Treća laboratorijska vježba iz OOUP

1. Tvornice (30% bodova)
Ova vježba razmatra oblikovanje tvornica koje ne ovise o konkretnim tipovima. Takve tvornice ćemo nazivati generičkim tvornicama, iako njihova izvedba ne mora biti povezana s generičkim programiranjem. Glavna prednost generičkih tvornica jest u tome što ih nije potrebno mijenjati kad želimo stvarati nove vrste komponenata. U svim vježbama ćemo razmatrati primjer s kućnim ljubimcima kojeg smo razmatrali i u prvoj laboratorijskoj vježbi. Htjet ćemo omogućiti stvaranje novih vrsta ljubimaca bez potrebe za mijenjanjem kôda koji inicira stvaranje. U C-u, ključni poziv bi bio:

```
struct Animal* p1=myfactory("cat", "Ofelija");
```
Zadatak generičke tvornice je asocirati simbolički identifikator `cat` s konkretnim podatkovnim tipom `struct Cat`. Na žalost, izvedbe generičkih tvornica čvrsto su vezane uz izvedbene detalje programskih jezika pa ćemo stoga pojedine jezike morati razmatrati ponaosob.

Obavezni dio vježbe uključuje izvedbu u C-u (1.1), te *barem jednu* od izvedbi u Pythonu (1.2), C++-u (1.3) i Javi (1.4). Naravno, uvijek su dobrodošli i studenti koji riješe više od obaveznog dijela.

1.1. Prvo ćemo pogledati kako bismo generičku tvornicu izveli u C-u. S obzirom da C ima vrlo ograničene mogućnosti introspekcije, jedini donekle portabilni način da taj cilj postignemo jest zapakirati konkretne objekte u dinamičke biblioteke (.dll,.so). Međutim, prije nego što uronimo u detalje, definirajmo naš cilj sljedećim ispitnim izvornim kôdom.

```
#include "myfactory.h"

#include <stdio.h>
#include <stdlib.h>

typedef char const* (*PTRFUN)();

struct Animal{
  PTRFUN* vtable;
  // vtable entries:
  // 0: char const* name(void* this);
  // 1: char const* greet();
  // 2: char const* menu();
};

// parrots and tigers defined in respective dynamic libraries

// animalPrintGreeting and animalPrintMenu similar as in lab 1

int main(int argc, char *argv[]){
  for (int i=0; i<argc; ++i){
    struct Animal* p=(struct Animal*)myfactory(argv[i], "Modrobradi");
    if (!p){
      printf("Creation of plug-in object %s failed.\n", argv[i]);
      continue;
    }

    animalPrintGreeting(p);
    animalPrintMenu(p);
    free(p);
  }
```

```
}
```

Zadatci vježbe su sljedeći:

1. Implementirati funkciju

```
void* myfactory(char const* libname, char const* ctorarg);
```

   Funkcija treba i) otvoriti biblioteku zadanu prvim argumentom (`libname`), ii) učitati iz nje funkciju `create`, iii) pozvati create sa svojim drugim argumentom (`ctorarg`), te iv) dobiveni pokazivač vratiti pozivatelju. Zbog jednostavnosti, funkcija treba pretpostaviti da se tražena biblioteka nalazi u tekućem kazalu te dekorirati ime biblioteke tekućim kazalom './' i standardnom ekstenzijom `.so` (UNIX) ili `.dll` (Windows). Prije pisanja kôda preporučamo proučiti funkcije `dlopen` i `dlsym` (UNIX), odnosno `LoadLibrary` i `GetProcAddress` (Windows). Također, provjerite da li u potpunosti razumijete **značenje** deklaracije tipa `PTRFUN`. Neka prototip funkcije bude u `myfactory.h`, a izvedba u `myfactory.c`.

3. Implementirati funkcije `animalPrintGreeting` i `animalPrintMenu`. Implementacija će biti vrlo slična kao i u vježbi 1. Međutim, pripazite da sada do imena ljubimca dolazimo pozivom metode `name` (indeks 0 virtualne tablice). Sada bi se glavni program trebao moći prevesti (na UNIXU: `gcc main.c myfactory.c -ldl`). Pokretanje dobivene izvršne datoteke treba rezultirati porukom `Creation of plug-in objects failed.`.
4. Implementirati dinamičke biblioteke `parrot.so` i `tiger.so` (odnosno `parrot.dll` i `tiger.dll` na Windowsima). Izvedba će ponovo biti vrlo slična izvedbama odgovarajućih funkcija u prvoj laboratorijskoj vježbi. Izvorni kod treba i) definirati konkretni tip ljubimca strukturom koja će osim pokazivača na virtualnu tablicu imati i pokazivač na ime. ii) implementirati funkcije ("metode"): `name`, `greet` i `menu`. iii) definirati virtualnu tablicu, te iv) definirati funkciju za stvaranje novih objekata na gomili s prototipom `void* create(char const* name);` Izvorni kôd dinamičkih biblioteka smjestite u datoteke `tiger.c` i `parrot.c` te ih prevedite (na UNIXU, za `parrot.c`: `gcc -shared -fPIC parrot.c -o parrot.so`)
5. Ponovo testirati glavni program i otkloniti eventualne nedostatke. Sljedeća preporuka bi vam mogle pomoći da ne lutate previše. Neka se sve datoteke izvornog koda (`main.c`, `myfactory.c`, `myfactory.h`, `parrot.c`, `tiger.c`) nalaze u istom kazalu. Prevođenje i testiranje tada možete provesti pasteanjem sljedećeg bloka naredbi u terminal (naravno, možete napraviti i skriptu).

```
gcc main.c myfactory.c -ldl
gcc -shared -fPIC tiger.c -o tiger.so
gcc -shared -fPIC parrot.c -o parrot.so
./a.out
```

10. Predložiti rješenje koje bi klijentima biblioteke omogućilo fleksibilnost pri alociranju memorijskog prostora za novi objekt. Vaše rješenje mora omogućiti stvaranje objekata na stogu odnosno unutar odvojeno alociranog memorijskog prostora, kao što je bilo traženo i u prvom zadatku prve vježbe. Uputa 1: generičku tvornicu potrebno je prekrojiti. Uputa 2: biblioteke trebaju definirati dvije dodatne funkcijer; prva funkcija (možete je zvati sizeof) vraća veličinu objekata u bajtovima; druga funkcija (možete je zvati construct) inicijalizira objekt u memorijskom prostoru kojeg zadaje pozivatelj. Uputa 3: memoriju na stogu možete zauzeti lokalnim poljem znakova ili pozivom funkcije alloca (ili _malloca); na Windowsima funkcionira samo ova druga opcija.

1.2. Sada ćemo razmotriti izvedbu generičkih tvornica u Pythonu. Kako vježba ne bi bila prelaka, razmotrit ćemo malo teži problem. Potrebno je napisati program koji će napraviti sljedeće: i) iz svake

datoteke izvornog koda u kazalu plugins instancirati po jednog ljubimca, te ii) sve instancirane ljubimce predstaviti načinom glasanja i omiljenim obrokom. Ispitni program bi izgledao ovako:

```
def test():
  pets=[]
  # obiđi svaku datoteku kazala plugins
  for mymodule in os.listdir('plugins'):
    moduleName, moduleExt = os.path.splitext(mymodule)
    # ako se radi o datoteci s Pythonskim kodom ...
    if moduleExt=='.py':
      # instanciraj ljubimca ...
      ljubimac=myfactory(moduleName)('Ljubimac '+str(len(pets)))
      # ... i dodaj ga u listu ljubimaca
      pets.append(ljubimac)

  # ispiši ljubimce
  for pet in pets:
    printGreeting(pet)
    printMenu(pet)
```

Ispis ispitnog programa treba izgledati ovako:

```
Ljubimac 0 pozdravlja: Sto mu gromova!
Ljubimac 0 voli brazilske orahe.
Ljubimac 1 pozdravlja: Mijau!
Ljubimac 1 voli mlako mlijeko.
```

Upute:

1. Ljubimce predstavi razredima `tiger` i `parrot` koje ćeš smjestiti u istoimene module kazala plugins. Kao i do sada, ljubimci trebaju definirati konstruktor koji u argumentu prima ime ljubimca, te metode `name`, `greet` i `menu`. Pojedine ljubimce možeš opisati s 9 redaka Pythona.
2. Funkcija `myfactory` treba koristiti funkciju `import_module` modula `importlib`, te ugrađenu funkciju `getattr`. Za definiciju funkcije dovoljna su 3 retka uključujući i zaglavlje.

1.3. U C++-u možemo dinamički stvarati komponente iz dinamičkih biblioteka, ali tim se ovdje nećemo baviti jer smo to već obradili u odjeljku koji se odnosi na programski jezik C. Ovdje ćemo međutim pokazati jednu drugu mogućnost, a ta je da tvornica stvara objekte konkretnih razreda koji su prevedeni i ugrađeni u izvršnu datoteku, ali bez da bude ovisna o njima. Ključ za ostvarivanje te funkcionalnosti je mogućnost da lokalne varijable u dosegu datoteke inicijaliziramo povratnom vrijednošću funkcije za koju će se prevoditelj pobrinuti da bude pozvana na samom početku izvđenja programa, prije ulaska u funkciju `main()`. Naš zadatak će biti analogan zadatku u Pythonu: proiterirati svim ukompajliranim ljubimcima te svakog od njih predstaviti načinom glasanja i omiljenim obrokom. U izvedbi valja koristiti sljedeće pretpostavke.

1. Svi ljubimci izvode razred Animal:

```
class Animal{
public:
  virtual char const* name()=0;
  virtual char const* greet()=0;
  virtual char const* menu()=0;
};
```

8. Svaki konkretni tip treba izvesti u zasebnoj komponenti. Konkretni tipovi trebaju definirati:
   - konstruktor čiji argument je ime ljubimca.
   - statičku funkciju koja konstruira nove objekte, npr:

```
    static void* myCreator(const std::string& arg){
      return new Parrot(arg);
    }
```

- statičku varijablu čija incijalizacijska funkcija registrira gornju funkciju u tvornici:

```
    static int hreg=myfactory::instance().registerCreator("parrot",
myCreator);
```

- Ova ideja opisana je u knjizi Modern C++ Design Andreia Alexandrescua. Međutim, u **standardu** postoji odredba (3.6.2) prema kojoj bi prevoditelj bio slobodan odgoditi incijalizaciju lokalnih varijabli u dosegu datoteke sve do trenutka kad se pozove prva funkcija iz te datoteke. S obzirom na to da se funkcije komponenata s konkretnim tipovima ne pozivaju izravno, to bi značilo da bi se naša inicijalizacija mogla ne obaviti nikad. Ipak, dominantno mišljenje eksperata je da se ta odredba odnosi isključivo na dinamičke biblioteke, te da sve postojeće implementacije pozivaju inicijalizaciju svih statički linkanih modula (a to bi bio upravo naš slučaj) prije poziva funkcije main(). Detalje možete čitati u **raspravi** na comp.lang.c++.

Definicije razreda i pojedinih metoda konkretnih razreda mogu biti u istoj datoteci.

9. Neka tvornica bude jedinstveni objekt (singleton) koji:
   - održava asocijativno polje `creators_` koje povezuje simbolička imena razreda s pokazivačima na odgovarajuće konstrukcijske funkcije
   - konkretnim razredima omogućava registriranje konstrukcijske funkcije metodom `registerCreator`
   - klijentima omogućava kreiranje novih objekata preko simboličkog imena metodom `create`
   - klijentima omogućava dohvaćanje vektora sa simboličkim imenima svih konkrentih objekata

```
10.    // myfactory.hpp
11.    class myfactory{
12.    public:
13.      typedef void* (*pFunCreator)(const std::string&);
14.      typedef std::map<std::string, pFunCreator> MyMap;
15.    public:
16.      static myfactory &instance();
17.    public:
18.      int registerCreator(const std::string &id, pFunCreator pfn);
19.    public:
20.      void *create(const std::string &id, const std::string &arg);
21.      std::vector<std::string> getIds();
22.    private:
23.      myfactory();
24.      ~myfactory();
25.      myfactory(const myfactory&);
26.      MyMap creators_;
27.    };
28.
29.    // myfactory.cpp
30.    myfactory& myfactory::instance(){
31.      static myfactory theFactory;
32.      return theFactory;
33.    }
34.
35.   // some implementations missing...
```

Zbog jednostavnosti, predložili smo da funkcija create vraća `void*` pa klijenti tvornice moraju koristiti ružne pretvorbe pokazivača (castove). Bolje rješenje je moguće izvesti upotrebom predložaka, a to zainteresiranim studentima ostavljamo za neobaveznu vježbu. Glavni program može izgledati ovako:

```
int main(void){
  myfactory& fact(myfactory::instance());
  std::vector<std::string> vecIds=fact.getIds();
  for (int i=0; i<vecIds.size(); ++i){
    std::ostringstream oss;
    oss <<"Ljubimac " <<i;
    Animal* pa=(Animal*) fact.create(vecIds[i], oss.str());
    printGreeting(*pa);
    printMenu(*pa);
    delete pa;
  }
}
```

Preporuka: sve datoteke (`main.cpp`, `animal.hpp`, `myfactory.cpp`, `myfactory.hpp`, `parrot.cpp`, `tiger.cpp`) smjestite u isto kazalo. U tom slučaju prevođenje i pokretanje dobivamo jednostavno s:

```
g++ *cpp; ./a.out
```

Napomena za one koji bi ovako nešto htjeli isprogramirati u C-u. Na žalost, C-ov standard ne predviđa mogućnost pozivanja korisničkih funkcija prije funkcije `main()`. Međutim, nema nikakvih načelnih prepreka da se takva funkcionalnost ostvari, pa većina prevoditelja za to nudi nestandardna proširenja (gcc: constructor attribute, msvc: pragma data_seg autostart).

1.4. Na kraju pogledajmo kako bismo generičku tvornicu mogli ostvariti u Javi. Pretpostavimo da je definiran apstraktni razred `hr.fer.zemris.ooup.lab2.model.Animal` s apstraktnim metodama:

```
	public abstract String name();
	public abstract String greet();
	public abstract String menu();
```

i konkretnim metodama:

```
	public void animalPrintGreeting() {
		...
	}

	public void animalPrintMenu() {
		...
	}
```

Pretpostavimo da će korisnik konkretne implementacije razreda `Animal` stavljati u paket `hr.fer.zemris.ooup.lab2.model.plugins` i da će svaka životinja imati konstruktor koji prima jedan string (njezino ime). Primjerice:

```
public class Parrot extends Animal {

	private String animalName;

	public Parrot(String name) {
		...
	}
```

```
}
```
Potrebno je napisati razred `AnimalFactory` koji ima statičku metodu `newInstance` kojom stvara proizvoljnu životinju ali na način da ne postoji *compile-time* ovisnost, odnosno da se sve razrješava tijekom izvođenja programa:

```
public class AnimalFactory {

	public static Animal newInstance(String animalKind, String name) {
		...
		return ...;
	}

}
```
Uz pretpostavku da su `.class` datoteke za navedene životinje dostupne u *classpath*-u Javinom virtualnom stroju, dinamičko učitavanje novih razreda moguće je obaviti statičkom metodom `forName` razreda `Class`:

```
Class<Animal> clazz = null;
clazz =
(Class<Animal>)Class.forName("hr.fer.zemris.ooup.lab2.model.plugins."+animalK
ind);
```
Jednom kad imamo referencu na razred, stvaranje novog primjerka razreda preko defaultnog konstruktora moguće je obaviti pozivom metode `newInstance()`:

```
Animal animal = (Animal)clazz.newInstance();
```
Međutim, u ovom slučaju to neće raditi jer naše životinje nemaju defaultni konstruktor već konstruktor koji prima jedan argument tipa `String`. Stoga se možemo osloniti na *Java Reflection API*, potražiti konstruktor koji prima jedan `String` i potom ga pozvati, kako je ilustrirano u nastavku:

```
Constructor<?> ctr = clazz.getConstructor(String.class);
Animal animal = (Animal)ctr.newInstance(name);
```
U slučaju da `.class` datoteke nisu dostupne u *classpath*-u Javinom virtualnom stroju već su negdje drugdje na disku, stvaranje novih primjeraka uporabom poziva `Class.forName(...)` neće raditi. U tom slučaju potrebno je stvoriti primjerak *ClassLoader*-objekta kojem će se kao argument dati staza do mjesta na kojem se nalaze `.class` datoteke pa koristiti njegovu metodu `loadClass(...)` ili pak inačicu metode`Class.forName` koja prima i referencu na *ClassLoader* koji treba koristiti. Evo primjera:

```
ClassLoader parent = AnimalFactory.class.getClassLoader();

URLClassLoader newClassLoader = new URLClassLoader(
	new URL[] {
		// Dodaj jedan direktorij (završava s /)
		new File("D:/java/plugins/").toURI().toURL(),
		// Dodaj jedan konkretan JAR (ne završava s /)
		new File("D:/java/plugins-jarovi/zivotinje.jar").toURI().toURL()
	}, parent);
```
Sada možemo pisati:

```
Class<Animal> clazz =
(Class<Animal>)newClassLoader.loadClass("hr.fer.zemris.ooup.lab2.model.plugin
s."+animalKind);
```
ili

```
Class<Animal> clazz =
(Class<Animal>)Class.forName("hr.fer.zemris.ooup.lab2.model.plugins."+animalK
ind, true, newClassLoader);
```

Međutim, ako se koriste *ClassLoader*-i, važno je napomenuti da bi naš *Factory* razred trebao pamtiti referencu na već stvoreni *ClassLoader* i uvijek koristiti isti *ClassLoader* za učitavanje iste vrste životinja.

2. Grafičko sučelje za uređivač teksta (70% bodova)

2.1 Upoznavanje s osnovnim komponentama GUI-ja

Proučiti osnovne razrede standardne grafičke biblioteke jezika po izboru koji korisniku omogućavaju stvaranje potpuno prilagođenih komponenata grafičkog korisničkog sučelja (npr. javax.swing.JComponent u Javinoj biblitoteci Swing, System.Windows.Forms.Control u grafičkoj biblioteci C#-a, ili tkinter.TK u standardnoj biblioteci Pythona). Odabrati osnovni razred koji nudi grafičke primitive poput drawLine, drawPolygon ili drawText (ne koristiti razrede koji nude formatirani prikaz grafike i teksta). Proučiti ostvarivanje osnovne interakcije s odabranim osnovnim razredom poput iscrtavanja geometrijskih oblika i teksta te očitavanja pritisnutih tipaka. Nasljeđivanjem osnovnog razreda oblikujte grafičku komponentu koja:

- prikazuje jednu vodoravnu i jednu okomitu crvenu liniju debljine jednog piksela,
- dva retka proizvoljnog teksta,
- zatvara prozor u kojem je smještena kad korisnik pritisne tipku Enter.

Napišite program koji prikazuje prethodno oblikovanu komponentu u zasebnom prozoru (npr. javax.swing.JFrame u Javinoj biblitoteci Swing, System.Windows.Forms.Form u biblioteci C#-a), te završava s radom kad komponenta zatvori svoj prozor.

Vidimo da osnovni razred omogućava da grafički podsustav samostalno poziva naš kod za crtanje kad god se za to javi potreba, iako je oblikovan i izveden davno prije naše grafičke komponente. Koji oblikovni obrazac to omogućava?

Vidimo također da naša grafička komponenta preko osnovnog razreda može dobiti informacije o pritisnutim tipkama bez potrebe za čekanjem u radnoj petlji. Koji oblikovni obrazac to omogućava?

2.2 Osnovni model dokumenta i njegovo prikazivanje

Oblikovati grafičku komponentu TextEditor koja će korisnicima omogućiti prikazivanje i jednostavno uređivanje teksta. Vaš zadatak je pratiti položaj kursora, pritiske na tipke, iscrtavati liniju koja predstavlja kursor te iscrtati tekst na površini komponente. Vaša komponenta treba se temeljiti na primitivnim prozorima (npr. JFrame pod Swingom odnosno Tk() pod tkinterom) te ne smije koristiti komponente visoke razine (npr. Text pod tkinterom odnosno JTextArea pod Swingom).

Sve podatke o tekstu kojeg uređujemo, položaju kursora te trenutno označenom dijelu teksta (selekciji) potrebno je enkapsulirati zasebnim razredom TextEditorModel. TextEditor treba sadržavati TextEditorModel. TextEditorModel treba sadržavati sljedeće podatkovne članove:

- lines ... lista redaka teksta
- selectionRange ... koordinate (redak,stupac) početka i kraja označenog dijela teksta
- cursorLocation ... koordinate trenutnog položaja kursora, odnosno znaka ispred kojeg se nalazi kursor

Koordinate je potrebno enkapsulirati razredom Location. Raspon koordinata je potrebno enkapsulirati razredom LocationRange koji sadrži dva primjerka razreda Location. TextEditorModel u konstruktoru treba primiti znakovni niz s početnim tekstom, razlomiti ga na znakovima prelaska u novi red, te ga pretvoriti u listu redaka.

Isprobajte ispravan rad razvijenih komponenata tako da:

- stvorite primjerak komponente TextEditorModel te ga inicijalizirajte znakovnim nizom s više redaka
- stvorite primjerak grafičke komponente TextEditor, te je povežite s prethodno stvorenim primjerkom razreda TextEditorModel
- povežite taj primjerak sa zasebnim prozorom te prikažite prozor
- provjerite je li tekstovni sadržaj komponente dobro prikazan

2.3 Iteriranje preko redaka teksta

U razred TextEditorModel dodajte i sljedeće dvije metode:

- Iterator allLines(); vraća iterator koji prolazi kroz sve retke dokumenta
- Iterator linesRange(int index1, int index2); vraća iterator koji prolazi kroz dani raspon redaka (prvi uključiv, drugi isključiv).

Ažurirajte kod za prikaz redaka teksta tako da koristi razvijeni iterator.

2.4 Pomicanje kursora

Podrška za kretanje kursora treba uzeti u obzir da se promjene položaja trebaju odraziti i na prikaz kursora u uređivaču kao i u statusnoj traci koju ćemo naknadno dodati. Kako biste to podržali, omogućite razredu TextEditorModel da bude izdavač informacije o položaju kursora:

- definirajte sučelje promatrača CursorObserver s metodom updateCursorLocation(Location loc)
- u razredu TextEditorModel omogućite prijavu i odjavu promatrača tipa CursorObserver
- u razred TextEditorModel dodati metode moveCursorLeft(), moveCursorRight(), moveCursorUp() i moveCursorDown() koje pokušavaju promijeniti položaj kursora
- svaku uspješnu promjenu položaja kursora razred TextEditorModel treba dojaviti svim promatračima

Izmijenite TextEditor na način da nakon svakog pritiska tipaka za pomicanje kursora pozove odgovarajuću metodu razreda TextEditorModel. Također izmijenite metodu za iscrtavanje razreda TextEditor tako da prikazuje položaj kursora kratkom okomitom crtom. Osigurajte da TextEditor dobiva informacije o promjenama položaja kursora bilo da TextEditor naslijedi CursorObserver i prijavi se kao promatrač u svom konstruktoru, bilo da se za to koristi pomoćni (ili još bolje anonimni) razred.

2.5 Označavanje teksta

Omogućiti razredu TextEditorModel da bude izdavač informacije o promjenama teksta:

- definirajte sučelje promatrača TextObserver s metodom updateText()
- u razredu TextEditorModel omogućite prijavu i odjavu promatrača tipa TextObserver
- u razred TextEditorModel dodati metode:
    - deleteBefore(); kojom se briše znak iza kojeg je kursor i pomiče poziciju kursora unatrag (lijevo)
    - deleteAfter(); kojom briše znak ispred kojeg je kursor i ne mijenja poziciju kursora
    - deleteRange(LocationRange r); koja briše predani raspon znakova
    - getSelectionRange(): LocationRange; koja vraća raspon položaja koji čine trenutnu selekciju
    - setSelectionRange(LocationRange range); koja postavlja raspon položaja koji čine trenutnu selekciju
- svaku uspješnu promjenu teksta razred TextEditorModel treba dojaviti svim promatračima

Izmjenite TextEditor tako da pomicanje kursora dok je pritisnuta tipka Shift ažurira aktivnu selekciju u modelu. Također izmijenite metodu za iscrtavanje razreda TextEditor tako da dio teksta koji je označen prikazuje drugačijom pozadinskom bojom. Pozivanjem prethodno definiranih metoda osigurajte da pritisak tipke Backspace briše znak ispred kursora a pritisak tipke Del briše znak iza kursora, ako ne postoji selekcija. Ako postoji selekcija, pritisak bilo koje od tih tipaka briše selektirani tekst.

2.6 Modificiranje teksta unosom novih znakova preko tipkovnice

Omogućite razredu TextEditorModel da modificira dokument umetanjem znakova.

- u razred TextEditorModel dodati metode:
  - insert(char c); kojom se umeće znak na mjesto na kojem je kursor i pomiče se kursor
  - insert(String text); kojom se umeće proizvoljan tekst (potencijalno više redaka) na mjesto na kojem je kursor i pomiče se kursor

Izmjenite TextEditor tako da pritisak znakovnih, numeričkih i sličnih tipaka ažurira trenutni dokument ubacivanjem znakova na poziciju na kojoj je kursor. Ako u trenutku ubacivanja znaka postoji definirana selekcija, ona se najprije briše a potom se ubacuje znak.

Ako je ascii vrijednost znaka 10 (tj. Enter), metoda treba redak u kojem je kursor prelomiti na dva retka: prvi čine svi znakovi koji su bili ispred trenutne pozicije kursora a drugi redak čine znakovi koji su bili iza pozicije kursora; time se povećava broj redaka teksta za jedan. Položaj kursora mijenja se tako da odgovara početku retka koji je nastao od znakova koji su bili iza trenutnog položaja kursora.

2.7 Izrada prilagođenog međuspremnika (clipboard)

Omogućiti razredu TextEditor da koristi vlastiti clipboard koji omogućava prijenos više dijelova teksta. Definirajte razred ClipboardStack koji ima podatkovni član Stack texts; te popratne metode za stavljanje teksta na stog, micanje teksta sa stoga, čitanje teksta s vrha stoga ali bez micanja, ispitivanje ima li teksta na stogu te brisanje stoga.

Omogućiti razredu ClipboardStack da bude izdavač informacije o promjenama u clipboardu:

- definirajte sučelje promatrača ClipboardObserver s metodom updateClipboard()
- u razredu ClipboardStack omogućite prijavu i odjavu promatrača tipa ClipboardObserver

Modificirajte razred TextEditor tako da podrži sljedeće kombinacije tipaka:

- CTRL+C trenutnu selekciju (ako postoji) pusha na stack u clipboard;
- CTRL+X trenutnu selekciju (ako postoji) pusha na stack u clipboard i potom je briše u modelu teksta;
- CTRL+V tekst s vrha stoga u clipboardu čita (ali ne miče) i ubacuje ga u model pozivom metode insert(...)
- CTRL+SHIFT+V tekst s vrha stoga u clipboardu čita i uklanja ga sa stoga te umeće u model teksta pozivom metode insert(...)

2.8 Opoziv izmjena u dokumentu (undo)

Definirajte sučelje EditAction koji sadrži metode execute_do() i execute_undo(), u skladu s oblikovnim obrascem Naredba.

Potom definirajte razred UndoManager čija je struktura sljedeća:

- Stack<EditAction> undoStack;
- Stack<EditAction> redoStack;

- undo(); // skida naredbu s undoStacka, pusha je na redoStack i izvrsava
- push(EditAction c); // brise redoStack, pusha naredbu na undoStack

Neka razred UndoManager bude Subjekt (OO Promatrač) za informacije o statusu stogova undoStack (prazan ili ne) i redoStack (prazan ili ne). Izzborničke stavke za pokretanje naredbi undo i redo mogu biti Promatrači koji se onemogućuju kada je odgovarajuci stog prazan.

Prođite kroz sve metode modela koje mijenjaju tekst (umeću znakove, brišu znakove) te u svakoj od njih stvorite primjerak razreda izvedenog iz EditAction koji pamti informacije potrebne za provođenje i poništavanje akcije (npr. što se briše, što se dodaje, na kojoj lokaciji itd.). Stvoreni objekt pohranite na stog primjerka UndoManagera.

Modificirajte razred TextEditor tako da podrži sljedeće kombinacije tipaka:

- CTRL+Z skida naredbu sa stoga undoStack, prebacuje je na redoStack i pokreće metodu execute_undo().
- CTRL+Y skida naredbu sa stoga redoStack, pokreće metodu execute_do() i prebacuje naredbu na undoStack.

Osigurajte da u programu u svakom trenutku može postojati samo jedan primjerak razreda UndoManager primjenom obrasca Jedinstveni objekt.

2.9 Izborničke stavke i alatne trake

Proučite kako se u grafičkoj biblioteci radi s izbornicima (u Javi to su razredi JMenuBar, JMenu, JMenuItem). U prozor dodajte strukturu izbornika koja je prikazana u nastavku:

```
File
 +- Open
 +- Save
 +- Exit
Edit
 +- Undo
 +- Redo
 +- Cut
 +- Copy
 +- Paste
 +- Paste and Take
 +- Delete selection
 +- Clear document
Move
 +- Cursor to document start
 +- Cursor to document end
```

Proučite kako se u grafičkoj biblioteci radi s alatnim trakama (u Javi to je razred JToolbar). U prozor na vrh dodajte jednu alatnu traku koja sadrži gumbe Undo, Redo, Cut, Copy, Paste.

Osigurajte da izborničke stavke i gumbi alatne trake pokreću za to predviđene akcije. Također osigurajte da su pojedine izborničke stavke te gumbi alatne trake omogućeni samo kada je to smisleno; primjerice, stavka Cut ili Copy ne smije biti omogućena ako ne postoji selekcija u dokumentu; Undo ne smije biti omogućen ako ne postoji barem jedna naredba na undo stogu a Redo ne smije biti omogućen ako ne postoji barem jedna naredba na redo stogu; Paste ne smije biti omogućen ako je clipboard prazan. Kojim oblikovnim obrascem ovo možete postići?

2.10 Statusna traka

Proučite kako se u grafičkoj biblioteci radi sa statusnim trakama (u Javi za to može poslužiti razred JLabel s obrubom). Opremite program statusnom trakom u kojoj se ispisuje trenutni položaj kursora te broj redaka teksta. Kojim oblikovnim obrascem ovo možete postići?

2.11 Dinamičko dodavanje proširenja programa (plugin)

Kao posljednji dio ovog zadatka potrebno je omogućiti dinamičko proširenje funkcionalnosti programa dodavanjem pluginova. Svaki plugin se definira kao biblioteka koja sadrži razred koji izvodi sljedeće sučelje:

```
interface Plugin {
  String getName(); // ime plugina (za izbornicku stavku)
  String getDescription(); // kratki opis
  void execute(TextEditorModel model, UndoManager undoManager, ClipboardStack
clipboardStack);
}
```

Metoda execute(...) prima reference na sve relevantne razrede kako bi mogla ostvariti proizvoljnu funkcionalnost. Potrebno je napraviti sljedeće pluginove:

- **Statistika**: plugin koji broji koliko ima redaka, riječi i slova u dokumentu i to prikazuje korisniku u dijalogu.
- **VelikoSlovo**: prolazi kroz dokument i svako prvo slovo riječi mijenja u veliko ("ovo je tekst" ==> "Ovo Je Tekst").

Ovisno u kojem jeziku radite, program treba na odgovarajući način na disku potražiti i učitati sve raspoložive pluginove te ih dinamički dodati u strukturu izbornika. Za tu potrebu u izborničku strukturu treba dodati još i izbornik "Plugins" čije će stavke program dinamički stvoriti prilikom pokretanja, temeljem pronađenih pluginova.

3. Pametni pokazivači (5% bodova)

Napomena: Ova vježba nije obavezna, ali slobodno je riješite.

Proučite način korištenja pametnih pokazivača posljednjeg standarda jezika C++: **1**, **2**, **3**. Napišite kratke ispitne programe za `unique_ptr`, `shared_ptr` i `weak_ptr`. Proučite izvorni kod implementacije tih komponenata u distribuciji vašeg prevodioca (<memory>) i pokušajte razumjeti kako te komponente rade, na konceptualnoj razini.

4. Parsanje aritmetičkih izraza (5% bodova)

Napomena: Ova vježba nije obavezna, ali slobodno je riješite.

Potrebno je napisati jednostavni rekurzivni program za parsiranje, prikazivanje i evaluiranje aritmetičkih izraza. Program treba podržavati operacije zbrajanja, oduzimanja, množenja i dijeljenja, te grupiranje zagradama nad brojčanim i simboličkim podatcima. Evaluiranje simboličkih podataka implementirajte prozivanjem globalnog rječnika `Symbols`.

Neka se program temelji na funkciji `parseExpression` koja treba kreirati kompozit koji predstavlja sintaksno stablo izraza. Sve komponente kompozita trebaju definirati konstruktor, metodu `toStr` koja sadržaj kompozita izražava znakovnim nizom (u Pythonu ovu metodu ima smisla nazvati `__str__` ili `__repr__`), te metodu `evaluate` koja evaluira vrijednost kompozita.

Ispitajte razvijeni program na način da ručno postavite vrijednosti simbola, pokrenete parsiranje izraza zadanog znakovim nizom, te zatim ispišete parsirani izraz kao i njegovu numeričku vrijednost. Preporučeni primjer ispitivanja u interaktivnoj ljusci Pythona prikazan je u nastavku.

```
>>> tree=parseExpression("6*(x+4)/2-3-x")
>>> tree.toStr()
((((6.0*(x+4.0))/2.0)-3.0)-x)
>>> tree.evaluate() # ovo ne radi jer ne znamo x!
...
KeyError: 'x'
>>> Symbols['x']=5
>>> tree.evaluate()
19.0
>>> x=5; 6*(x+4)/2-3-x # proba
19.0
>>> Symbols['x']=4     # radi i za drugi x
>>> tree.evaluate()
17.0
```

Pomoć

Parsiranje može biti vrlo težak zadatak ako problemu ne pristupimo na pravi način. Situaciju posebno kompliciraju sljedeći problemi:

- različiti prioriteti operatora,
- grupiranje izraza zagradama,
- desna asocijativnost operatora.

Stoga elegantno rješenje možemo postići ako razrješavamo operatore redoslijedom koji odgovara njihovom prioritetu. Pri tome moramo paziti da zanemarimo operatore koji su uokvireni zagradama. Kako bismo pogodili asocijativnost, operatore treba uzimati u obzir s desna na lijevo. Kad obradimo sve operatore, potrebno je provjeriti radi li se o izrazu kojeg grupiraju zagrade. Ako to nije slučaj, vraćamo atomarni izraz koji može biti broj ili simbol.

Kôd koji implementira gore skicirani algoritam parsiranja prikazan je u nastavku. Prikazani kôd treba modificirati na način da na izlazu proizvede kompozit kojeg jednostavno možemo ispisati i evaluirati, u skladu s uputama danim na početku zadatka. Traženi program može se napisati u manje od 60 redaka Pythona, uključujući i 20 redaka modificirane funkcije `parse`.

```
# adapted from  http://news.ycombinator.com/item?id=284842
def parse(strinput):
  for operator in ["+-", "*/"]:
    depth = 0
    for p in range(len(strinput) - 1, -1, -1):
      if strinput[p] == ')': depth += 1
      elif strinput[p] == '(': depth -= 1
      elif depth==0 and strinput[p] in operator:
        # strinput is a compound expression
        return (strinput[p], parse(strinput[:p]), parse(strinput[p+1:]))
  strinput = strinput.strip()
  if strinput[0] == '(':
    # strinput is a parenthesized expression?
    return parse(strinput[1:-1])
  # strinput is an atom!
  return strinput
```