
        REQUIRE_THROWS_AS(FilterStudenata(ulaz, 0.0), invalid_argument
            ↪ );
    }
}
```

Тест не пролази компилацију, с обзиром да класа није дефинисана, те га сматрамо као да је пао. Дефинишимо класу и одговарајући конструктор и имплементирајмо проверу првог аргумента.

```
#pragma once

#include <stdexcept>
#include "Student.hpp"

using namespace std;

class FilterStudenata
{
```

```
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(uslov)
        , _vrednost(vrednost)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
    }

private:
    char _uslov;
    double _vrednost;
};
```

Тест сада пролази, па је неопходно извршити рефакторисање кода. Издвајамо проверу у помоћни приватни метод који врши проверу пре него што се додели вредност одговарајућој чланици класе.

```
#pragma once

#include <stdexcept>
#include "Student.hpp"

using namespace std;

class FilterStudenata
{
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(proveriUslov(uslov))
        , _vrednost(vrednost)
    {
    }

private:
    char proveriUslov(char uslov)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
        return uslov;
    }

    char _uslov;
    double _vrednost;
};
```

Грешка коју бисмо сада направили јесте да напишемо тест који проверава да ли се објекат класе успешно конструише уколико се проследи ваљан карактер. Проблем који бисмо добили писањем овог теста јесте да ће он сигурно проћи без иједне написане линије кода (тзв. *зелени тест*), што се сматра (потенцијалним) промашајем у техници развоја вођеним тестовима. Наиме, тестирање ове функционалности се крије у претходно написаном тесту, те је овај тест сувишан[1].

[1]Напоменимо да ово није увек случај, већ да зелени тестови могу индиковати и неке другачије

Нови тест који пишемо проверава да ли оператор позива функцијског објекта може да се позове са тривијалним улазима. У нашем случају, то је празан вектор објеката. Оно што очекујемо као одговор јесте празан вектор стрингова.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include "Student.hpp"
#include "FilterStudenata.hpp"

TEST_CASE("FilterStudenata", "[class]")
{
    SECTION("Konstruktor prijavljuje invalid_argument izuzetak ako se
       ↪ prosledi karakter koji nije l, g ili e")
    {
        // Arrange
        const auto ulaz = 'x';

        // Act + Assert
        REQUIRE_THROWS_AS(FilterStudenata(ulaz, 0.0), invalid_argument
           ↪ );
    }

    SECTION("operator() vraca prazan vektor stringova ako se prosledi
       ↪ prazan vektor studenata")
    {
        // Arrange
        const FilterStudenata filter('e', 0.0);
        const vector<Student> ulaz;
        const vector<string> ocekivanIzlaz;

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }
}
```

Тест не пролази, с обзиром да оператор није дефинисан. Дефинишимо га и имплементирајмо га тако да покријемо само написани тест и ништа више од тога[2].

```
#pragma once

#include <stdexcept>
#include <string>
#include "Student.hpp"

using namespace std;

class FilterStudenata
```

проблеме. На пример, појава зеленог теста може значити да функционалност коју тестирамо тим тестом смо већ имплементирати, а можда није било потребно. Другим речима, имплементацијом неке од претходних функционалности смо имплементирали (макар) још једну функционалност. Ово се коси са приступом „један тест – једна функционалност", те је потребно вратити се уназад и проверити смисао написаних тестова и одговарајућих имплементација.

[2]Све више од овога што смо имплементирали води ка зеленим тестовима и проблемима из претходне фусноте.

```
{
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(proveriUslov(uslov))
        , _vrednost(vrednost)
    {
    }

    vector<string> operator()(const vector<Student> &studenti) const
    {
        return {};
    }

private:
    char proveriUslov(char uslov)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
        return uslov;
    }

    char _uslov;
    double _vrednost;
};
```

Сви тестови сада пролазе, па је неопходно извршити рефакторисање кода. С обзиром да је имплементација била тривијална, одлучујемо да нема потребе за рефакторисањем.

Сада желимо да развијемо неопходно филтрирање података. Прво ћемо се фокусирати на провере по једнакости. За почетак, напишимо тест за тестирање случаја када се проследи један објекат који задовољава услов једнакости.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include "Student.hpp"
#include "FilterStudenata.hpp"

TEST_CASE("FilterStudenata", "[class]")
{
    SECTION("Konstruktor prijavljuje invalid_argument izuzetak ako se
       ↪ prosledi karakter koji nije l, g ili e")
    {
        // Arrange
        const auto ulaz = 'x';

        // Act + Assert
        REQUIRE_THROWS_AS(FilterStudenata(ulaz, 0.0), invalid_argument
           ↪ );
    }

    SECTION("operator() vraca prazan vektor stringova ako se prosledi
       ↪ prazan vektor studenata")
    {
        // Arrange
```

```
        const FilterStudenata filter('e', 0.0);
        const vector<Student> ulaz;
        const vector<string> ocekivanIzlaz;

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor sa jednim imenom i prezimenom ako
        ↪ se prosledi vektor sa jednim studentom koji ispunjava uslov
        ↪ jednakosti")
    {
        // Arrange
        const FilterStudenata filter('e', 10.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0) };
        const vector<string> ocekivanIzlaz{ "Pera Peric" };

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }
}
```

Очекивано, последњи тест пада с обзиром да оператор враћа празан вектор у свим случајевима. Стога, неопходно је додати проверу да ли је вектор празан. Уколико има макар један податак, онда желимо да имплементирамо исправну проверу за први податак у вектору.

```
#pragma once

#include <stdexcept>
#include <string>
#include "Student.hpp"

using namespace std;

class FilterStudenata
{
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(proveriUslov(uslov))
        , _vrednost(vrednost)
    {
    }

    vector<string> operator()(const vector<Student> &studenti) const
    {
        if (studenti.empty())
        {
            return {};
        }
```

```
        vector<string> filtrirani;

        const auto &student = studenti[0];
        if (_uslov == 'e')
        {
            if (student.prosecnaOcena() == _vrednost)
            {
                const auto imePrezime = student.imePrezime();
                filtrirani.push_back(imePrezime);
            }
        }

        return filtrirani;
    }

private:
    char proveriUslov(char uslov)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
        return uslov;
    }

    char _uslov;
    double _vrednost;
};
```

Сви тестови пролазе, међутим, оператор позива функцијског објекта постаје загушен, те ћемо издвојити филтрирање по једнакости у помоћни метод класе.

```
#pragma once

#include <stdexcept>
#include <string>
#include "Student.hpp"

using namespace std;

class FilterStudenata
{
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(proveriUslov(uslov))
        , _vrednost(vrednost)
    {
    }

    vector<string> operator()(const vector<Student> &studenti) const
    {
        if (studenti.empty())
        {
            return {};
        }

        vector<string> filtrirani;
```

```
        const auto &student = studenti[0];
        filtrirajStudentaPoJednakosti(student, filtrirani);

        return filtrirani;
    }

private:
    char proveriUslov(char uslov)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
        return uslov;
    }

    void filtrirajStudentaPoJednakosti(const Student &student, vector<
        ↪ string> &filtrirani) const
    {
        if (_uslov == 'e')
        {
            if (student.prosecnaOcena() == _vrednost)
            {
                const auto imePrezime = student.imePrezime();
                filtrirani.push_back(imePrezime);
            }
        }
    }

    char _uslov;
    double _vrednost;
};
```

Како провера за једног студента ради коректно, прелазимо на тестирање случаја када имамо произвољан број студената од којих неки задовољавају услов, а неки не. У овом тесту додајемо `CHECK` проверу да ли дужина резултујућег вектора одговара исправној дужини пре него што проверимо садржај вектора макроом `REQUIRE`. Иако није било неопходно, овај тест показује како можемо комбиновати више макроа у тестирању различитих аспеката једног теста.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include "Student.hpp"
#include "FilterStudenata.hpp"

TEST_CASE("FilterStudenata", "[class]")
{
    SECTION("Konstruktor prijavljuje invalid_argument izuzetak ako se
        ↪ prosledi karakter koji nije l, g ili e")
    {
        // Arrange
        const auto ulaz = 'x';

        // Act + Assert
        REQUIRE_THROWS_AS(FilterStudenata(ulaz, 0.0), invalid_argument
            ↪ );
```

```
    }

    SECTION("operator() vraca prazan vektor stringova ako se prosledi
       ↪ prazan vektor studenata")
    {
        // Arrange
        const FilterStudenata filter('e', 0.0);
        const vector<Student> ulaz;
        const vector<string> ocekivanIzlaz;

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor sa jednim imenom i prezimenom ako
       ↪  se prosledi vektor sa jednim studentom koji ispunjava uslov
       ↪  jednakosti")
    {
        // Arrange
        const FilterStudenata filter('e', 10.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0) };
        const vector<string> ocekivanIzlaz{ "Pera Peric" };

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor imena i prezimena samo onih
       ↪ studenata koji ispunjavaju uslov jednakosti ako se prosledi
       ↪ vektor sa vise studenata")
    {
        // Arrange
        const FilterStudenata filter('e', 10.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0),
           ↪ Student("Laza", "Lazic", 8.4), Student("Milica", "
           ↪ Jovanovic", 10.0), Student("Sonja", "Stojanovic", 10.0),
           ↪  Student("Ana", "Nikolic", 9.5) };
        const vector<string> ocekivanIzlaz{ "Pera Peric", "Milica
           ↪ Jovanovic", "Sonja Stojanovic" };

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        CHECK(dobijenIzlaz.size() == ocekivanIzlaz.size());
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }
}
```

С обзиром на разумно рефакторисан код који имамо у овом тренутку, имплементација тече праволинијски додавањем петље која тестира студенте из просле-

ђеног вектора редом уместо провере само првог студента.

```
#pragma once

#include <stdexcept>
#include <string>
#include "Student.hpp"

using namespace std;

class FilterStudenata
{
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(proveriUslov(uslov))
        , _vrednost(vrednost)
    {
    }

    vector<string> operator()(const vector<Student> &studenti) const
    {
        if (studenti.empty())
        {
            return {};
        }

        vector<string> filtrirani;

        for (const auto &student : studenti)
        {
            filtrirajStudentaPoJednakosti(student, filtrirani);
        }

        return filtrirani;
    }

private:
    char proveriUslov(char uslov)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
        return uslov;
    }

    void filtrirajStudentaPoJednakosti(const Student &student, vector<
       ↪ string> &filtrirani) const
    {
        if (_uslov == 'e')
        {
            if (student.prosecnaOcena() == _vrednost)
            {
                const auto imePrezime = student.imePrezime();
                filtrirani.push_back(imePrezime);
            }
        }
```

```
    }

    char _uslov;
    double _vrednost;
};
```

Поново, одлучујемо да смо задовољни текућим дизајном и настављамо без рефакторисања.

Наредним тест прелазимо на развој поређења по релацији „мање од”. Тест који наводимо је готово идентичан као претходни, са разликом у карактеру који се прослеђује конструктору и очекиваном излазу.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include "Student.hpp"
#include "FilterStudenata.hpp"

TEST_CASE("FilterStudenata", "[class]")
{
    SECTION("Konstruktor prijavljuje invalid_argument izuzetak ako se
       ↪ prosledi karakter koji nije l, g ili e")
    {
        // Arrange
        const auto ulaz = 'x';

        // Act + Assert
        REQUIRE_THROWS_AS(FilterStudenata(ulaz, 0.0), invalid_argument
           ↪ );
    }

    SECTION("operator() vraca prazan vektor stringova ako se prosledi
       ↪ prazan vektor studenata")
    {
        // Arrange
        const FilterStudenata filter('e', 0.0);
        const vector<Student> ulaz;
        const vector<string> ocekivanIzlaz;

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor sa jednim imenom i prezimenom ako
       ↪  se prosledi vektor sa jednim studentom koji ispunjava uslov
       ↪  jednakosti")
    {
        // Arrange
        const FilterStudenata filter('e', 10.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0) };
        const vector<string> ocekivanIzlaz{ "Pera Peric" };

        // Act
        const auto dobijenIzlaz = filter(ulaz);
```

```
        // Assert
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor imena i prezimena samo onih
        ↪ studenata koji ispunjavaju uslov jednakosti ako se prosledi
        ↪ vektor sa vise studenata")
    {
        // Arrange
        const FilterStudenata filter('e', 10.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0),
            ↪ Student("Laza", "Lazic", 8.4), Student("Milica", "
            ↪ Jovanovic", 10.0), Student("Sonja", "Stojanovic", 10.0),
            ↪  Student("Ana", "Nikolic", 9.5) };
        const vector<string> ocekivanIzlaz{ "Pera Peric", "Milica
            ↪ Jovanovic", "Sonja Stojanovic" };

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        CHECK(dobijenIzlaz.size() == ocekivanIzlaz.size());
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor imena i prezimena samo onih
        ↪ studenata koji ispunjavaju uslov \"manje od\" ako se
        ↪ prosledi vektor sa vise studenata")
    {
        // Arrange
        const FilterStudenata filter('l', 10.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0),
            ↪ Student("Laza", "Lazic", 8.4), Student("Milica", "
            ↪ Jovanovic", 10.0), Student("Sonja", "Stojanovic", 10.0),
            ↪  Student("Ana", "Nikolic", 9.5) };
        const vector<string> ocekivanIzlaz{ "Laza Lazic", "Ana Nikolic
            ↪ " };

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        CHECK(dobijenIzlaz.size() == ocekivanIzlaz.size());
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }
}
```

Оператор сада не може директно да се ослони на позив метода за поређење по једнакости, већ је неопходно да изврши селекцију неког од помоћних метода за поређење. Због тога, имплементирамо додатни помоћни метод који врши селекцију по типу релације и помоћни метод за филтрирање по новој релацији.

```
#pragma once

#include <stdexcept>
#include <string>
```

```
#include "Student.hpp"

using namespace std;

class FilterStudenata
{
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(proveriUslov(uslov))
        , _vrednost(vrednost)
    {
    }

    vector<string> operator()(const vector<Student> &studenti) const
    {
        if (studenti.empty())
        {
            return {};
        }

        vector<string> filtrirani;

        for (const auto &student : studenti)
        {
            filtrirajStudenta(student, filtrirani);
        }

        return filtrirani;
    }

private:
    char proveriUslov(char uslov)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
        return uslov;
    }

    void filtrirajStudenta(const Student &student, vector<string> &
        ↪ filtrirani) const
    {
        if (_uslov == 'e')
        {
            filtrirajStudentaPoJednakosti(student, filtrirani);
        }
        else if (_uslov == 'l')
        {
            filtrirajStudentaPoManjeOd(student, filtrirani);
        }
    }

    void filtrirajStudentaPoJednakosti(const Student &student, vector<
        ↪ string> &filtrirani) const
    {
```

```
        if (student.prosecnaOcena() == _vrednost)
        {
            const auto imePrezime = student.imePrezime();
            filtrirani.push_back(imePrezime);
        }
    }

    void filtrirajStudentaPoManjeOd(const Student &student, vector<
        ↪ string> &filtrirani) const
    {
        if (student.prosecnaOcena() < _vrednost)
        {
            const auto imePrezime = student.imePrezime();
            filtrirani.push_back(imePrezime);
        }
    }

    char _uslov;
    double _vrednost;
};
```

Приметимо да методи *филтрирајСтудентаПоЈеднакости* и *филтрирајСтудентаПоМањеОд* имају идентичан код у телу услова. Дуплицирање кода желимо да избегнемо, те рефакторишемо код тако да ови методи врше само проверу услова релација. С обзиром да смо применили значење метода, мењамо и њихов назив како би исправно описао оно што ови методи израчунавају. Филтрирање издвајамо у метод *филтрирајСтудента* и тиме решавамо све проблеме са дизајном[3].

```
#pragma once

#include <stdexcept>
#include <string>
#include "Student.hpp"

using namespace std;

class FilterStudenata
{
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(proveriUslov(uslov))
        , _vrednost(vrednost)
    {
    }

    vector<string> operator()(const vector<Student> &studenti) const
    {
        if (studenti.empty())
        {
            return {};
        }

        vector<string> filtrirani;

        for (const auto &student : studenti)
```

[3]Пажљиви читалац ће приметити још један проблем, али њега остављамо за наредни тест.

```
        {
            filtrirajStudenta(student, filtrirani);
        }

        return filtrirani;
    }

private:
    char proveriUslov(char uslov)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
        return uslov;
    }

    void filtrirajStudenta(const Student &student, vector<string> &
       ↪ filtrirani) const
    {
        auto ispunjavaUslov = false;
        if (_uslov == 'e')
        {
            ispunjavaUslov = ispunjavaUslovJednakosti(student);
        }
        else if (_uslov == 'l')
        {
            ispunjavaUslov = ispunjavaUslovManjeOd(student);
        }

        if (!ispunjavaUslov)
        {
            return;
        }

        const auto imePrezime = student.imePrezime();
        filtrirani.push_back(imePrezime);
    }

    bool ispunjavaUslovJednakosti(const Student &student) const
    {
        return student.prosecnaOcena() == _vrednost;
    }

    bool ispunjavaUslovManjeOd(const Student &student) const
    {
        return student.prosecnaOcena() < _vrednost;
    }

    char _uslov;
    double _vrednost;
};
```

Последњи тест који пишемо тестира филтрирање по релацији „веће од”.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"
```

```
#include "Student.hpp"
#include "FilterStudenata.hpp"

TEST_CASE("FilterStudenata", "[class]")
{
    SECTION("Konstruktor prijavljuje invalid_argument izuzetak ako se
        ↪ prosledi karakter koji nije l, g ili e")
    {
        // Arrange
        const auto ulaz = 'x';

        // Act + Assert
        REQUIRE_THROWS_AS(FilterStudenata(ulaz, 0.0), invalid_argument
            ↪ );
    }

    SECTION("operator() vraca prazan vektor stringova ako se prosledi
        ↪ prazan vektor studenata")
    {
        // Arrange
        const FilterStudenata filter('e', 0.0);
        const vector<Student> ulaz;
        const vector<string> ocekivanIzlaz;

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor sa jednim imenom i prezimenom ako
        ↪  se prosledi vektor sa jednim studentom koji ispunjava uslov
        ↪  jednakosti")
    {
        // Arrange
        const FilterStudenata filter('e', 10.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0) };
        const vector<string> ocekivanIzlaz{ "Pera Peric" };

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor imena i prezimena samo onih
        ↪ studenata koji ispunjavaju uslov jednakosti ako se prosledi
        ↪ vektor sa vise studenata")
    {
        // Arrange
        const FilterStudenata filter('e', 10.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0),
            ↪ Student("Laza", "Lazic", 8.4), Student("Milica", "
            ↪ Jovanovic", 10.0), Student("Sonja", "Stojanovic", 10.0),
```

```
            ↪  Student("Ana", "Nikolic", 9.5) };
        const vector<string> ocekivanIzlaz{ "Pera Peric", "Milica
            ↪ Jovanovic", "Sonja Stojanovic" };

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        CHECK(dobijenIzlaz.size() == ocekivanIzlaz.size());
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor imena i prezimena samo onih
        ↪ studenata koji ispunjavaju uslov \"manje od\" ako se
        ↪ prosledi vektor sa vise studenata")
    {
        // Arrange
        const FilterStudenata filter('l', 10.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0),
            ↪ Student("Laza", "Lazic", 8.4), Student("Milica", "
            ↪ Jovanovic", 10.0), Student("Sonja", "Stojanovic", 10.0),
            ↪  Student("Ana", "Nikolic", 9.5) };
        const vector<string> ocekivanIzlaz{ "Laza Lazic", "Ana Nikolic
            ↪ " };

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        CHECK(dobijenIzlaz.size() == ocekivanIzlaz.size());
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }

    SECTION("operator() vraca vektor imena i prezimena samo onih
        ↪ studenata koji ispunjavaju uslov \"vece od\" ako se prosledi
        ↪  vektor sa vise studenata")
    {
        // Arrange
        const FilterStudenata filter('g', 9.0);
        const vector<Student> ulaz{ Student("Pera", "Peric", 10.0),
            ↪ Student("Laza", "Lazic", 8.4), Student("Milica", "
            ↪ Jovanovic", 10.0), Student("Sonja", "Stojanovic", 10.0),
            ↪  Student("Ana", "Nikolic", 9.5) };
        const vector<string> ocekivanIzlaz{ "Pera Peric", "Milica
            ↪ Jovanovic", "Sonja Stojanovic", "Ana Nikolic" };

        // Act
        const auto dobijenIzlaz = filter(ulaz);

        // Assert
        CHECK(dobijenIzlaz.size() == ocekivanIzlaz.size());
        REQUIRE(dobijenIzlaz == ocekivanIzlaz);
    }
}
```

Имплементација је праволинијска, те само приказујемо код у наставку.

```
#pragma once
```

```
#include <stdexcept>
#include <string>
#include "Student.hpp"

using namespace std;

class FilterStudenata
{
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(proveriUslov(uslov))
        , _vrednost(vrednost)
    {
    }

    vector<string> operator()(const vector<Student> &studenti) const
    {
        if (studenti.empty())
        {
            return {};
        }

        vector<string> filtrirani;

        for (const auto &student : studenti)
        {
            filtrirajStudenta(student, filtrirani);
        }

        return filtrirani;
    }

private:
    char proveriUslov(char uslov)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
        return uslov;
    }

    void filtrirajStudenta(const Student &student, vector<string> &
        ↪ filtrirani) const
    {
        auto ispunjavaUslov = false;
        if (_uslov == 'e')
        {
            ispunjavaUslov = ispunjavaUslovJednakosti(student);
        }
        else if (_uslov == 'l')
        {
            ispunjavaUslov = ispunjavaUslovManjeOd(student);
        }
        else
```

```
        {
            ispunjavaUslov = ispunjavaUslovVeceOd(student);
        }

        if (!ispunjavaUslov)
        {
            return;
        }

        const auto imePrezime = student.imePrezime();
        filtrirani.push_back(imePrezime);
    }

    bool ispunjavaUslovJednakosti(const Student &student) const
    {
        return student.prosecnaOcena() == _vrednost;
    }

    bool ispunjavaUslovManjeOd(const Student &student) const
    {
        return student.prosecnaOcena() < _vrednost;
    }

    bool ispunjavaUslovVeceOd(const Student &student) const
    {
        return student.prosecnaOcena() > _vrednost;
    }

    char _uslov;
    double _vrednost;
};
```

Приметимо да метод који смо именовали *филтрирајСтудента* заправо врши две функционалности – проверу задовољења услова неке од релација и филтрирање студента. Идеално, свака функција треба да обухвата једну функционалност, те издвајамо код за проверу услова у нови помоћни метод и тиме завршавамо развој класе *ФилтерСтудената*.

```
#pragma once

#include <stdexcept>
#include <string>
#include "Student.hpp"

using namespace std;

class FilterStudenata
{
public:
    FilterStudenata(char uslov, double vrednost)
        : _uslov(proveriUslov(uslov))
        , _vrednost(vrednost)
    {
    }

    vector<string> operator()(const vector<Student> &studenti) const
    {
```

```
        if (studenti.empty())
        {
            return {};
        }

        vector<string> filtrirani;

        for (const auto &student : studenti)
        {
            filtrirajStudenta(student, filtrirani);
        }

        return filtrirani;
    }

private:
    char proveriUslov(char uslov)
    {
        if (uslov != 'l' && uslov != 'g' && uslov != 'e')
        {
            throw invalid_argument("uslov");
        }
        return uslov;
    }

    void filtrirajStudenta(const Student &student, vector<string> &
       ↪ filtrirani) const
    {
        if (!ispunjavaUslov(student))
        {
            return;
        }

        const auto imePrezime = student.imePrezime();
        filtrirani.push_back(imePrezime);
    }

    bool ispunjavaUslov(const Student &student) const
    {
        if (_uslov == 'e')
        {
            return ispunjavaUslovJednakosti(student);
        }
        else if (_uslov == 'l')
        {
            return ispunjavaUslovManjeOd(student);
        }
        else if (_uslov == 'g')
        {
            return ispunjavaUslovVeceOd(student);
        }
        return false;
    }

    bool ispunjavaUslovJednakosti(const Student &student) const
    {
```

```
        return student.prosecnaOcena() == _vrednost;
    }

    bool ispunjavaUslovManjeOd(const Student &student) const
    {
        return student.prosecnaOcena() < _vrednost;
    }

    bool ispunjavaUslovVeceOd(const Student &student) const
    {
        return student.prosecnaOcena() > _vrednost;
    }

    char _uslov;
    double _vrednost;
};
```

Задатак 3.2.1.

Решење 3.2.2. За разлику од претходног задатка, имплементацију помоћне класе *Породица* развијаћемо паралелно са класом *ЛокаторСупермаркета*. Наравно, и даље је акценат задатка на развоју класе *ЛокаторСупермаркета*, те нећемо писати јединичне тестове за класу *Породица*. Ипак, из решења које следи у наставку текста, биће јасно којим редоследом се ова класа развија, те писање тестова остављамо читаоцима за вежбу.

Започнимо развој конструктора класе *ЛокаторСупермаркета*. У питању је једноставан конструктор који очекује две насумичне вредности у покретном зарезу и мора да конструише објекат без испаљивања изузетка.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include "LokatorSupermarketa.hpp"

TEST_CASE("LokatorSupermarketa", "[class]")
{
    SECTION("Konstruktor LokatorSupermarketa uspesno konstruise
       ↪ nasumicni objekat")
    {
        // Arrange
        const auto x = 0.0;
        const auto y = 0.0;

        // Act + Assert
        REQUIRE_NOTHROW(LokatorSupermarketa(x, y));
    }
}
```

Класа није дефинисана, те тест пада. Дефинишимо класу и одговарајући конструктор.

```
#pragma once

class LokatorSupermarketa
{
public:
```

```
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

private:
    double _x;
    double _y;
};
```

Сада тест пролази, али нема потребе за рефакторисањем. Уместо тога, прелазимо на писање првог теста за метод *додајПородицу*, који проверава да ли је објекат који му се прослеђује додат у колекцију свих породица. За почетак, потребно је да имплементирамо класу *Породица*.

```
#pragma once

class Porodica
{
public:
    Porodica(double x, double y)
        : _x(x), _y(y)
    {
    }

private:
    double _x;
    double _y;
};
```

Сада можемо написати тест за додавање једне нове породице.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include "LokatorSupermarketa.hpp"

TEST_CASE("LokatorSupermarketa", "[class]")
{
    SECTION("Konstruktor LokatorSupermarketa uspesno konstruise
       ↪ nasumicni objekat")
    {
        // Arrange
        const auto x = 0.0;
        const auto y = 0.0;

        // Act + Assert
        REQUIRE_NOTHROW(LokatorSupermarketa(x, y));
    }

    SECTION("Metod dodajPorodicu uspesno pamti jednu novu porodicu")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto ocekivanRezultat = 1;

        // Act
```

```
        const auto dobijenRezultat = lokator.dodajPorodicu(p1);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }
}
```

Приметимо да, према написаном тесту, очекујемо да метод *додајПородицу* враћа број породица које су запамћене. Ово није наведено у захтеву задатка, међутим, неопходно је да на неки начин израчунамо ову информацију како бисмо тестирали имплементацију. Један алтернативни приступ би био додавање новог јавног метода који ово израчунава. Иако бисмо на тај начин нарушили описани јавни интерфејс, могуће је да клијенти и развијачи нису могли да предвиде ову функционалност у тренутку писања захтева, те се и овај приступ може сматрати разумним избором. Дакле, одговарајуће решење за овакве ситуације се добија дискусијом са клијентима и развијачима и проналажењем решења које најбоље одговара контексту у оквиру којег се класа развија. Ми ћемо се држати приступа који стоји у тесту.

```
#pragma once

#include <vector>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.push_back(p);
        return _porodice.size();
    }

private:
    double _x;
    double _y;
    vector<Porodica> _porodice;
};
```

Имплементација је била довољно једноставна и ограничена да нам рефакторисање поново није неопходно. Прелазимо на писање наредног теста који провера случај дупликата.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include "LokatorSupermarketa.hpp"

TEST_CASE("LokatorSupermarketa", "[class]")
{
```

```
    SECTION("Konstruktor LokatorSupermarketa uspesno konstruise
        ↪ nasumicni objekat")
    {
        // Arrange
        const auto x = 0.0;
        const auto y = 0.0;

        // Act + Assert
        REQUIRE_NOTHROW(LokatorSupermarketa(x, y));
    }

    SECTION("Metod dodajPorodicu uspesno pamti jednu novu porodicu")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto ocekivanRezultat = 1;

        // Act
        const auto dobijenRezultat = lokator.dodajPorodicu(p1);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Metod dodajPorodicu ne pamti duplikate porodica")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto p2 = p1;
        const auto ocekivanRezultat = 1;

        // Act
        lokator.dodajPorodicu(p1);
        const auto dobijenRezultat = lokator.dodajPorodicu(p2);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }
}
```

Како бисмо проверили постојање дупликата, потребно је да тестирамо једнакост прослеђеног објекта са свим запамћеним објектима. Додатно, у случају да је колекција породица била празна, потребно је додати прослеђени објекат. Тако долазимо до наредне имплементације.

```
#pragma once

#include <vector>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
```

```
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        if (_porodice.empty())
        {
            _porodice.push_back(p);
        }

        for (const auto &porodica : _porodice)
        {
            if (!(porodica == p))
            {
                _porodice.push_back(p);
            }
        }
        return _porodice.size();
    }

private:
    double _x;
    double _y;
    vector<Porodica> _porodice;
};
```

Како би ова имплементација била коректна, потребно је додати оператор поређења по једнакости у класи *Породица*.

```
#pragma once

class Porodica
{
public:
    Porodica(double x, double y)
        : _x(x), _y(y)
    {
    }

    bool operator==(const Porodica &p) const
    {
        return _x == p._x && _y == p._y;
    }

private:
    double _x;
    double _y;
};
```

Метод *додајПородицу* је значајно нарастао и примарни је кандидат за рефакторисање. Као и у претходном задатку, и овде бисмо могли раздвојити функционалности на помоћне методе класе *ЛокаторСупермаркета*. Међутим, рефакторисање не укључује нужно само измену кода, већ и измене у неким другим одлукама приликом дизајна имплементације. На пример, уместо да користимо структуру података `vector` и да ручно имплементирамо све неопходне провере, могли бисмо да

искористимо структуру података `set` која већ имплементира све неопходне провере.

```
#pragma once

#include <set>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.insert(p);
        return _porodice.size();
    }

private:
    double _x;
    double _y;
    set<Porodica> _porodice;
};
```

Поново, како би нова имплементација била коректна, потребно је изменити класу *Породица*, овога пута додавањем оператора поређења по релацији „мање од”. С обзиром да тачке у дводимензионалној равни немају природно уређење по овој релацији, одлучили смо се за наредну имплементацију овог оператора због ефикасности израчунавања.

```
#pragma once

class Porodica
{
public:
    Porodica(double x, double y)
        : _x(x), _y(y)
    {
    }

    bool operator==(const Porodica &p) const
    {
        return _x == p._x && _y == p._y;
    }

    bool operator<(const Porodica &p) const
    {
        return _x < p._x && _y < p._y;
    }

private:
    double _x;
```

```
    double _y;
};
```

Тестови пролазе, као и пре рефакторисања. На овакве (и сличне) одлуке у променама дизајна утичу случајеви употребе у којима се класа која се развија користи. У случају измена структура података, потребно је анализирати све операције и размислити о структури података која ће најефикасније решавати посао. У нашем случају је свеједно за коју структуру података ћемо се одлучити, али желимо да прикажемо како се приликом развоја вођеног тестирањем и овакве одлуке једноставно могу увести у апликацију без бриге о увођењу нових дефеката, с обзиром на постојање тестова који ће открити нове дефекте (уколико до њих дође)[4].

Пређимо на развој оператора позива функцијском објекта. Прво ћемо тестирати случај уколико се проследи невалидан квадрант.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include <stdexcept>
#include "LokatorSupermarketa.hpp"

TEST_CASE("LokatorSupermarketa", "[class]")
{
    SECTION("Konstruktor LokatorSupermarketa uspesno konstruise
       ↪ nasumicni objekat")
    {
        // Arrange
        const auto x = 0.0;
        const auto y = 0.0;

        // Act + Assert
        REQUIRE_NOTHROW(LokatorSupermarketa(x, y));
    }

    SECTION("Metod dodajPorodicu uspesno pamti jednu novu porodicu")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto ocekivanRezultat = 1;

        // Act
        const auto dobijenRezultat = lokator.dodajPorodicu(p1);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Metod dodajPorodicu ne pamti duplikate porodica")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto p2 = p1;
        const auto ocekivanRezultat = 1;
```

[4]Наравно, успешност откривања дефеката тестовима јединица кода зависи искључиво од квалитета написаних тестова.

```
        // Act
        lokator.dodajPorodicu(p1);
        const auto dobijenRezultat = lokator.dodajPorodicu(p2);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Operator() prijavljuje izuzetak tipa invalid_argument
       ↪ ukoliko se prosledi kvadrant koji nije 1, 2, 3 ili 4")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);

        // Act + Assert
        REQUIRE_THROWS_AS(lokator(5), invalid_argument);
    }
}
```

Имплементација је сасвим праволинијска.

```
#pragma once

#include <set>
#include <stdexcept>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.insert(p);
        return _porodice.size();
    }

    double operator()(unsigned kvadrant) const
    {
        if (kvadrant != 1u && kvadrant != 2u && kvadrant != 3u &&
           ↪ kvadrant != 4u)
        {
            throw invalid_argument("kvadrant");
        }

        return -1.0;
    }

private:
    double _x;
    double _y;
```

```
    set<Porodica> _porodice;
};
```

Оно што нам „заудара” у датом коду јесте имплементација провере аргумента у самом оператору, те ћемо ту проверу издвојити у помоћни метод.

```
#pragma once

#include <set>
#include <stdexcept>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.insert(p);
        return _porodice.size();
    }

    double operator()(unsigned kvadrant) const
    {
        proveriKvadrant(kvadrant);

        return -1.0;
    }

private:
    void proveriKvadrant(unsigned kvadrant) const
    {
        if (kvadrant != 1u && kvadrant != 2u && kvadrant != 3u &&
            ↪ kvadrant != 4u)
        {
            throw invalid_argument("kvadrant");
        }
    }

    double _x;
    double _y;
    set<Porodica> _porodice;
};
```

Додајемо нови тест који се односи на случај када позивамо оператор позива функцијског објекта, а да пре тога нисмо додали ниједну породицу. У оваквим, специјалним случајевима, треба да одлучимо шта би била повратна вредност. С обзиром да очекујемо да оператор израчунава суму растојања, могли бисмо вратити било који негативан број у покретном зарезу, на пример, -1. Проблем са овим приступом јесте што вредност -1 није „специјалнија” од неке друге негативне вредности, па чак ни од нуле (с обзиром да не можемо имати породице чије локације се

поклапају са супермаркетом, на основу захтева задатка). Због тога, требало би избегавати овакве, „лажне” специјалне вредности. Уместо да оператор враћа чисту вредност у покретном зарезу, згодније би било да враћа *опциону* вредност у покретном зарезу, односно, објекат класе `optional`, за који можемо проверити да ли заиста садржи неку вредност или не.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include <stdexcept>
#include <optional>
#include "LokatorSupermarketa.hpp"

TEST_CASE("LokatorSupermarketa", "[class]")
{
    SECTION("Konstruktor LokatorSupermarketa uspesno konstruise
       ↪ nasumicni objekat")
    {
        // Arrange
        const auto x = 0.0;
        const auto y = 0.0;

        // Act + Assert
        REQUIRE_NOTHROW(LokatorSupermarketa(x, y));
    }

    SECTION("Metod dodajPorodicu uspesno pamti jednu novu porodicu")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto ocekivanRezultat = 1;

        // Act
        const auto dobijenRezultat = lokator.dodajPorodicu(p1);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Metod dodajPorodicu ne pamti duplikate porodica")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto p2 = p1;
        const auto ocekivanRezultat = 1;

        // Act
        lokator.dodajPorodicu(p1);
        const auto dobijenRezultat = lokator.dodajPorodicu(p2);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Operator() prijavljuje izuzetak tipa invalid_argument
```

```
        ↪ ukoliko se prosledi kvadrant koji nije 1, 2, 3 ili 4")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);

        // Act + Assert
        REQUIRE_THROWS_AS(lokator(5), invalid_argument);
    }

    SECTION("Operator() vraca prazan optional ukoliko lokator nema
        ↪ porodica")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const optional<double> ocekivanRezultat;

        // Act
        const auto dobijenRezultat = lokator(1);

        // Assert
        REQUIRE(dobijenRezultat.has_value() == ocekivanRezultat.
            ↪ has_value());
    }
}
```

Тест очекивано пада, с обзиром да враћамо опциону вредност која садржи број у покретном зарезу -1. Исправка је довољно једноставна да не захтева рефакторисање.

```
#pragma once

#include <set>
#include <stdexcept>
#include <optional>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.insert(p);
        return _porodice.size();
    }

    optional<double> operator()(unsigned kvadrant) const
    {
        proveriKvadrant(kvadrant);

        return {};
```

```
    }

private:
    void proveriKvadrant(unsigned kvadrant) const
    {
        if (kvadrant != 1u && kvadrant != 2u && kvadrant != 3u &&
            ↪ kvadrant != 4u)
        {
            throw invalid_argument("kvadrant");
        }
    }

    double _x;
    double _y;
    set<Porodica> _porodice;
};
```

Тестирајмо сада случај када локатор садржи информацију о једној породици и када израчунава суму растојања за квадрант у којем се та породица налази. Очекујемо да повратна вредност одговара растојању до те породице.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include <stdexcept>
#include <optional>
#include "LokatorSupermarketa.hpp"

TEST_CASE("LokatorSupermarketa", "[class]")
{
    SECTION("Konstruktor LokatorSupermarketa uspesno konstruise
        ↪ nasumicni objekat")
    {
        // Arrange
        const auto x = 0.0;
        const auto y = 0.0;

        // Act + Assert
        REQUIRE_NOTHROW(LokatorSupermarketa(x, y));
    }

    SECTION("Metod dodajPorodicu uspesno pamti jednu novu porodicu")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto ocekivanRezultat = 1;

        // Act
        const auto dobijenRezultat = lokator.dodajPorodicu(p1);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Metod dodajPorodicu ne pamti duplikate porodica")
    {
```

```
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto p2 = p1;
        const auto ocekivanRezultat = 1;

        // Act
        lokator.dodajPorodicu(p1);
        const auto dobijenRezultat = lokator.dodajPorodicu(p2);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Operator() prijavljuje izuzetak tipa invalid_argument
        ↪ ukoliko se prosledi kvadrant koji nije 1, 2, 3 ili 4")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);

        // Act + Assert
        REQUIRE_THROWS_AS(lokator(5), invalid_argument);
    }

    SECTION("Operator() vraca prazan optional ukoliko lokator nema
        ↪ porodica")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const optional<double> ocekivanRezultat;

        // Act
        const auto dobijenRezultat = lokator(1);

        // Assert
        REQUIRE(dobijenRezultat.has_value() == ocekivanRezultat.
            ↪ has_value());
    }

    SECTION("Operator() vraca rastojanje do jedne porodice iz
        ↪ ispravnog kvadranta koja je jedina dodata")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        lokator.dodajPorodicu(Porodica(3.0, 4.0));
        const optional<double> ocekivanRezultat(5.0);

        // Act
        const auto dobijenRezultat = lokator(1);

        // Assert
        CHECK(dobijenRezultat.has_value() == ocekivanRezultat.
            ↪ has_value());
        REQUIRE(dobijenRezultat.value() == ocekivanRezultat.value());
    }
}
```

Оператор позива функцијског објекта мора да провери да ли је колекција породица празна пре него ли приступи првом елементу.

```
#pragma once

#include <set>
#include <stdexcept>
#include <optional>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.insert(p);
        return _porodice.size();
    }

    optional<double> operator()(unsigned kvadrant) const
    {
        proveriKvadrant(kvadrant);

        if (_porodice.empty())
        {
            return {};
        }

        auto sumaRastojanja = 0.0;
        const auto &porodica = *_porodice.cbegin();
        const auto rastojanjeDoPorodice = porodica.rastojanjeDo(_x, _y
            ↪ );
        sumaRastojanja += rastojanjeDoPorodice;

        return sumaRastojanja;
    }

private:
    void proveriKvadrant(unsigned kvadrant) const
    {
        if (kvadrant != 1u && kvadrant != 2u && kvadrant != 3u &&
            ↪ kvadrant != 4u)
        {
            throw invalid_argument("kvadrant");
        }
    }

    double _x;
    double _y;
    set<Porodica> _porodice;
```

```
};
```

Функционалност израчунавања растојања између неке породице и произвољне координате смо енкапсулирали у класи *Породица*.

```
#pragma once

#include <cmath>

using namespace std;

class Porodica
{
public:
    Porodica(double x, double y)
        : _x(x), _y(y)
    {
    }

    bool operator==(const Porodica &p) const
    {
        return _x == p._x && _y == p._y;
    }

    bool operator<(const Porodica &p) const
    {
        return _x < p._x && _y < p._y;
    }

    double rastojanjeDo(double x, double y) const
    {
        return sqrt(pow(_x - x, 2) + pow(_y - y, 2));
    }

private:
    double _x;
    double _y;
};
```

Рефакторисање се састоји у издвајању помоћног метода за ажурирање растојања за једну породицу.

```
#pragma once

#include <set>
#include <stdexcept>
#include <optional>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }
```

```
    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.insert(p);
        return _porodice.size();
    }

    optional<double> operator()(unsigned kvadrant) const
    {
        proveriKvadrant(kvadrant);

        if (_porodice.empty())
        {
            return {};
        }

        auto sumaRastojanja = 0.0;
        const auto &porodica = *_porodice.cbegin();
        azurirajSumuRastojanjaZaPorodicu(porodica, sumaRastojanja);

        return sumaRastojanja;
    }

private:
    void azurirajSumuRastojanjaZaPorodicu(const Porodica &porodica,
       ↪ double &sumaRastojanja) const
    {
        const auto rastojanjeDoPorodice = porodica.rastojanjeDo(_x, _y
           ↪ );
        sumaRastojanja += rastojanjeDoPorodice;
    }

    void proveriKvadrant(unsigned kvadrant) const
    {
        if (kvadrant != 1u && kvadrant != 2u && kvadrant != 3u &&
           ↪ kvadrant != 4u)
        {
            throw invalid_argument("kvadrant");
        }
    }

    double _x;
    double _y;
    set<Porodica> _porodice;
};
```

Очекиван избор за наредни тест јесте случај сличан претходном са разликом да породица не живи у квадранту који се прослеђује оператору позива функцијског објекта.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include <stdexcept>
#include <optional>
#include "LokatorSupermarketa.hpp"
```

```
TEST_CASE("LokatorSupermarketa", "[class]")
{
    SECTION("Konstruktor LokatorSupermarketa uspesno konstruise
       ↪ nasumicni objekat")
    {
        // Arrange
        const auto x = 0.0;
        const auto y = 0.0;

        // Act + Assert
        REQUIRE_NOTHROW(LokatorSupermarketa(x, y));
    }

    SECTION("Metod dodajPorodicu uspesno pamti jednu novu porodicu")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto ocekivanRezultat = 1;

        // Act
        const auto dobijenRezultat = lokator.dodajPorodicu(p1);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Metod dodajPorodicu ne pamti duplikate porodica")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto p2 = p1;
        const auto ocekivanRezultat = 1;

        // Act
        lokator.dodajPorodicu(p1);
        const auto dobijenRezultat = lokator.dodajPorodicu(p2);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Operator() prijavljuje izuzetak tipa invalid_argument
       ↪ ukoliko se prosledi kvadrant koji nije 1, 2, 3 ili 4")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);

        // Act + Assert
        REQUIRE_THROWS_AS(lokator(5), invalid_argument);
    }

    SECTION("Operator() vraca prazan optional ukoliko lokator nema
       ↪ porodica")
    {
```

```
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const optional<double> ocekivanRezultat;

        // Act
        const auto dobijenRezultat = lokator(1);

        // Assert
        REQUIRE(dobijenRezultat.has_value() == ocekivanRezultat.
            ↪ has_value());
    }

    SECTION("Operator() vraca rastojanje do jedne porodice iz
        ↪ ispravnog kvadranta koja je jedina dodata")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        lokator.dodajPorodicu(Porodica(3.0, 4.0));
        const optional<double> ocekivanRezultat(5.0);

        // Act
        const auto dobijenRezultat = lokator(1);

        // Assert
        CHECK(dobijenRezultat.has_value() == ocekivanRezultat.
            ↪ has_value());
        REQUIRE(dobijenRezultat.value() == ocekivanRezultat.value());
    }

    SECTION("Operator() vraca prazan optional za lokator koji sadrzi
        ↪ jednu porodicu iz neispravnog kvadranta")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        lokator.dodajPorodicu(Porodica(3.0, 4.0));
        const optional<double> ocekivanRezultat;

        // Act
        const auto dobijenRezultat = lokator(2);

        // Assert
        REQUIRE(dobijenRezultat.has_value() == ocekivanRezultat.
            ↪ has_value());
    }
}
```

За имплементацију овог захтева, потребно је да објекти класе *Породица* знају да провере да ли се налазе у одговарајућем квадранту.

```
#pragma once

#include <cmath>

using namespace std;

class Porodica
{
public:
```

```
    Porodica(double x, double y)
        : _x(x), _y(y)
    {
    }

    bool operator==(const Porodica &p) const
    {
        return _x == p._x && _y == p._y;
    }

    bool operator<(const Porodica &p) const
    {
        return _x < p._x && _y < p._y;
    }

    double rastojanjeDo(double x, double y) const
    {
        return sqrt(pow(_x - x, 2) + pow(_y - y, 2));
    }

    bool ziviUKvadrantu(unsigned kvadrant) const
    {
        if (kvadrant == 1u)
        {
            return signbit(_x) == signbit(_y) && _x + _y >= 0.0;
        }
        if (kvadrant == 2u)
        {
            return signbit(_x) != signbit(_y) && _x <= 0.0 && _y >=
                ↪ 0.0;
        }
        if (kvadrant == 3u)
        {
            return signbit(_x) == signbit(_y) && _x + _y <= 0.0;
        }
        return signbit(_x) == signbit(_y) && _x >= 0.0 && _y <= 0.0;
    }

private:
    double _x;
    double _y;
};
```

Сада, локатор мора да врши додатну проверу, а то је да ли породица живи у квадранту, пре него што пређе на израчунавање суме растојања. Како је ова провера енкапсулирана у класу *Породица*, нећемо рефакторисати овај корак.

```
#pragma once

#include <set>
#include <stdexcept>
#include <optional>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
```

```
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.insert(p);
        return _porodice.size();
    }

    optional<double> operator()(unsigned kvadrant) const
    {
        proveriKvadrant(kvadrant);

        if (_porodice.empty())
        {
            return {};
        }

        const auto &porodica = *_porodice.cbegin();
        if (!porodica.ziviUKvadrantu(kvadrant))
        {
            return {};
        }

        auto sumaRastojanja = 0.0;
        azurirajSumuRastojanjaZaPorodicu(porodica, sumaRastojanja);
        return sumaRastojanja;
    }

private:
    void azurirajSumuRastojanjaZaPorodicu(const Porodica &porodica,
        ↪ double &sumaRastojanja) const
    {
        const auto rastojanjeDoPorodice = porodica.rastojanjeDo(_x, _y
            ↪ );
        sumaRastojanja += rastojanjeDoPorodice;
    }

    void proveriKvadrant(unsigned kvadrant) const
    {
        if (kvadrant != 1u && kvadrant != 2u && kvadrant != 3u &&
            ↪ kvadrant != 4u)
        {
            throw invalid_argument("kvadrant");
        }
    }

    double _x;
    double _y;
    set<Porodica> _porodice;
};
```

Коначно, преостало је да имплементирамо подршку за више породица од којих

се макар две налазе у траженом квадранту, а макар једна не.

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch.hpp"

#include <stdexcept>
#include <optional>
#include "LokatorSupermarketa.hpp"

TEST_CASE("LokatorSupermarketa", "[class]")
{
    SECTION("Konstruktor LokatorSupermarketa uspesno konstruise
       ↪ nasumicni objekat")
    {
        // Arrange
        const auto x = 0.0;
        const auto y = 0.0;

        // Act + Assert
        REQUIRE_NOTHROW(LokatorSupermarketa(x, y));
    }

    SECTION("Metod dodajPorodicu uspesno pamti jednu novu porodicu")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto ocekivanRezultat = 1;

        // Act
        const auto dobijenRezultat = lokator.dodajPorodicu(p1);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Metod dodajPorodicu ne pamti duplikate porodica")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        const Porodica p1(3.0, 4.0);
        const auto p2 = p1;
        const auto ocekivanRezultat = 1;

        // Act
        lokator.dodajPorodicu(p1);
        const auto dobijenRezultat = lokator.dodajPorodicu(p2);

        // Assert
        REQUIRE(dobijenRezultat == ocekivanRezultat);
    }

    SECTION("Operator() prijavljuje izuzetak tipa invalid_argument
       ↪ ukoliko se prosledi kvadrant koji nije 1, 2, 3 ili 4")
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
```

```
    // Act + Assert
    REQUIRE_THROWS_AS(lokator(5), invalid_argument);
}

SECTION("Operator() vraca prazan optional ukoliko lokator nema
    ↪ porodica")
{
    // Arrange
    LokatorSupermarketa lokator(0.0, 0.0);
    const optional<double> ocekivanRezultat;

    // Act
    const auto dobijenRezultat = lokator(1);

    // Assert
    REQUIRE(dobijenRezultat.has_value() == ocekivanRezultat.
        ↪ has_value());
}

SECTION("Operator() vraca rastojanje do jedne porodice iz
    ↪ ispravnog kvadranta koja je jedina dodata")
{
    // Arrange
    LokatorSupermarketa lokator(0.0, 0.0);
    lokator.dodajPorodicu(Porodica(3.0, 4.0));
    const optional<double> ocekivanRezultat(5.0);

    // Act
    const auto dobijenRezultat = lokator(1);

    // Assert
    CHECK(dobijenRezultat.has_value() == ocekivanRezultat.
        ↪ has_value());
    REQUIRE(dobijenRezultat.value() == ocekivanRezultat.value());
}

SECTION("Operator() vraca prazan optional za lokator koji sadrzi
    ↪ jednu porodicu iz neispravnog kvadranta")
{
    // Arrange
    LokatorSupermarketa lokator(0.0, 0.0);
    lokator.dodajPorodicu(Porodica(3.0, 4.0));
    const optional<double> ocekivanRezultat;

    // Act
    const auto dobijenRezultat = lokator(2);

    // Assert
    REQUIRE(dobijenRezultat.has_value() == ocekivanRezultat.
        ↪ has_value());
}

SECTION("Operator() vraca ukupno rastojanje za lokator koji sadrzi
    ↪  vise od jedne porodice iz ispravnog kvadranta i makar jednu
    ↪  porodicu iz neispravnog kvadranta")
```

```
    {
        // Arrange
        LokatorSupermarketa lokator(0.0, 0.0);
        lokator.dodajPorodicu(Porodica(3.0, 4.0));
        lokator.dodajPorodicu(Porodica(-3.0, 4.0));
        lokator.dodajPorodicu(Porodica(6.0, 8.0));
        const optional<double> ocekivanRezultat(15.0);

        // Act
        const auto dobijenRezultat = lokator(1);

        // Assert
        REQUIRE(dobijenRezultat.has_value() == ocekivanRezultat.
            ↪ has_value());
        REQUIRE(dobijenRezultat.value() == ocekivanRezultat.value());
    }
}
```

Прво издвајамо у помоћну колекцију све оне породице које се налазе у траженом квадранту. Затим, ако макар једна породица живи у том квадранту, израчунавамо суму растојања.

```
#pragma once

#include <set>
#include <stdexcept>
#include <optional>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.insert(p);
        return _porodice.size();
    }

    optional<double> operator()(unsigned kvadrant) const
    {
        proveriKvadrant(kvadrant);

        set<Porodica> porodiceIzKvadranta;
        for (auto iter = _porodice.cbegin(); iter != _porodice.cend();
            ↪  ++iter)
        {
            if (iter->ziviUKvadrantu(kvadrant))
            {
                porodiceIzKvadranta.insert(*iter);
            }
```

```
        }
        if (porodiceIzKvadranta.empty())
        {
            return {};
        }

        auto sumaRastojanja = 0.0;
        for (auto iter = porodiceIzKvadranta.cbegin(); iter !=
            ↪ porodiceIzKvadranta.cend(); ++iter)
        {
            azurirajSumuRastojanjaZaPorodicu(*iter, sumaRastojanja);
        }

        return sumaRastojanja;
    }

private:
    void azurirajSumuRastojanjaZaPorodicu(const Porodica &porodica,
        ↪ double &sumaRastojanja) const
    {
        const auto rastojanjeDoPorodice = porodica.rastojanjeDo(_x, _y
            ↪ );
        sumaRastojanja += rastojanjeDoPorodice;
    }

    void proveriKvadrant(unsigned kvadrant) const
    {
        if (kvadrant != 1u && kvadrant != 2u && kvadrant != 3u &&
            ↪ kvadrant != 4u)
        {
            throw invalid_argument("kvadrant");
        }
    }

    double _x;
    double _y;
    set<Porodica> _porodice;
};
```

Оператор позива функцијског објекта имплементира више од једне функционалности, те ћемо их све издвојити у помоћне методе. Тиме завршавамо развој класе.

```
#pragma once

#include <set>
#include <stdexcept>
#include <optional>
#include "Porodica.hpp"

using namespace std;

class LokatorSupermarketa
{
public:
    LokatorSupermarketa(double x, double y)
        : _x(x), _y(y)
```

```
    {
    }

    size_t dodajPorodicu(const Porodica &p)
    {
        _porodice.insert(p);
        return _porodice.size();
    }

    optional<double> operator()(unsigned kvadrant) const
    {
        proveriKvadrant(kvadrant);

        const auto porodiceIzKvadranta = izdvojPorodiceIzKvadranta(
            ↪ kvadrant);
        if (porodiceIzKvadranta.empty())
        {
            return {};
        }

        return sumirajRastojanjaDoPorodica(porodiceIzKvadranta);
    }

private:
    set<Porodica> izdvojPorodiceIzKvadranta(unsigned kvadrant) const
    {
        set<Porodica> porodiceIzKvadranta;
        for (auto iter = _porodice.cbegin(); iter != _porodice.cend();
            ↪  ++iter)
        {
            if (iter->ziviUKvadrantu(kvadrant))
            {
                porodiceIzKvadranta.insert(*iter);
            }
        }
        return porodiceIzKvadranta;
    }

    double sumirajRastojanjaDoPorodica(const set<Porodica> &porodice)
        ↪ const
    {
        auto sumaRastojanja = 0.0;
        for (auto iter = porodice.cbegin(); iter != porodice.cend();
            ↪ ++iter)
        {
            azurirajSumuRastojanjaZaPorodicu(*iter, sumaRastojanja);
        }

        return sumaRastojanja;
    }

    void azurirajSumuRastojanjaZaPorodicu(const Porodica &porodica,
        ↪ double &sumaRastojanja) const
    {
        const auto rastojanjeDoPorodice = porodica.rastojanjeDo(_x, _y
            ↪ );
```

```
        sumaRastojanja += rastojanjeDoPorodice;
    }

    void proveriKvadrant(unsigned kvadrant) const
    {
        if (kvadrant != 1u && kvadrant != 2u && kvadrant != 3u &&
            ↪ kvadrant != 4u)
        {
            throw invalid_argument("kvadrant");
        }
    }

    double _x;
    double _y;
    set<Porodica> _porodice;
};
```

Задатак 3.2.2.

Решење 3.2.3.

Задатак 3.2.3.

Решење 3.2.4.

Задатак 3.2.4.

Решење 3.2.5.

Задатак 3.2.5.

Глава 4

Рефакторисање и свођење проблема

Ово поглавље је у припреми!

4.1 Рефакторисање

Задатак 4.1.1. Препознати „заударање” у наредном коду и применити одговарајућу технику рефакторисања како би се решио проблем.

```
class RegistarZaposlenih
{
    // ...

public:
    void IsplatiZaraduZaposlenom(Zaposleni z)
    {
        auto ukupanBrojSati = 0;
        for (const auto &radniDan : z.RadniDani())
        {
            ukupanBrojSati += radniDan.brojRadnihSati();
        }

        auto indeksRadnika = -1;
        auto i = 0;
        for (const auto &radnik : Radnici)
        {
            if (radnik == z)
            {
                indeksRadnika = i;
                break;
            }
            ++i;
        }
        if (indeksRadnika == -1)
        {
            return;
        }
        const auto ukupnaZarada = ukupanBrojSati * Ugovori[
            ↪ indeksRadnika].ZaradaPoSatu();
```

```
        const ZahtevZaIsplatu zahtev(InformacijeOKompaniji.Racun());
        zahtev.PostaviRacun(z.Racun());
        zahtev.PostaviIznos(ukupnaZarada);
        zahtev.Posalji();

        const ElektronskaPoruka ePoruka;
        ePoruka.PostaviPosiljaoca(InformacijeOKompaniji.EAdresa());
        ePoruka.DodajPrimaoca(z.EAdresa());
        ePoruka.DodajPoruku("Postovani " + z.Ime() +
            ",\n\nNa Vas racun je uplacena zarada u iznosu od " +
            to_string(ukupnaZarada) + " dinara.");
        ePoruka.Posalji();
    }

private:
    vector<Zaposleni> Radnici;
    vector<Ugovor> Ugovori;
    Kompanija InformacijeOKompaniji;
};
```

Решење 4.1.1.

Задатак 4.1.2. Препознати „заударање” у наредном коду и применити одговарајућу технику рефакторисања како би се решио проблем.

```
class RegistarZaposlenih
{
    // ...

public:
    int PronadjiIndeksRadnika(Zaposleni z) const
    {
        auto indeksRadnika = -1;
        auto i = 0;
        for (const auto &radnik : Radnici)
        {
            if (radnik == z)
            {
                indeksRadnika = i;
                break;
            }
            ++i;
        }
        return indeksRadnika;
    }

    double IzracunajUkupanBrojRadnihSati(Zaposleni z) const
    {
        auto ukupanBrojSati = 0;
        for (const auto &radniDan : z.RadniDani())
        {
            ukupanBrojSati += radniDan.brojRadnihSati();
        }
        return ukupanBrojSati;
    }
```

```
    void PosaljiZahtevZaIsplatu(Zaposleni z, double ukupnaZarada)
        ↪ const
    {
        const ZahtevZaIsplatu zahtev(InformacijeOKompaniji.Racun());
        zahtev.PostaviRacun(z.Racun());
        zahtev.PostaviIznos(ukupnaZarada);
        zahtev.Posalji();
    }

    void PosaljiElektronskuPorukuZaposlenom(Zaposleni z) const
    {
        const ElektronskaPoruka ePoruka;
        ePoruka.PostaviPosiljaoca(InformacijeOKompaniji.EAdresa());
        ePoruka.DodajPrimaoca(z.EAdresa());
        ePoruka.DodajPoruku("Postovani " + z.Ime() +
            ",\n\nNa Vas racun je uplacena zarada u iznosu od " +
            to_string(ukupnaZarada) + " dinara.");
        ePoruka.Posalji();
    }

private:
    vector<Zaposleni> Radnici;
    vector<Ugovor> Ugovori;
    Kompanija InformacijeOKompaniji;
};
```

Решење 4.1.2.

Задатак 4.1.3. Препознати „заударање” у наредном коду и применити одговарајућу технику рефакторисања како би се решио проблем.

```
class Student
{
    // ...
private:
    int Indeks;
    string Jmbg;
    string EAdresa;
    double ProsecnaOcena;
    string DatumUpisa;
};
```

Решење 4.1.3.

Задатак 4.1.4. Препознати „заударање” у наредном коду и применити одговарајућу технику рефакторисања како би се решио проблем.

Решење 4.1.4.

Задатак 4.1.5. Препознати „заударање” у наредном коду и применити одговарајућу технику рефакторисања како би се решио проблем.

```
class ObradjivacOperacija
{
    // ...
```

```
public:
    void ObradiOperaciju(Operacija *o, Podatak *p)
    {
        switch (o->Tip())
        {
        case Operacija::Tip::Kopiraj:
            dynamic_cast<Kopiraj *>(o)->Kopiraj(p);
        case Operacija::Tip::Iseci:
            dynamic_cast<Iseci *>(o)->Iseci(p);
        case Operacija::Tip::Nalepi:
            dynamic_cast<Nalepi *>(o)->Nalepi(p);
        default:
            throw Izuzetak("Nepoznata operacija: " + to_string(o->Tip
                ↪ ()));
        }
    }
};
```

Решење 4.1.5.

Задатак 4.1.6. Препознати „заударање” у наредном коду и применити одговарајућу технику рефакторисања како би се решио проблем.

Решење 4.1.6.

Задатак 4.1.7. Препознати „заударање” у наредном коду и применити одговарајућу технику рефакторисања како би се решио проблем.

```
class KolekcijaStudenata
{
    // ...
public:
    vector<Student> DohvatiStudente() const
    {
        return Studenti;
    }

    void PostaviStudente(vector<Student> studenti)
    {
        Studenti = studenti;
    }
private:
    vector<Student> Studenti;
};

class Kontroler
{
    // ...
public:
    Student DohvatiNajboljegStudenta() const
    {
        auto najboljiProsek = 0.0;
        Student najboljiStudent;
        for (const auto &student : Studenti.DohvatiStudente())
        {
```

```
            if (student.Prosek() > najboljiProsek)
            {
                najboljiProsek = student.Prosek();
                najboljiStudent = student;
            }
        }

        return najboljiStudent;
    }

    void DodajNovogStudenta(Student s)
    {
        auto tekuciStudenti = Studenti.DohvatiStudente();
        for (const auto &student : Studenti.DohvatiStudente())
        {
            if (student == s)
            {
                return;
            }
        }
        tekuciStudenti.push_back(s);
        Studenti.PostaviStudente(tekuciStudenti);
    }
private:
    KolekcijaStudenata Studenti;
};
```

Решење 4.1.7.

Задатак 4.1.8. Препознати „заударање” у наредном коду и применити одговарајућу технику рефакторисања како би се решио проблем.

Решење 4.1.8.

4.2 Свођење проблема

Задатак 4.2.1. Написати функцију *бројеви* која са стандардног улаза учитава целе бројеве до краја улаза, а затим те бројеве уређује у растућем редоследу и исписује их на стандардни излаз у растућем и опадајућем редоследу. Свести функцију на алгоритме стандардне библиотеке.

Решење 4.2.1.

Задатак 4.2.2. Написати функцију *палиндром* која за дати стринг проверава да ли је палиндром. Свести функцију на алгоритме стандардне библиотеке.

Решење 4.2.2.

Задатак 4.2.3. Написати функцију *пронађиОдабране* која за дати вектор стрингова израчунава вектор индикатора. Индикатори у резултујућем вектору имају вредност `true` ако стринг почиње карактером звезде, а `false` иначе. Свести функцију на алгоритме стандардне библиотеке.

Решење 4.2.3.

Задатак 4.2.4. Написати функцију *направиНискуОдНепарних* која за дати вектор целих бројева израчунава ниску која се састоји само од непарних бројева из датог вектора. Свести функцију на алгоритме стандардне библиотеке.

Решење 4.2.4.

Задатак 4.2.5. Написати функцију *распоредиБројеве* која за дати вектор целих бројева израчунава вектор који има распоређене бројеве тако да се на почетку налазе бројеви који су дељиви бројем три, затим бројеви који дају остатак један при дељењу бројем три и, на крају, бројеви који дају остатак два при дељењу бројем три. Свести функцију на алгоритме стандардне библиотеке.

Решење 4.2.5.

Задатак 4.2.6. Написати функцију *селекција* која прихвата три итератора на вектор ниски: (1) итератор на почетак селекције – *почетак*, (2) итератор на крај селекције – *крај* и (3) итератор на позицију испред које је потребно померити интервал дефинисан итераторима *почетак* и *крај* – *одредиште*. Функција враћа пар итератора на цео интервал у којем је дошло до померања. Свести функцију на алгоритме стандардне библиотеке.

Решење 4.2.6.

Задатак 4.2.7. Написати функцију *бројУзастопнихЈеднаких* која прихвата вектор целих бројева и израчунава број узастопних једнаких бројева датог вектора. Свести функцију на алгоритме стандардне библиотеке.

Решење 4.2.7.

Задатак 4.2.8. Написати функцију *провериУређеност* која за дати стринг проверава да ли су карактери уређени неопадајуће. Свести функцију на алгоритме стандардне библиотеке.

Решење 4.2.8.

Задатак 4.2.9. Свести функцију из задатка 3.1.1 на алгоритме стандардне библиотеке, а затим, написаним тестовима проверити тачност рефакторисаног решења.

Решење 4.2.9.

Задатак 4.2.10. Свести функцију из задатка 3.1.2 на алгоритме стандардне библиотеке, а затим, написаним тестовима проверити тачност рефакторисаног решења.

Решење 4.2.10.

Задатак 4.2.11. Свести функцију из задатка 3.1.3 на алгоритме стандардне библиотеке, а затим, написаним тестовима проверити тачност рефакторисаног решења.

Решење 4.2.11.

Задатак 4.2.12. Свести функцију из задатка 3.1.4 на алгоритме стандардне библиотеке, а затим, написаним тестовима проверити тачност рефакторисаног решења.

Решење 4.2.12.

Задатак 4.2.13. Свести функцију из задатка 3.1.5 на алгоритме стандардне библиотеке, а затим, написаним тестовима проверити тачност рефакторисаног решења.

Решење 4.2.13.

4.3 Решења задатака

Рефакторисање

Решење 4.1.1.

Задатак 4.1.1.

Решење 4.1.2.

Задатак 4.1.2.

Решење 4.1.3.

Задатак 4.1.3.

Решење 4.1.4.

Задатак 4.1.4.

Решење 4.1.5.

Задатак 4.1.5.

Решење 4.1.6.

Задатак 4.1.6.

Решење 4.1.7.

Задатак 4.1.7.

Решење 4.1.8.

Задатак 4.1.8.

Свођење проблема

Решење 4.2.1. Учитавање вредности са стандардног улаза и исписивање вредности на стандардни излаз можемо свести на алгоритам копирања. У ту сврху, неопходно је направити итераторе на улазни ток и излазни ток, за шта нам могу помоћи шаблонске класе `istream_iterator` и `ostream_iterator`. Након што извршимо копирање вредности са стандардног улазног тока у помоћну колекцију, можемо је сортирати, а затим копирати вредности на стандардни излазни ток – прво у редоследу од почетка до краја колекције, а затим у обрнутом редоследу.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

void brojevi()
{
    vector<int> xs;

    copy(istream_iterator<int>(cin),
         istream_iterator<int>(),
```

```
        back_inserter(xs));
    sort(begin(xs),
        end(xs));
    copy(cbegin(xs),
        cend(xs),
        ostream_iterator<int>(cout, " "));
    cout << endl;
    copy(crbegin(xs),
        crend(xs),
        ostream_iterator<int>(cout, " "));
    cout << endl;
}
```

Задатак 4.2.1.

Решење 4.2.2. Испитивање да ли је ниска палиндром могуће је извршити поређењем по једнакости карактера од почетка до краја и карактера од краја до почетка. Напоменимо да алгоритам `equal` не проверава да ли су интервали једнаке дужине. У овом задатку то није неопходно с обзиром да су дужина интервала од почетка до краја и дужина интервала од краја до почетка исте ниске, једнаке.

```
#include <iostream>
#include <algorithm>
#include <string>
#include <iterator>

using namespace std;

bool palindrom(const string &str)
{
    return equal(cbegin(str), cend(str), crbegin(str));
}
```

Задатак 4.2.2.

Решење 4.2.3. Ово је класичан пример класе проблема трансформисања сваке вредности полазног интервала у неку нову вредност (не нужно истог типа). У те сврхе користимо алгоритам `transform` који прихвата почетак и крај интервала елемената које је неопходно трансформисати у нове вредности, затим итератор на почетак колекције у коју ће бити смештене вредности добијене трансформацијом и позивни објекат који извршава трансформацију.

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>

using namespace std;

vector<bool> pronadjiOdabrane(const vector<string> &stavke)
{
    vector<bool> indikatori;

    transform(cbegin(stavke),
```

```
            cend(stavke),
            back_inserter(indikatori),
            [](const auto &stavka)
            { return !stavka.empty() && stavka[0] == '*'; });

    return indikatori;
}
```

Задатак 4.2.3.

Решење 4.2.4. Овај задатак бисмо могли да урадимо прво трансформисањем свих бројева из вектора у стрингове алгоритмом `transform` тако што се непарни бројеви трансформишу у своје карактерне репрезенције, а парни бројеви у празне стрингове. Затим бисмо могли да применимо алгоритам `accumulate` тако да надовежемо све ниске. Међутим, бржа алтернатива јесте да операције трансформације и надовезивања компонујемо у једну и да је искористимо у алгоритму `accumulate`.

```
#include <vector>
#include <numeric>
#include <iostream>
#include <string>

using namespace std;

string napraviNiskuOdNeparnih(const vector<int> &brojevi)
{
    return accumulate(
        cbegin(brojevi),
        cend(brojevi),
        string(),
        [](const auto &akumulator, const auto &broj)
        { return broj % 2 != 0 ? (akumulator + to_string(broj)) :
            ↪ akumulator; });
}
```

Задатак 4.2.4.

Решење 4.2.5. За ово решење можемо искористити алгоритам `stable_partition` који дели интервал одређен итераторима на два подинтервала коришћењем предикатског позивног објекта. Након партиционисања, први подинтервал ће се састојати од елемената из интервала који задовољавају дати предикат, а други подинтервал ће се састојати од преосталих елемената. Алгоритам враћа итератор на први елемент из другог подинтервала. Дакле, искористићемо овај алгоритам два пута. Први пут партиционишемо читаву колекцију на све бројеве дељиве бројем три и оне које то нису. Други пут партиционишемо само онај део колекције који садржи елементе из другог подинтервала претходног партиционисања. Друго партиционисање вршимо по бројевима који дају остатак један при дељењу бројем три.

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator>

using namespace std;
```

```
using ParIteratora = pair<vector<int>::const_iterator, vector<int>::
    ↪ const_iterator>;

ParIteratora rasporediBrojeve(vector<int> &brojevi)
{
    const auto krajPrveParticije =
        stable_partition(begin(brojevi),
                         end(brojevi),
                         [](const auto &broj)
                         { return broj % 3 == 0; });
    const auto krajDrugeParticije =
        stable_partition(krajPrveParticije,
                         end(brojevi),
                         [](const auto &broj)
                         { return broj % 3 == 1; });
    return {krajPrveParticije, krajDrugeParticije};
}
```

Задатак 4.2.5.

Решење 4.2.6. Овде користимо алгоритам `rotate` који извршава леву ротацију интервала одређен итераторима *први* и *последњи* (први и трећи аргумент алгоритма) око итератора *новиПрви* (други аргумент алгоритма). Алгоритам ротира елементе из интервала $[\textit{први}, \textit{последњи}]$ тако да елемент на који показује итератор *новиПрви* постане први елемент у интервалу, а елемент на који показује итератор $\textit{новиПрви} - 1$ постане последњи елемент у интервалу. Алгоритам враћа итератор на нову локацију елемента на који показује итератор *први*, тј. $\textit{први} + (\textit{последњи} - \textit{новиПрви})$.

Да бисмо искористили како се овај алгоритам користи, треба препознати наредне три ситуације до којих може доћи у извршавању наше функције (називи итератора у наставку решења се односе на називе аргумената из текста задатка):

- Итератор *одредиште* се налази испред итератора *почетак*. У овом случају:
  - Потребно је ротирати елементе интервала $[\textit{одредиште}, \textit{крај})$ тако да елемент на који показује итератор *почетак* буде нови први елемент.
  - Повратна вредност је интервал одређен итератором *одредиште* и повратном вредности алгоритма `rotate`.
- Итератор *одредиште* се налази након итератора *крај*. У овом случају:
  - Потребно је ротирати елементе интервала $[\textit{почетак}, \textit{одредиште})$ тако да елемент на који показује итератор *крај* буде нови први елемент.
  - Повратна вредност је интервал одређен повратном вредности алгоритма `rotate` и итератором *одредиште*.
- Итератор *одредиште* се налази унутар интервала дефинисаним итераторима *почетак* и *крај*. У овом случају:
  - Потребно је ротирати елементе интервала $[\textit{почетак} - (\textit{крај} - \textit{одредиште}), \textit{крај})$ тако да елемент на који показује итератор *одредиште* буде нови први елемент.
  - Повратна вредност је интервал одређен итераторима $\textit{почетак} - (\textit{крај} - \textit{одредиште})$ и повратном вредности алгоритма `rotate`.

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <iterator>
#include <string>

using namespace std;

using StringIter = vector<string>::iterator;

pair<StringIter, StringIter> slide_selection(StringIter pocetak,
    ↪ StringIter kraj, StringIter odrediste)
{
    if (odrediste < pocetak)
    {
        return {odrediste, rotate(odrediste, pocetak, kraj)};
    }

    if (kraj < odrediste)
    {
        return {rotate(pocetak, kraj, odrediste), odrediste};
    }

    const auto first = pocetak - (kraj - odrediste);
    return {first, rotate(first, pocetak, kraj)};
}
```

Задатак 4.2.6.

Решење 4.2.7. Захтев задатка подразумева да посматрамо парове суседних вредности колекције. За сваки пар треба одредити да ли су вредности једнаке, а затим треба агрегирати број једнаких вредности. Дакле, имамо две операције: поређење вредности по једнакости и агрегација резултата поређења. Ово би требало да нас подсети на скаларни производ вектора. У скаларном производу такође имамо две операције: множење вредности над паровима координата и сабирање добијених производа. Другим речима, проблем из овог задатка можемо свести на скаларни производ. У ту сврху можемо искористити алгоритам `inner_product`, који примењујемо тако што дефинишемо операције „множења” и „сабирања” у терминима нашег проблема. Још један „трик” који треба приметити у решењу јесте да немамо два вектора чији скаларни производ рачунамо, већ први „вектор” представља елементе од првог до претпоследњег елемента у датој колекцији, а други „вектор” представља елементе од другог до последњег елемента у датој колекцији.

```
#include <iostream>
#include <string>
#include <numeric>
#include <vector>
#include <iterator>

using namespace std;

int brojUzastopnihJednakih(const vector<int> &brojevi)
{
    return inner_product(
        cbegin(brojevi),
```

```
        cend(brojevi) - 1,
        cbegin(brojevi) + 1,
        int(),
        [](const auto &akumulator, const auto &broj)
        { return akumulator + broj; },
        [](const auto &leviBroj, const auto &desniBroj)
        { return leviBroj == desniBroj ? 1 : 0; });
}
```

Задатак 4.2.7.

Решење 4.2.8. Решење овог задатка користи исти приступ као претходно решење, са разликом у дефиницији операција „множења” (у овом случају, испитивање уређења карактера ниске) и „сабирања” (у овом случају, конјукција Булових вредности) у скаларном производу.

```
#include <iostream>
#include <string>
#include <numeric>
#include <vector>
#include <iterator>

using namespace std;

bool proveriUredjenost(const string &tekst)
{
    return inner_product(
        cbegin(tekst),
        cend(tekst) - 1,
        cbegin(tekst) + 1,
        true,
        [](const auto &akumulator, const auto &jednaki)
        { return akumulator && jednaki; },
        [](const auto &leviKarakter, const auto &desniKarakter)
        { return leviKarakter <= desniKarakter; });
}
```

Задатак 4.2.8.

Решење 4.2.9.

Задатак 4.2.9.

Решење 4.2.10.

Задатак 4.2.10.

Решење 4.2.11.

Задатак 4.2.11.

Решење 4.2.12.

Задатак 4.2.12.

Решење 4.2.13.

Задатак 4.2.13.

Глава 5

Параметарски полиморфизам

Ово поглавље је у припреми!

5.1 Шаблонске функције

Задатак 5.1.1. Написати шаблонску функцију *већиОд* која прихвата две вредности и враћа већу од њих.

Решење 5.1.1.

Задатак 5.1.2. Написати шаблонску функцију *иницијализуј* која прихвата итераторе на почетак и крај интервала и иницијалну вредност. Функција иницијализује дати интервал инкрементирањем иницијалне вредности.

Решење 5.1.2.

Задатак 5.1.3. Написати шаблонску функцију *највећиЕлемент* која проналази итератор на највећи елемент у интервалу који је одређен итераторима на почетак и крај, а који се прослеђују као аргументи ове функције.

Решење 5.1.3.

Задатак 5.1.4. Написати шаблонску функцију *акумулирај* која израчунава збир елемената интервала који је одређен итераторима на почетак и крај, а који се прослеђују као аргументи ове функције. Функција прихвата и додатни аргумент који представља иницијалну вредност збира.

Решење 5.1.4.

Задатак 5.1.5. Написати шаблонску функцију *селекцијаСаФилтрирањем* која прихвата четири аргумента: (1) итератор на почетак интервала – *почетак*, (2) итератор на крај интервала – *крај*, (3) итератор на позицију око којег је потребно померити елементе из интервала дефинисаним првим двама итераторима – *одредиште* и (4) „позивни објекат” (ламбда функцију или функцијски објекат) који одређује који елементи из интервала ће бити померени око итератора *одредиште* – *предикат*. Претпоставити да се итератор *одредиште* налази у интервалу дефинисаном итераторима *почетак* и *крај*.

Решење 5.1.5.

Задатак 5.1.6. „Шаблонизовати” функцију из задатка 4.2.7.

Решење 5.1.6.

Задатак 5.1.7. „Шаблонизовати” функцију из задатка 4.2.8.

Решење 5.1.7.

5.2 Шаблонске класе

Задатак 5.2.1. Написати шаблонску класу *Пар* која садржи две вредности потенцијално различитог типа. Имплементирати метод за размену вредности са другим објектом исте класе.

Решење 5.2.1.

Задатак 5.2.2. Написати шаблонску класу *Поређење* која имплементира најопштије операторе поређења по једнакости и неједнакости. Искористити ову класу за имплементацију ових оператора над класом из задатка 5.2.1.

Решење 5.2.2.

Задатак 5.2.3. „Шаблонизовати” класу из задатка 2.2.2.

Решење 5.2.3.

5.3 Решења задатака

Шаблонске функције

Решење 5.1.1.

Задатак 5.1.1.

Решење 5.1.2.

Задатак 5.1.2.

Решење 5.1.3.

Задатак 5.1.3.

Решење 5.1.4.

Задатак 5.1.4.

Решење 5.1.5. Потребно је партиционисати два полуотворена подинтервала: први је подинтервал одређен итераторима *почетак* и *одредиште*, а други је подинтервал одређен итераторима *одредиште* и *крај*. Приликом партиционисања првог подинтервала, потребно је користити позивни објекат (на пример, ламбда функцију) који враћа супротан резултат у односу на прослеђени позивни објекат *предикат*, како би се на почетку овог подинтервала налазиле вредности које не задовољавају услове позивног објекта *предикат*, а на његовом крају (тј. испред итератора *одредиште*) налазиле вредности које задовољавају услове позивног објекта *предикат*.

```
#include <algorithm>

using namespace std;

template <typename It, typename Pred>
void selekcijaSaFiltriranjem(It pocetak, It kraj, It odrediste, Pred
    ↪ predikat)
{
    stable_partition(pocetak, odrediste, [predikat](const auto &
        ↪ element) { return !predikat(element); });
    stable_partition(odrediste, kraj, predikat);
}
```

Задатак 5.1.5.

Решење 5.1.6.

Задатак 5.1.6.

Решење 5.1.7.

Задатак 5.1.7.

Шаблонске класе

Решење 5.2.1.

Задатак 5.2.1.

Решење 5.2.2.

Задатак 5.2.2.

Решење 5.2.3.

Задатак 5.2.3.

Глава 6

Конкурентно програмирање

Ово поглавље је у припреми!

6.1 Конкурентни задаци

6.2 Синхронизација конкурентних задатака

6.3 Решења задатака

Конкурентни задаци

Синхронизација конкурентних задатака

Глава 7

Серијализација и десеријализација

Ово поглавље је у припреми!

7.1 Обрада XML података

7.2 Обрада JSON података

7.3 Обрада бинарних података

7.4 Решења задатака

Обрада XML података

Обрада JSON података

Обрада бинарних података

Глава 8

Апликације са графичким корисничким интерфејсом

Ово поглавље је у припреми!

Како би се свакодневним корисницима олакшало коришћење апликација, програмери имплементирају *графичке корисничке интерфејсе* путем којих корисници „комуницирају” са програмским интерфејсом услуга које те апликације пружају. Поред дизајнирања графичких корисничких интерфејса који су „удобни за рад”, треба узети у обзир да редослед операција које један корисник извршава не мора бити нужно исти као редослед операција које неки други корисник извршава, посредством тог интерфејса. У том смислу, програмери су дужни да обезбеде провере исправности редоследа операција и да упозоре кориснике уколико они користе графички кориснички интерфејс на неисправан начин. У овој глави разматрамо технике дизајна и развоја апликација са графичким корисничким интерфејсом. Додатно, разматрамо специјалну класу ових апликација које, поред класичних контрола као што су поље за унос, дугме и сл. подразумевају и управљање графичким елементима, као што су прости облици (правоугаоници, елипсе), текст, слике и др.

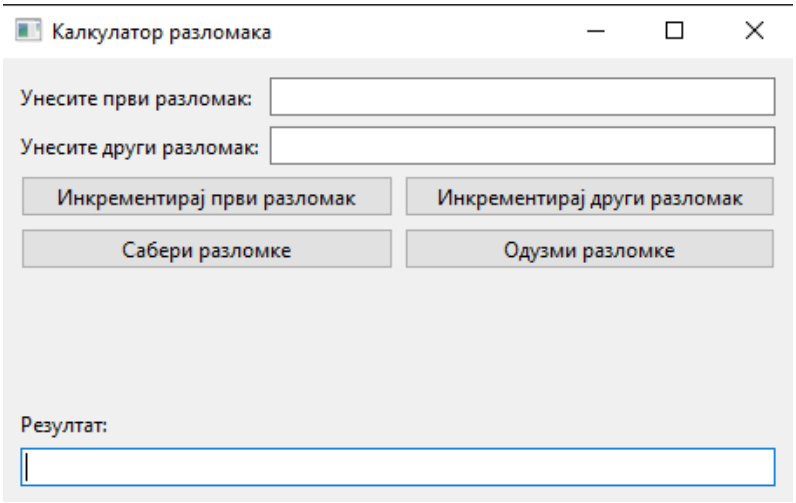
8.1 Графички кориснички интерфејси

Задатак 8.1.1. Коришћењем библиотеке *Qt* имплементирати апликацију за једноставне операције са разломцима. Графички кориснички интерфејс дизајнирати као на слици испод. Омогућити следеће операције:

- Кликом на дугме „Инкрементирај први разломак”, апликација чита вредност првог разломка. Уколико је разломак унет у исправном формату, инкрементирати га и исписати нови разломак у истом пољу. У супротном, приказати поруку о грешци у пољу „Резултат”.
- Кликом на дугме „Инкрементирај други разломак”, апликација чита вредност другог разломка. Уколико је разломак унет у исправном формату, инкрементирати га и исписати нови разломак у истом пољу. У супротном, приказати поруку о грешци у пољу „Резултат”.
- Кликом на дугме „Сабери разломке”, апликација чита вредности оба разломка. Уколико су оба разломка унета у исправном формату, сабрати их и прика-

зати резултат у пољу „Резултат”. У супротном, приказати поруку о грешци у истом пољу.

- Кликом на дугме „Одузми разломке”, апликација чита вредности оба разломка. Уколико су оба разломка унета у исправном формату, одузети их и приказати резултат у пољу „Резултат”. У супротном, приказати поруку о грешци у истом пољу.



Решење 8.1.1.

Задатак 8.1.2. Коришћењем библиотеке *Qt* имплементирати апликацију за израчунавање сложених аритметичких израза са разломцима. Графички кориснички интерфејс дизајнирати као на наредној слици.

Омогућити следеће операције:

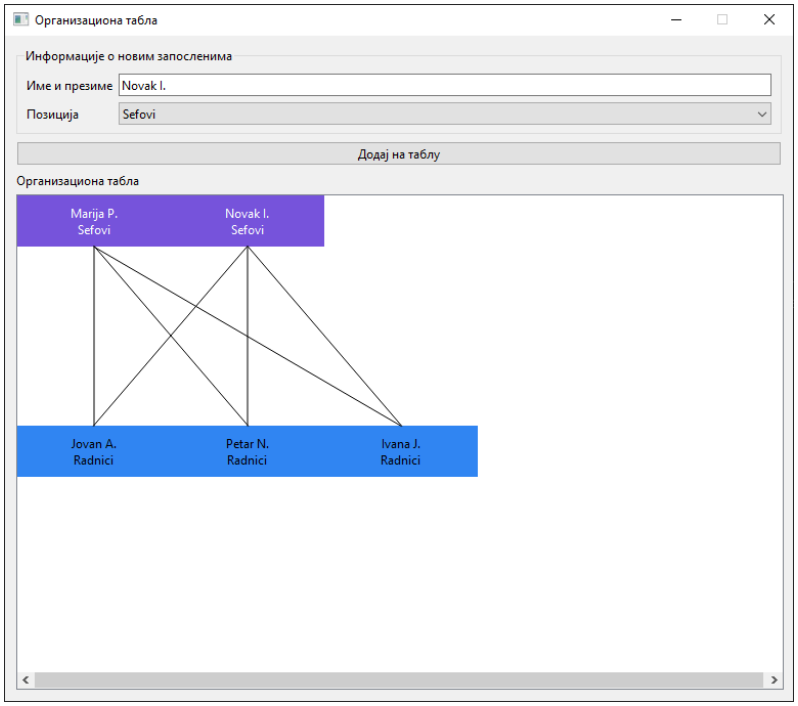
- Кликом на дугмад са леве стране, корисник уноси разломке који формирају сложени аритметички израз. Приказати унос у једнолинијском пољу десно. Кликом на дугмад „++”, „-”, „∼” или „C” апликација редом: инкрементира, израчунава супротни разломак, израчунава реципрочни разломак или поништава унос, за тренутно унети разломак. Резултате ових операција унети у исто поље.

- Кликом на дугмад „+” и „-” са десне стране, апликација додаје тренутни разломак у сложени аритметички израз сабирањем, односно, одузимањем од тренутног стања израза. Израз се записује у вишелинијском пољу са десне стране.
- Кликом на дугме „=”, апликација на израз додаје знак „=” и исписује резултат аритметичког израза.
- Кликом на дугме „Поништи израчунавање” се брише тренутно стање сложеног израза, а кликом на дугме „Изађи” се апликација завршава.

Решење 8.1.2.

8.2 Радни оквир графичке сцене

Задатак 8.2.1. Коришћењем библиотеке *Qt* имплементирати апликацију за исцртавање организационе табле. Сваки запослени има своје име и презиме, као и позицију у компанији која може бити „Шефови” или „Радници”. Графички кориснички интерфејс дизајнирати као на наредној слици.



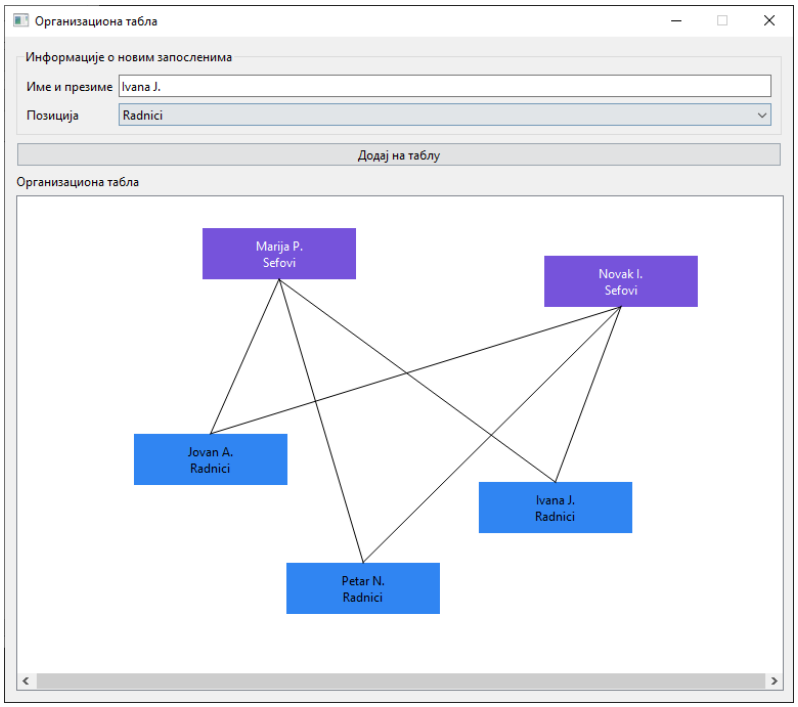
Омогућити следеће операције:

- У оквиру групе поља „Информације о новим запосленима” корисник може да унесе име и презиме нових запослених, као и да одабере позицију у компанији из падајуће листе. Кликом на дугме „Додај на таблу”, потребно је нацртати одговарајући елемент који представља новог запосленог. Сваки елемент приказује информације о имену и презимену запосленог, као и о називу позиције. Додатно, потребно је линијама повезати све шефове са свим радницима.
- У зависности од позиције новог запосленог:

– Омогућити различито приказивање елемената на табли. Шефови се приказују љубичастом позадином и белим текстом, а радници плавом позадином и црним текстом.
– Позиционирати шефове од врха табле, а раднике од „средишњег реда". Сви елементи започињу приказивање од леве ивице и ређају се удесно док има места. Када више нема места, започети приказивање елемената у новом реду.

Решење 8.2.1.

Задатак 8.2.2. Допунити имплементацију задатка 8.2.1 тако да се омогући померање елемената на сцени. Омогућити да се померањем елемената исправно исцртавају све везе између елемената.



Решење 8.2.2.

8.3 Решења задатака

Графички кориснички интерфејси

Решење 8.1.1.

Задатак 8.1.1.

Решење 8.1.2.

Задатак 8.1.2.

Радни оквир графичке сцене

Решење 8.2.1.

Задатак 8.2.1.

Решење 8.2.2.

Задатак 8.2.2.

Глава 9

Архитектура софтвера

Ово поглавље је у припреми!

9.1 Модел-поглед архитектура

9.2 Решења задатака

Модел-поглед архитектура

Глава 10

Комбиновани задаци

Ово поглавље је у припреми!

10.1 C++/STL апликације

10.2 Qt апликације

Задатак 10.2.1. Коришћењем библиотеке *Qt* имплементирати апликацију која израчунава суму разлика узастопних целих бројева. Слика 10.1 илуструје како програм треба да изгледа приликом покретања.

(а) Графички кориснички интерфејс имплементирати тако да изгледа као интерфејс приказан на слици 10.1. Прозор направити тако да се приликом промене његове величине, компоненте аутоматски померају да задрже дати распоред.

(б) Кликом на дугме *Генериши*, прочитати из једнолинијског текстуалног поља поред ознаке *Величина низа* строго позитиван цели број K и генерисати K целих бројева из интервала од -100 до 100 (укључујући оба). Бројеве сачувати у неку погодну структуру података, на пример, вектор B. Бројеве уписати у вишелинијско текстуално поље са размаком, као што је приказано на слици 10.2.

(в) Кликом на дугме *Израчунај*, прочитати из једнолинијског текстуалног поља поред ознаке *Број паралелних израчунавања* строго позитиван цели број H. Апликација затим започиње рад H нити, при чему свака нит добија интервал који се састоји од K/H бројева из вектора B. Претпоставити да је број K дељив бројем H. При томе, нит не сме да добија копије генерисаних елемената, већ индекс почетка и краја. На пример, за генерисане бројеве на слици 10.2:

- Нит 1 добија индексе 0 и 2,
- Нит 2 добија индексе 3 и 5,
- ...

(г) Свака нит треба да израчуна суму разлика за онај интервал одређен индексима које је добила. Разлике се рачунају тако што се од елемента који се посматра одузима наредни елемент из вектора. На пример:

- Нит 1, која је добила индексе 0 и 2 (који дефинишу интервал бројева 75, 99 и 81), рачуна:

$$suma = (75 - 99) + (99 - 81) + (81 - 7) = 68$$

- Нит 2, која је добила индексе 3 и 5 (који дефинишу интервал бројева -7, -97 и -85), рачуна:

$$suma = (7 - (-97)) + (-97 - (-85)) + (-85 - 77) = -70$$

- ...

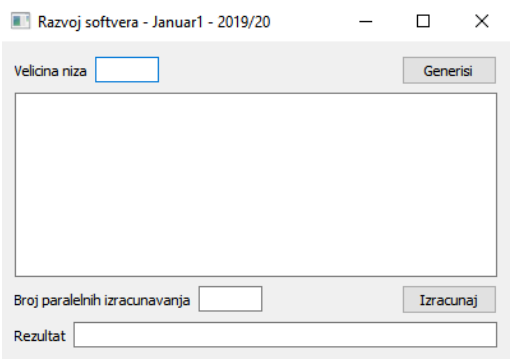
Након што заврши израчунавање, нит шаље израчунату суму главној нити.

Све нити морају да оперишу над истим подацима.

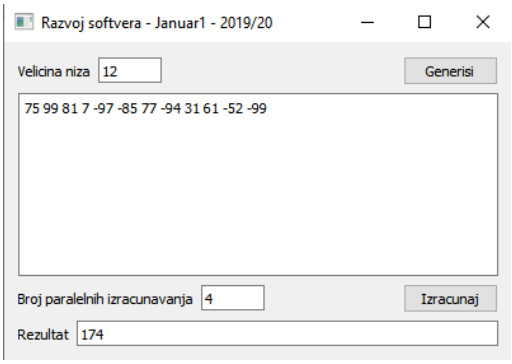
(д) По дохватању сума разлика из осталих нити, главна нит сабира дохваћене суме. Када се последња нит заврши, финални збир се уписује у једнолинијско текстуално поље поред ознаке *Резултат*.

(е) Обезбедити механизме за коректан рад у конкурентном окружењу и водити рачуна о раду са динамичким ресурсима.

Решење 10.2.1.



Слика 10.1: Приказ графичког интерфејса за задатак 10.2.1.



Слика 10.2: Пример покретања апликације за задатак 10.2.1.

Задатак 10.2.2. Линеарна регресија представља метод апроксимације вредности функције помоћу модела чији параметри линеарно зависе. У нашем случају, покушаћемо да апроксимирамо непознату функцију чије узорке имамо дате као податке на основу којих желимо да пронађемо параметре модела (коефицијенти праве) тако да грешка коју наш модел прави буде минимална.

Ако је права p дата као $p : a + b \cdot x$ онда је можемо посматрати као функцију $f(x) = a + b \cdot x$, и дефинисати грешку:

$$MSE(xs, ys) = \sum_{i=0}^{N} (ys_i - f(x_i))^2,$$

где је `xs` вектор који представља податке, а `ys` вектор који садржи тачне вредности за податке.

Коришћењем библиотеке *Qt* имплементирати апликацију која решава преоблем линеарне регресије за податке $xs = [0, 1]$ и $ys = [1, 2]$. У наставку следи опис рада програма.

(а) Графички кориснички интерфејс имплементирати тако да изгледа као интерфејс приказан на слици 10.3. Прозор направити тако да се приликом промене његове величине, компоненте аутоматски померају да задрже дати распоред.

(б) Имплементирати класу `LinearRegression` која као чланске промењљиве поседује вредности `a` и `b` који дефинишу праву.

Имплементирати методе:

- `predict(double x)` - која израчунава вредности $f(x) = a + x \cdot b$
- `QString toQString()` - која врши конверзију објекта у стринг облика `f(x) = a + bx` (узети праве вредности за `a` i `b`)
- `double error(std::vector<double> xs, std::vector<double> ys)` - која израчунава грешку `MSE(xs, ys)`

(в) Имплементирати класу `Solver` која врши оптимизацију проблема линеарне регресије и наслеђује класу `QThread`. Конструктор као аргументе прихвата:

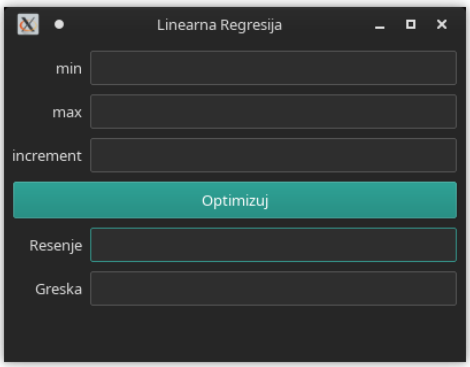
- `double min` - почетак интервала претраге
- `double max` - крај интервалал претраге
- `double increment` - корак инкрементирања приликом претраге
- `double std::vector<double> xs` - вектор улазних података
- `double std::vector<double> ys` - тачне вредности за улазне податке

Нит треба да врши претрагу параметара `a` и `b` линеарне регресије из интервала $[min, max]$ са инкременталним кораком једнаким вредности $increment$. Када год нит пронађе нови минимум, она емитује сигнал кроз који шаље параметре пронађене праве и колика је грешка.

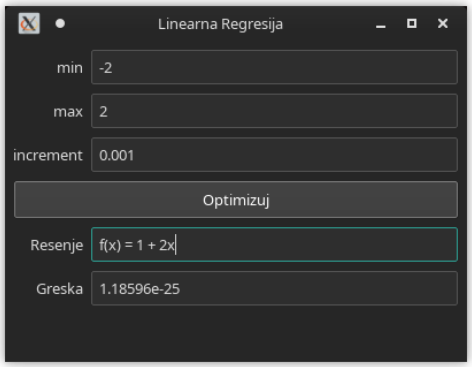
Све нити морају да оперишу над истим подацима.

(г) Кликом на дугме „Оптимизуј”, прочитати параметре `min`, `max` и `increment` и покренути нит која врши оптимизацију проблема линеарне регресије. Када се покрене нит, потребно је онемогућити да корисник може да кликне на дугме док је рачуница у току. Када год нит емитује догађај `update`, главна нит ажурира приказ на текстуалним пољима. Када нит заврши свој рад, поново се омогућава клик на дугме „Оптимизуј”.

Решење 10.2.2.



Слика 10.3: Графички интерфејс за задатак 10.2.2.



Слика 10.4: Пример покретања програма за задатак 10.2.2.

Задатак 10.2.3. Коришћењем библиотеке *Qt* имплементирати апликацију која врши симулацију конкурентне размене ресурса између 5 чворова у потпуној мрежи. Слика 10.5 илуструје како програм треба да изгледа приликом покретања. У наставку следи опис рада програма.

(а) Графички кориснички интерфејс имплементирати тако да изгледа као интерфејс приказан на слици 10.5. Прозор направити тако да се приликом промене његове величине, компоненте аутоматски померају да задрже дати распоред.

(б) Кликом на дугме *Генериши ресурсе*, прочитати из једнолинијског текстуалног поља поред ознаке *Укупно* строго позитиван цели број $У$ и генерисати 5 псеудослучајних строго позитивних целих бројева $ч_1$, ..., $ч_5$ из интервала од 1 до $У/2$ (укључујући оба). Ове вредности представљају ресурсе који се додељују сваком чвору у мрежи (један чвор је представљен једним вишелинијским текстуалним пољем), као што је приказано на слици 10.6. Обезбедити да сума ресурса у чворовима буде једнака укупној вредности, тј. да важи:

$$\sum_{i=1}^{5} ч_i = У. \tag{10.1}$$

Приказати генерисане ресурсе у мрежи вишелинијских текстуалних поља у доњем делу апликације. Корисник може да кликне дугме *Генериши ресурсе* колико год жели пута пре него што започне симулацију и сваки пут је потребно генерисати нове ресурсе у чворовима мреже према упутству изнад.

(в) Кликом на дугме *Започни размене*, потребно је да се започне циклус симулације размена између чворова. Док траје циклус симулације, група контрола *Контроле* на врху апликације треба да буде искључена, као што је приказано на слици 10.7.

Сваки циклус симулације се састоји од 20 итерација. У једној итерацији је потребно започети рад од 0 до 4 нити (нека је генерисани број нити у једној итерацији број $Н$) које врше конкурентну размену ресурса.

(г) Свакој нити у текућој итерацији циклуса је потребно проследити насумични индекс чвора $И$ и насумичну своту $С$ од 1 до $У/Н$. Свака нит у текућој итерацији циклуса треба да изврши размену од $С$ ресурса са свих чворова у мрежи

ка чвору *И*. Када нит заврши са радом, потребно је у апликацији приказати текуће стање ресурса у чворовима у симулацији, као што је приказано на слици 10.7.

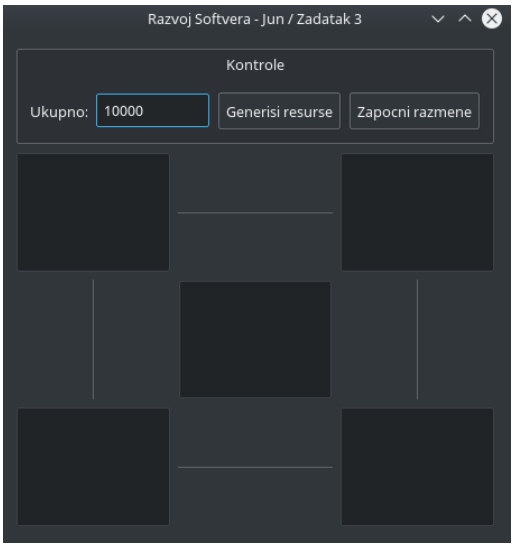
Приметите да је у току симулације важно да у сваком тренутку сума ресурса буде једнака вредности унетој у пољу *Укупно*, тј. да у сваком тренутку буде очувана једнакост из формуле (10.1). Додатно, могуће је да неки чворови имају негативан број ресурса - ово је дозвољено понашање у симулацији.

Све нити морају да оперишу над истим подацима.

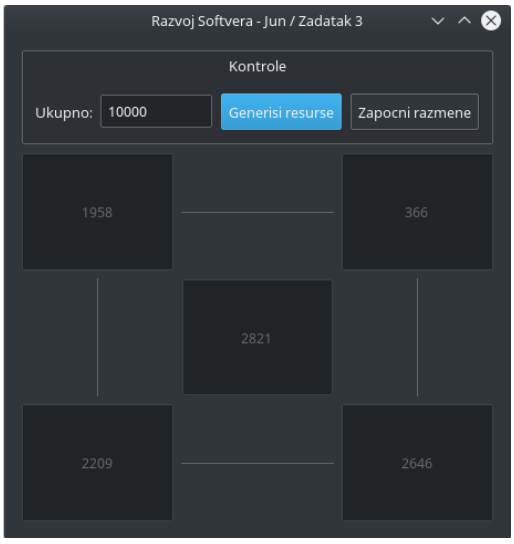
(д) Када се заврши један циклус симулације, потребно је да се поново омогући група контрола *Контроле* на врху апликације, као што је приказано на слици 10.8. Наравно, на крају симулације, мора да буде очувана једнакост из формуле (10.1).

(е) Обезбедити механизме за синхронизацију рада у конкурентном окружењу и водити рачуна о раду са динамичким ресурсима.

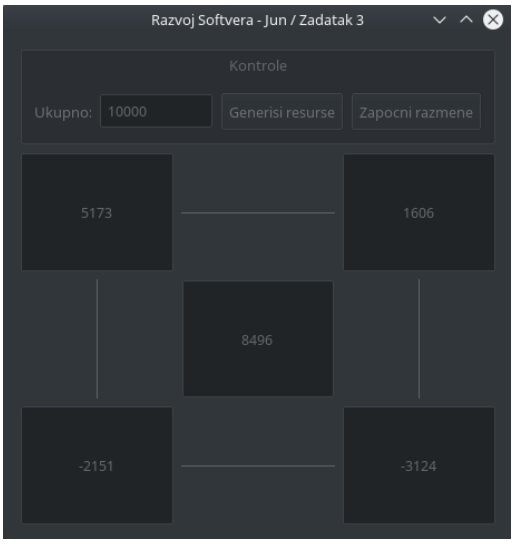
Решење 10.2.3.



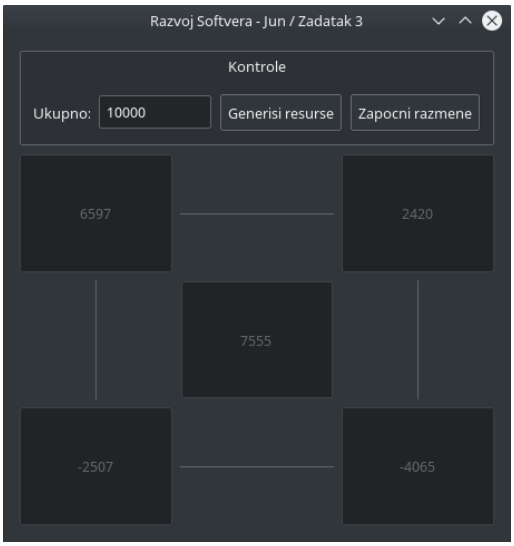
Слика 10.5: Приказ графичког интерфејса за задатак 10.2.3.



Слика 10.6: Пример распоређивања ресурса након клика на дугме *Генериши ресурсе* за задатак 10.2.3.



Слика 10.7: Пример размене ресурса након клика на дугме *Започни размене* (тј. током трајања симулације) за задатак 10.2.3.



Слика 10.8: Пример завршетка симулације размене за задатак 10.2.3.

Задатак 10.2.4. Коришћењем библиотеке *Qt* имплементирати апликацију која симулира конкурентне донације породицама у једном граду ради побољшања стандарда живота. Слика 10.9 илуструје како програм треба да изгледа приликом покретања. У наставку следи опис рада програма.

(а) Графички кориснички интерфејс имплементирати тако да изгледа као интерфејс приказан на слици 10.9. Елементе прозора распоредити тако да се приликом промене његове величине, компоненте аутоматски померају да задрже приказани распоред.

(б) При покретању програма генерисати податке о стандарду породица у граду. Град је представљен матрицом димензија 5×5 са насумичним подацима у интервалу $[-50, 50]$. Редови у матрици представљају улице у граду, а свако поље у реду представља стандард једне породице у тој улици. Затим, на основу ове матрице попунити `QTableWidget` испод лабеле „Приказ области”, као на слици 10.9. (Ако желите да промените ширину колона у `QTableWidget`-у, промените вредност атрибута `horizontalHeaderDefaultSectionSize` у *Design* погледу (на пример, на 50).)

(в) Симулација започиње кликом на дугме „Започни донације”. Покренути за сваку улицу по једну нит (која представља донатора за ту улицу), при чему се нити прослеђује индекс улице и насумична вредност донације у интервалу $[5, 15]$.

(г) Након што је покренута, нит извршава наредне кораке:

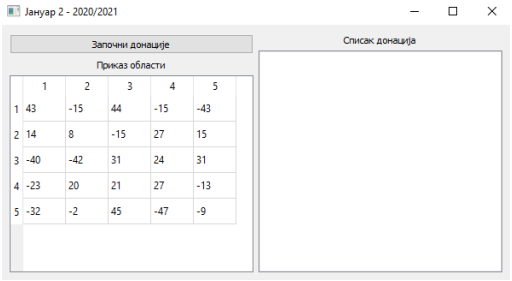
- Успављује се на насумичан број секунди X, где је $X \in [1, 5]$.
- Увећава стандард свим породицама у улици чији индекс је прослеђен нити приликом конструкције.

Све нити морају да оперишу над истим подацима.

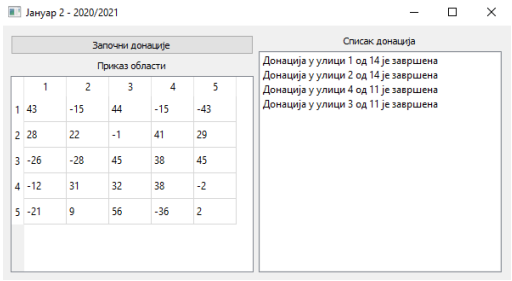
(д) Када једна нит заврши са донацијама у додељеној улици, обавестити главну нит да је дошло до донације у одговарајућој улици. Затим, у `QListWidget`-у „Списак донација” уписати да је донација завршена у тој улици, као и колика је била донација, као на слици 10.10. У исто време, проверити да ли постоји још нека породица у тој улици чији је стандард непозитиван. Уколико таква породица постоји, покренути још једну нит за исту улицу са насумичном донацијом. Симулација се завршава када све породице имају позитиван стандард, као на слици 10.11.

(ђ) Осигурати се да не долази до проблема у конкурентном окружењу и водити рачуна о раду са динамичким ресурсима.

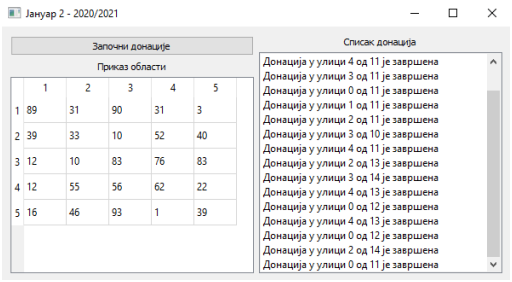
Решење 10.2.4.



Слика 10.9: Приказ графичког интерфејса за задатак 10.2.4.



Слика 10.10: Приказ током конкурентних донација када и даље постоје породице са ниским стандардом за задатак 10.2.4.



Слика 10.11: Приказ након што све породице имају позитиван стандард и донације су завршене за задатак 10.2.4.

Задатак 10.2.5. Коришћењем библиотеке *Qt* имплементирати апликацију која симулира грамзиву изградњу путева између градова у једној држави. Слика 10.12 илуструје како програм треба да изгледа приликом покретања. У наставку следи опис рада програма.

(а) Графички кориснички интерфејс имплементирати тако да изгледа као интерфејс приказан на слици 10.12. Елементе прозора распоредити тако да се приликом промене његове величине, компоненте аутоматски померају да задрже приказани распоред.

(б) Кликом на дугме „Учитај градове”:

- Учитати из текстуалне датотеке податке о градовима. Текстуална датотека је записана у JSON формату (видети пример датотеке испод).
- Сваки град представити као објекат класе `City`. Објекти ове класе се састоје од наредних атрибута: (1) назив града (`QString`), (2) x-позиција града (`double`), y-позиција града (`double`) и индикатор да ли је град посећен (`bool`). Обезбедити наредни јавни интерфејс ове класе:
  - Метод `void fromQVariant(const QVariant &variant)` десеријализује податке о једном граду из прослеђеног аргумента (видети датотеку `cities.json` у поставкама). Аргумент садржи податке о називу, x-позицији и y-позицији града. Посећеност града поставити на `false`.
  - Метод `QString toQString() const` израчунава текст са подацима о граду у формату који је приказан на слици 10.13.
- Приказати градове у `QListWidget` контроли „Градови” као на слици 10.13.

(в) Симулација започиње кликом на дугме „Изгради путеве” и састоји се од неколико итерација. У свакој итерацији симулације, покреће се по једна нит која добија редни број града. Прва итерација почиње градом са редним бројем 0. Када се за неки град покрене нит, означити да је тај град посећен.

(г) Након што је покренута, нит редом:

- Успављује се на 500 милисекунди.
- Проналази који град је најближи граду чији је редни број прослеђен приликом конструкције нити. Не узимати у обзир градове који су већ посећени.

Све нити морају да конкурентно оперишу над истим подацима.

(д) Када нит заврши са проналажењем најближег града, обавестити главну нит који је то град који је најближи тренутном граду. Затим, у `QTextEdit` контроли „Путеви” уписати информацију да се изградио нови пут између тих градова, као на слици 10.14. Затим, проверити да ли су сви градови посећени, и:

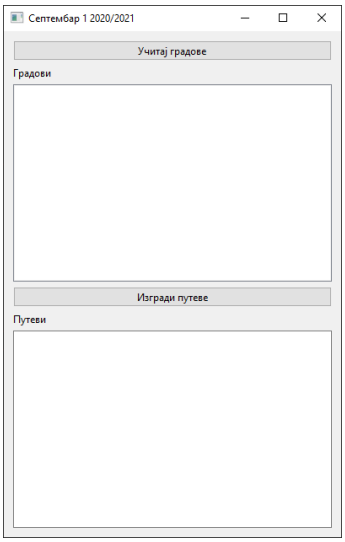
- Уколико нису, покренути нову нит која ће наставити изградњу следећег пута поступком описаним изнад. Водити рачуна да се изградња следећег пута наставља од последњег посећеног града.
- Уколико јесу, у истој контроли исписати информацију да су сви путеви изграђени, као на слици 10.14.

(ђ) Осигурати се да не долази до проблема у конкурентном окружењу и водити рачуна о раду са динамичким ресурсима.

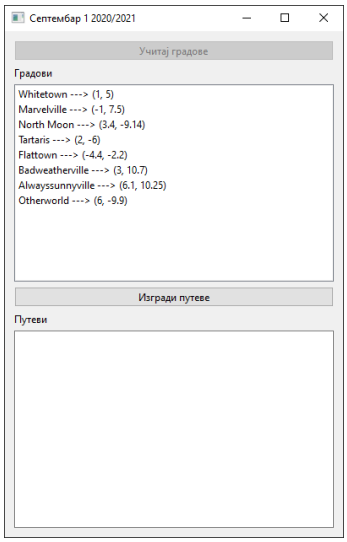
Код 10.1: Пример датотеке за десеријализацију.

```
[
  {
    "name": "Whitetown",
    "x": 1.0,
    "y": 5.0
  },
  {
    "name": "Marvelville",
    "x": -1.0,
    "y": 7.5
  },
  {
    "name": "North Moon",
    "x": 3.4,
    "y": -9.14
  },
  {
    "name": "Tartaris",
    "x": 2.0,
    "y": -6.0
  },
  {
    "name": "Flattown",
    "x": -4.4,
    "y": -2.2
  },
  {
    "name": "Badweatherville",
    "x": 3.0,
    "y": 10.7
  },
  {
    "name": "Alwayssunnyville",
    "x": 6.1,
    "y": 10.25
  },
  {
    "name": "Otherworld",
    "x": 6.0,
    "y": -9.9
  }
]
```

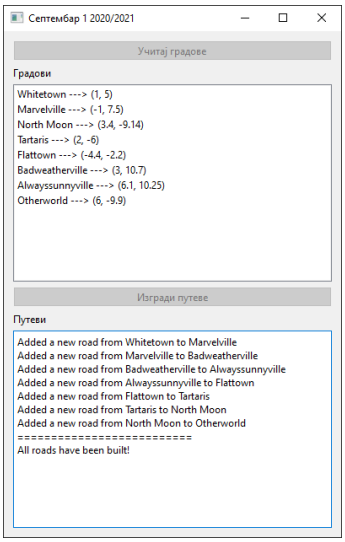
Решење 10.2.5.



Слика 10.12: Приказ графичког интерфејса за задатак 10.2.5.



Слика 10.13: Приказ након учитавања списка градова за задатак 10.2.5.



Слика 10.14: Приказ након што се завршила симулација изградње путева за задатак 10.2.5.

Задатак 10.2.6. Коришћењем библиотеке *Qt* имплементирати апликацију која симулира конкурентну куповину намирница за дате рецепте. Слика 10.15 илуструје како програм треба да изгледа приликом покретања. У наставку следи опис рада програма.

(а) Графички кориснички интерфејс имплементирати тако да изгледа као интерфејс приказан на слици 10.15. Елементе прозора распоредити тако да се приликом промене његове величине, компоненте аутоматски померају да задрже приказани распоред.

(б) Кликом на дугме „Учитај састојке”:

- Учитати из текстуалне датотеке податке о рецептима и њиховим састојцима. Текстуална датотека је записана у JSON формату (видети пример датотеке испод).
- Сваки рецепт представити као објекат класе `Recipe`. Објекти ове класе се састоје од наредних атрибута: (1) назив рецепта (`QString`), (2) списак састојака неопходних за прављење рецепта (`QVector<QString>`) и (3) списак тренутно купљених састојака (`QVector<QString>`). Обезбедити наредни јавни интерфејс ове класе:
  - Метод `void fromQVariant(const QVariant &variant)` десеријализује податке о једном рецепту из прослеђеног аргумента (видети датотеку `ingredients.json` у поставкама). Аргумент садржи податке о називу и неопходним састојцима. Списак тренутно купљених састојака поставити на празан.
  - Метод `QString toQString() const` израчунава текст са подацима о рецепту у формату који је приказан на сликама 10.16 и 10.17. Текст

се састоји од 2 линије. У првој линији се налази назив рецепта. У другој линији се налази онолико карактера `'X'` или `'O'` колико је састојака неопходно за прављење рецепта. Уколико састојак није купљен, приказати карактер `'X'`, а у супротном приказати карактер `'O'`.

- Приказати градове у `QTextEdit` контроли као на слици 10.16.

(в) Симулација започиње уношењем строго позитивног броја у поље „Број купаца”, а затим, кликом на дугме „Започни куповину”. За сваког купца покренути по једну нит.

(г) Након што је покренута, нит редом понавља наредне кораке насумичан број (највише 20) пута:

- Успављује се на 1 секунду.
- Насумично бира један састојак од свих састојака који су се јављали у рецептима.

Све нити морају да конкурентно оперишу над истим подацима.

(д) Сваки пут када нит одабере састојак, обавестити главну нит који је то састојак одабран. Затим, проћи кроз све рецепте и додати тај састојак као купљен. Коначно, у `QTextEdit` контроли приказати ново стање рецепата, као на слици 10.17.

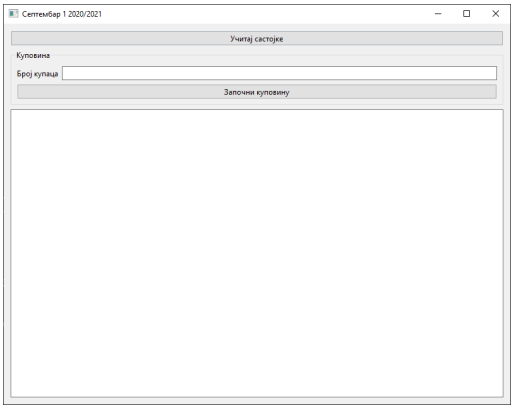
(ђ) Осигурати се да не долази до проблема у конкурентном окружењу и водити рачуна о раду са динамичким ресурсима.

Код 10.2: Пример датотеке за десеријализацију.

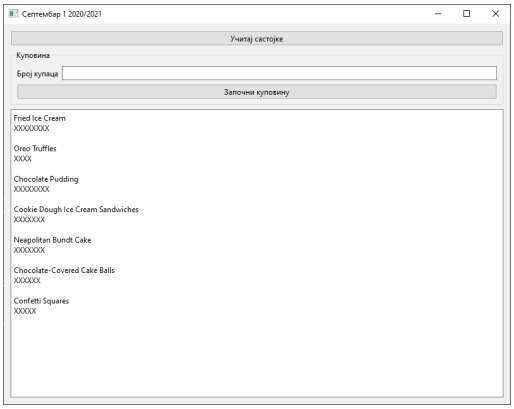
```
[
    {
        "title": "Fried Ice Cream",
        "ingredients": [
            "ice cream",
            "butter",
            "corn cereal",
            "cinnamon",
            "sugar",
            "whipped cream",
            "sprinkles",
            "cherries"
        ]
    },
    {
        "title": "Oreo Truffles",
        "ingredients": [
            "oreos",
            "cream cheese",
            "vanilla extract",
            "chocolate"
        ]
    },
    {
        "title": "Chocolate Pudding",
        "ingredients": [
```

```
            "sugar",
            "cocoa powder",
            "milk",
            "eggs",
            "chocolate",
            "butter",
            "vanilla extract",
            "whipped cream"
        ]
    },
    {
        "title": "Cookie Dough Ice Cream Sandwiches",
        "ingredients": [
            "butter",
            "sugar",
            "milk",
            "vanilla extract",
            "flour",
            "chocolate",
            "ice cream"
        ]
    },
    {
        "title": "Neapolitan Bundt Cake",
        "ingredients": [
            "cake mix",
            "water",
            "butter",
            "eggs",
            "strawberry syrup",
            "vanilla extract",
            "chocolate syrup"
        ]
    },
    {
        "title": "Chocolate-Covered Cake Balls",
        "ingredients": [
            "cake mix",
            "butter",
            "sugar",
            "vanilla extract",
            "chocolate",
            "sprinkles"
        ]
    },
    {
        "title": "Confetti Squares",
        "ingredients": [
            "chocolate",
            "butter",
            "vanilla extract",
            "peanuts",
            "rainbow marshmallows"
        ]
    }
]
```

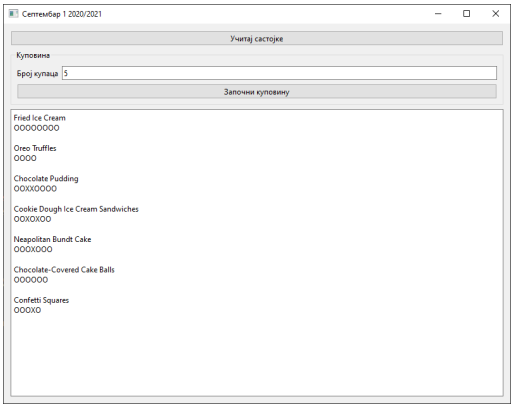
Решење 10.2.6.



Слика 10.15: Приказ графичког интерфејса за задатак 10.2.6.



Слика 10.16: Приказ након учитавања рецепата за задатак 10.2.6.



Слика 10.17: Приказ након што се завршила симулација куповине намирница за задатак 10.2.6.

Задатак 10.2.7. Коришћењем библиотеке *Qt* имплементирати апликацију која симулира конкурентан проток флуида из различитих извора у заједнички резервоар. Слика 10.18 илуструје како програм треба да изгледа приликом покретања. У наставку следи опис рада програма.

(а) Графички кориснички интерфејс имплементирати тако да изгледа као интерфејс приказан на слици 10.18. Прозор направити тако да се приликом промене његове величине, компоненте аутоматски померају да задрже дати распоред. Поља „Укупна запремина извора”, „Текућа запремина резервоара” и „Губитак” онемогућити за измену од стране корисника.

(б) Имплементирати класу `Source`:

- Приватни атрибути ове класе су: (1) назив извора и (2) почетна целобројна запремина флуида у том извору.

- Јавни метод `fromQVariant` прихвата објекат класе `QVariant` и десеријализује вредности из њега.
- Јавни метод `toQString` израчунава ниску која садржи информације о атрибутима објекта у формату који је приказан на слици 10.19. Уколико је извор исцрпљен (тј. запремина је нула), приказати ниску као на сликама 10.21 и 10.22.

Дозвољено је додавати и друге чланице по потреби.

(в) Кликом на дугме „Попуни изворе”:

- Учитати из текстуалне датотеке податке о изворима. Текстуална датотека је записана у XML формату (видети пример датотеке испод).
- На основу десеријализованих вредности попунити `QListWidget` „Извори”, као на слици 10.19.
- Попунити поље „Укупна запремина извора” тако да садржи суму генерисаних вредности извора, као на слици 10.19.

Пре започињања симулације, корисник може више пута кликнути на ово дугме и сваки пут се изнова учитавају нови извори.

(г) Симулација започиње уношењем позитивне запремине резервоара у поље „Укупна запремина резервоара” и кликом на дугме „Започни трансфер”. Прво поставити иницијалну вредност поља „Текућа запремина резервоара” и „Губитак” на 0. Затим, покренути за сваки извор по једну нит.

(д) Након што је покренута, нит понавља наредне кораке:

- Успављује се на насумичан број милисекунди X из скупа $500, 600, \ldots, 1000$.
- Истиче из насумичног извора насумичну запремину флуида V, где је $V \in [100, 200]$, али само ако је насумично одабран извор непразан. У противном, враћа се на претходни корак. Ако је у извору остало мање запремине него што је V, онда нит истиче тачно онолико запремине колико је преостало у извору (чиме се тај извор исцрпљује) и завршава са радом.

Све нити морају да оперишу над истим подацима.

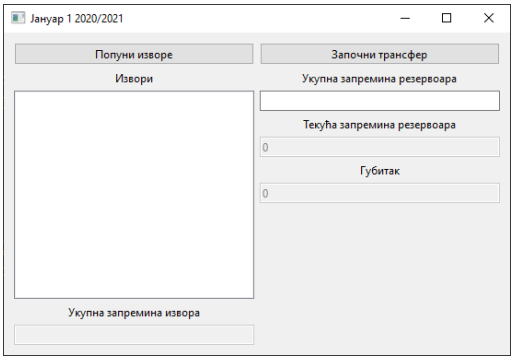
(ђ) Када једна нит истекне неку запремину V из неког извора, обавестити главну нит да је дошло до истицања. Затим, у пољу „Текућа запремина резервоара” додати количину флуида V која је истекла, као на слици 10.20. У случају да додата количина флуида превазилази укупну запремину у пољу „Укупна запремина резервоара”, уписати сав вишак у поље „Губитак”, као на слици 10.21. У сваком случају, потребно је ажурирати приказ свих извора у `QListWidget`-у, тако да у сваком тренутку осликавају текућу преосталу запремину сваког извора (или да су неки од извора истекли). Симулација се завршава када се сви извори исцрпе. На крају симулације се очекује да је укупна запремина извора била једнака збиру запремина резервоара и губитака, као на слици 10.22.

(е) Осигурати се да не долази до проблема у конкурентном окружењу и водити рачуна о раду са динамичким ресурсима.

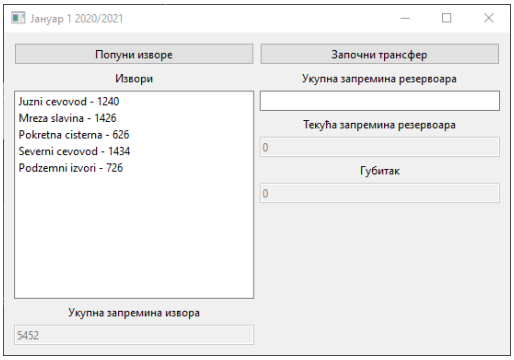
Код 10.3: Пример датотеке за десеријализацију.

```
<?xml version="1.0" encoding="UTF-8"?>
<sources type="QVariantList">
    <source type="QVariantMap">
        <volume type="uint">1240</volume>
        <name type="QString">Juzni cevovod</name>
    </source>
    <source type="QVariantMap">
        <volume type="uint">1426</volume>
        <name type="QString">Mreza slavina</name>
    </source>
    <source type="QVariantMap">
        <volume type="uint">626</volume>
        <name type="QString">Pokretna cisterna</name>
    </source>
    <source type="QVariantMap">
        <volume type="uint">1434</volume>
        <name type="QString">Severni cevovod</name>
    </source>
    <source type="QVariantMap">
        <volume type="uint">726</volume>
        <name type="QString">Podzemni izvori</name>
    </source>
</sources>
```

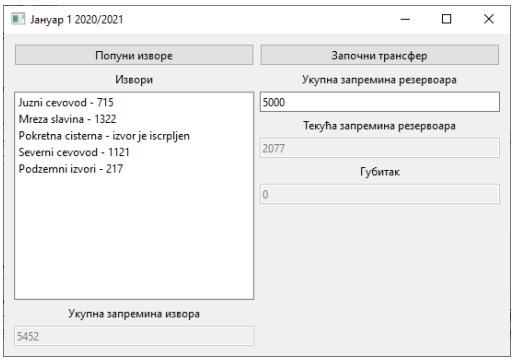
Решење 10.2.7.



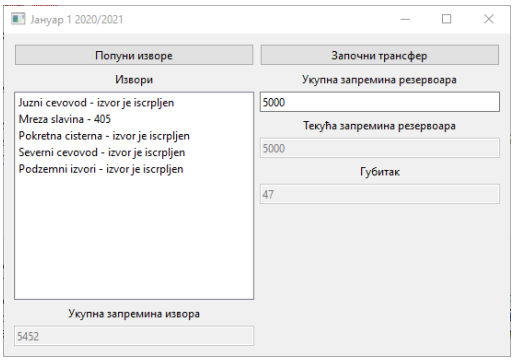
Слика 10.18: Приказ графичког интерфејса за задатак 10.2.7.



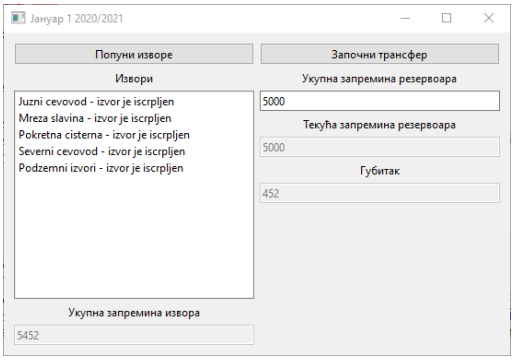
Слика 10.19: Приказ након клика на дугме „Попуни изворе” за задатак 10.2.7.



Слика 10.20: Приказ након уноса укупне запремине резервоара и клика на дугме „Започни трансфер” за задатак 10.2.7. У овом тренутку резервоар још увек није попуњен.



Слика 10.21: Приказ током трансфера флуида након што се резервоар напунио за задатак 10.2.7. Сва остала истицања из извора се бележе као губици.



Слика 10.22: Приказ када су се сви извори исцрпли за задатак 10.2.7. Приметите да је збир укупне запремине резервоара и запремине губитака једнак укупној запремини извора на почетку симулације.

10.3 Решења задатака

C++/STL апликације

Qt апликације

Решење 10.2.1.

Задатак 10.2.1.

Решење 10.2.2.

Задатак 10.2.2.

Решење 10.2.3.

Задатак 10.2.3.

Решење 10.2.4.

Задатак 10.2.4.

Решење 10.2.5.

Задатак 10.2.5.

Решење 10.2.6.

Задатак 10.2.6.

Решење 10.2.7.

Задатак 10.2.7.