LINK NA GOOGLE DOCUMENTS:
https://docs.google.com/document/d/1nPcpnWGu-b0M8SRnTU9uV0zi6ke1u81JWcH3zPisxrE/edit?hl=en&authkey=CI-f3NQJ

1. Zadan je razred koji implemetira dohvat, obradu i pohranu web stranica kako slijedi

ORIGINAL:

```
class WebPageProcessor{
    public:
      void processUrl(std::string url);
   private:
      std::string getPage(std::string url);
      std::string extractTextFromHTML(std::string html);
      void storeTextToDatabase(std::string text);
};

void WebPageProcesor::processUrl(std::string url) {
      std::string html = getPage(url);
      std::string text = extractTextFromHTML(html);
      storeTextToDatabase(text);
};
```

Na početku razvoja bilo je važno što prije isporučiti verziju koja ispravno radi za protokol HTTP stranice u HTML-u, te baze MySQL. Međutim, sada je potrebno podržati i mogućnost spremanja u bazu Oracle.
Pokušajte poboljšati organizaciju koda kako bi se potrebno proširenje postiglo u skladu s načelima oblikovanja koja su navedena u predavanjima. Objasni koja načela oblikovanja su zadovoljena u tvom tješenju i kako.

SOLUTION:

```
class ApstraktnaBaza {
   public:
      storeText(std::string text)=0;
}
class WebPageProcessor{
   ApstraktnaBaza& mojabaza;
   public:
  // Ovako se inicijalizira referenca u konstruktoru
      WebPageProcessor(ApstraktnaBaza &db) : mojaBaza(db){}
      void processUrl(std::string url);
   private:
      std::string getPage(std::string url);
      std::string extractTextFromHTML(std::string html);
      void storeTextToDatabase(std::string text);
};
```

```
void WebPageProcesor::processUrl(std::string url) {
      std::string html = getPage(url);
      std::string text = extractTextFromHTML(html);
      storeTextToDatabase(text);
};

void WebPageProcesor::storetextToDatabase(std::string text) {
      mojabaza->storeText(text);
}
KORISTENJE

class KonkretnaBaza : ApstraktnaBaza
public:
      storeText(std::string text){
            //do something
      }
};

int main() {
      KonkretnaBaza baz = new KonkretnaBaza();
      WebPageProcessor proc = new WebPageProcessor(baz);
      proc.procesUrl("nekiurl");
      }

//Mislim da bi ovo trebalo biti više manje to...po uzoru na labos.
//valjda onda nacelo inverzije ovisnosti ako ovisi o apstraktnoj bazi
//Injekcija ovisnosti (netko iz vana će odrediti koja se konkretna //
implementacija koristi) i NBP također.
```

2. Zadan je razred Animal:

ORIGINAL:

```
class Animal {
    public:
      virtual ~Animal(){};
      virtual void run()=0;
      virtual void swim()=0;
      virtual char const * getName()=0;
}

class MyDog : Dog {
    public:
      virtual char const *getName() {
            return "Rex";
```

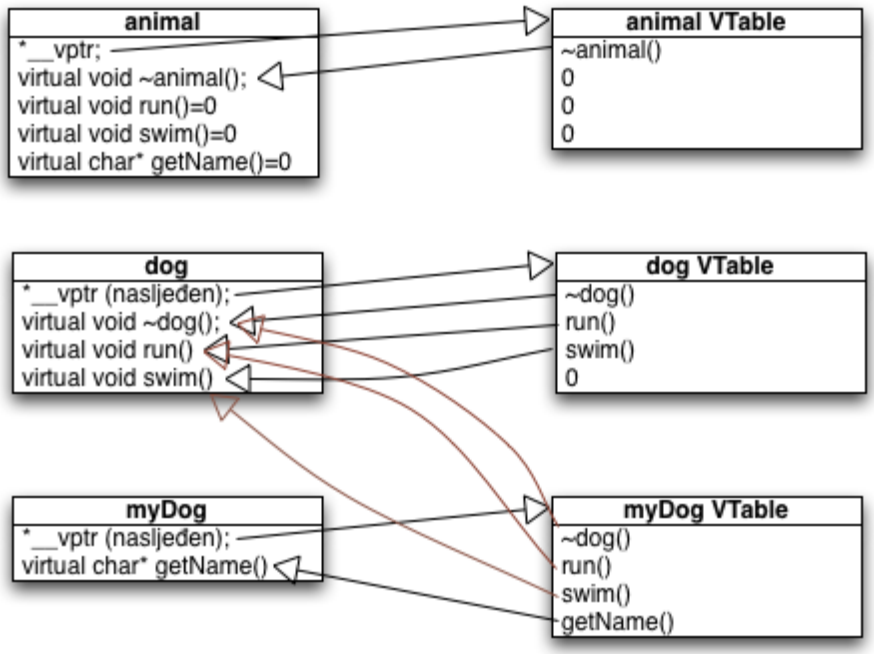
```
            };
      }
```

SOLUTION:

(a) Skicirajte definiciju razreda Dog

```
class Dog: public Animal {
      virtual ~Dog(){};
      virtual void run(){};
      virtual void swim(){};
};
```

(b) Skiciraj tablicu virtualnih funkcija za razred MyDog. Za svaki element tablice navedi na što se odnosi.



(c) Navedi pristupe memoriji koji se trebaju dogoditi prilikom izvršavanja funkcije:

ORIGINAL

```
bool xyzzy(Animal* x) {
      return x->getName() !=0;
}
```

SOLUTION:

1. pristupa se x
2. pristupa se virtualnoj tablici od X
3. pristupa se sadržaju u virtualnoj tablici od Animal na koji gledamo, i taj sadržaj se preda x-u kao argument

(d) Koja vrsta polimorfizma je prisutna u navedenom primjeru?

Dinamički polimorfizam

3. Zadana su sučelja hipotetskog programa za vektorsku grafiku. Komponenta **Shape** modelira pojedinačni element crteža, npr. trokut ili kvadrat. Komponenta ShapeGroup modelira grupu osnovnih elemenata kojoj se ne može pristupati i preko sučelja razreda **Shape**. Metoda **addToGroup** dodaje oblik zadanoj grupi, dok metoda **makeGroup** od oblika stvara **ShapeGroup** s jednim elementom.

```
ORIGINAL
// component Shape
class Shape{
    public:
      virtual int getX() = 0;
      virtual int getY() = 0;
    public:
      void addToGroup(ShapeGroup& group);
      ShapeGroup makeGroup();
}

//component ShapeGroup
class ShapeGroup: public Shape{
      // …
  private:
      std::list<Shape*> shapes;
}
```

Koje su loše strane prikazane organizacije komponenata Shape i ShapeGroup? Predloži promjene kojima bi se organizacija poboljšala uz zadržavanje iste funkcionalnosti.

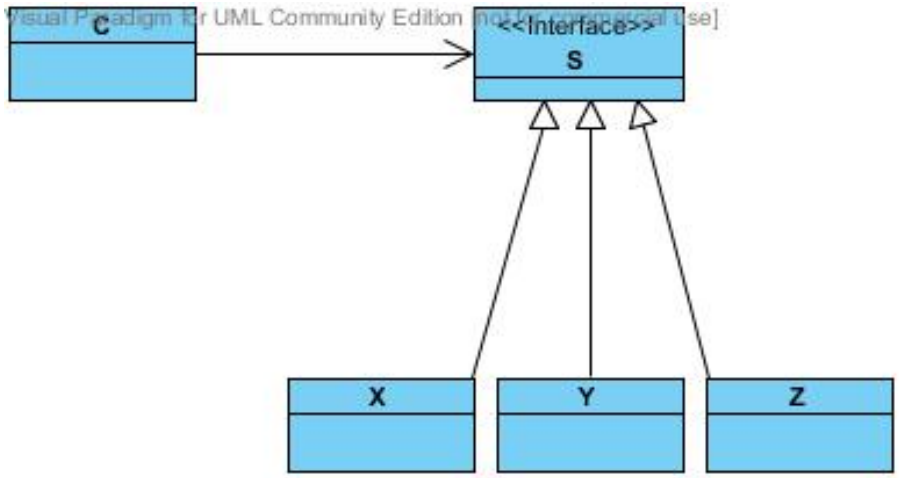
```
SOLUTION:

class AbstractShape{
      public:
       virtual int getX() = 0;
       virtual int getY() = 0;
};
class ShapeGroup: public AbstractShape{
      // ...
      private:
       std::list<AbstractShape*> shapes;
};
class ConcreteShape: public AbstractShape{
    public:
      virtual int getX() = 0;
      virtual int getY() = 0;
    public:
      void addToGroup(ShapeGroup& group);
```

```
      ShapeGroup makeGroup();

};

/* Mislim da bi to bilo to... Fora je u tome da u originalu Shape
ovisi o ShapeGroup, a ShapeGroup ovisi o Shape -> ciklus = BAD! (bar
kako kaže šegvić). E sad, ideja je da se jedan od tih razreda razdvoji
u 2 dijela, u ovom slučaju je to razred Shape, napiše se abstract
class (sučelje) koje ćemo nazvati AbstractShape, to sučelje će
naslijediti ShapeGroup. Konkretan Shape će naslijediti AbstractShape
i preuzeti metode. Nisam samo siguran za ove dodatne 2 metode, ali
budući da su one razlog za ciklus, mislim da je ovako ok.
Neka nikog ne zbuni... ConcreteShape zapravo nije konkretan. On je
isto apstraktan.  */
```

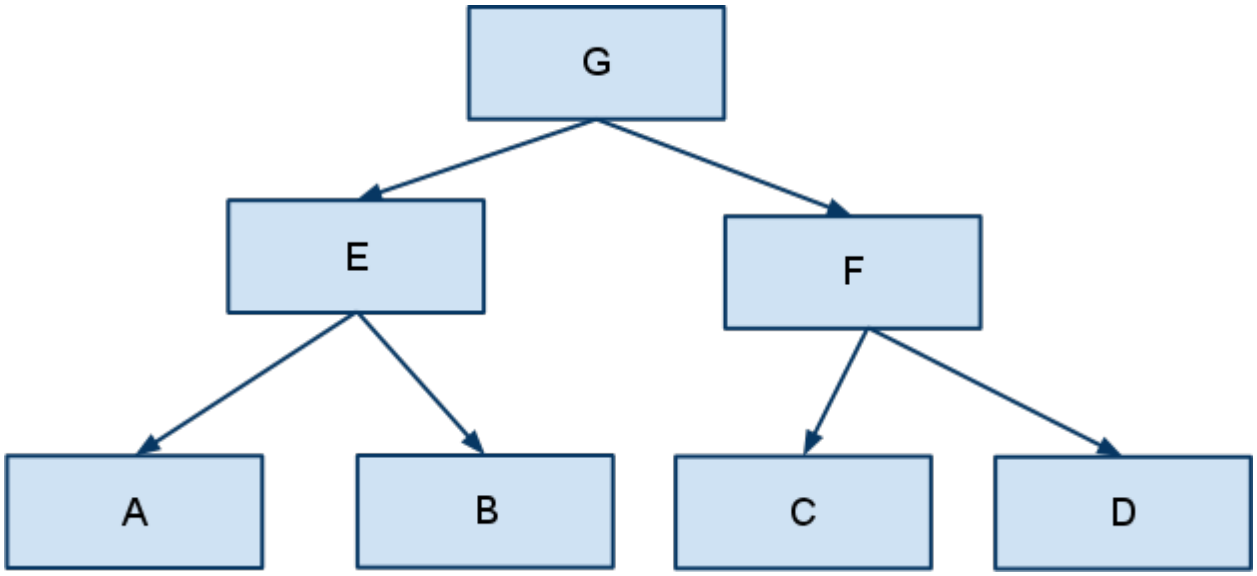
4. Apstraktni osnovni razred **S** definira virtualnu funkciju ***drive***. **S** nasljeđuju konkretni razredi **X**, **Y**, i **Z**. Razred **C** prima pokazivač na **S** u konstruktoru, sprema ga kao podatkovni član, te ga koristi u metodi navigate. Nacrtaj dijagram razreda te skiciraj implementaciju u C++-u.

```
class S {
public:
      virtual  void  drive() =0;
};

class X : public S {
public:
      void drive () {}
};

class Y : public S {
public:
      void drive () {}
};

class Z : public S {
public:
      void drive () {}
};

class C {
private:
      S* _s;
public:
      C(S *s) {
            this->_s = s;
```

```
        }

        void navigate () {
              _s->drive();
        }
}
```



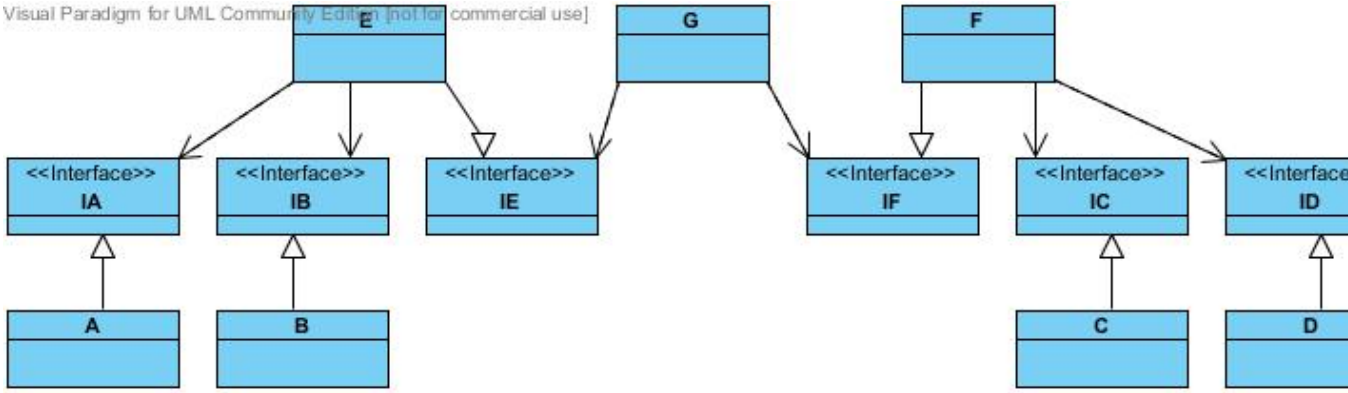
5. Neka je zadana struktura ovisnosti među komponentama A - G programskog sustava zadana kako slijedi!



Pokaži kako bismo sustav mogli preoblikovati na način da u skladu s načelom inverzije ovisnosti smanjimo utjecaj promjena u konkretnim komponentama na glavnu komponentu sustava.

SOLUTION:
// Znači radimo inverziju ovisnosti

Rješenje #2: Ovisnost prema apstraktnim klasama, tj. sučeljima. Glavna komponenta sada ovisi o sučeljima, pa promjene u konkretnim razredima ne utječu na glavnu komponentu.



6. Zadano je sučelje komponente **Database**. Konstruktor otvara komunikaciju s bazom, metoda **getAttribute** dohvaća zadani atribut prema zadanom ključu, dok metoda **query** zadaje SQL upit, te rezultat smješta u polje **queryResult.**

ORIGINAL:

```
class Database{
public:
      Database(std::string path);
      std::string getAttribute(   std::string key,
                                  std::string attribute);
      void query (std::string STLquery);
public:
      std::vector<std::string> queryResult;
      std::vector<std::string> keys;
      std::vector<std::string> attributes;
};
```

Predloži promjene koje bi klijentima komponente **Database** olakšale nadogradnju bez promjene.

SOLUTION:
Znači nesmijemo imati direktne pristupe podatkovnim članovima pa bi bilo dobro da te članove enkapsuliramo u nekakve metode. Npr. metode getResult(), getKeys(), getAttributes().

Dodatak1: trebalo bi staviti atribute samog razreda kao private da se skroz onemogući direktan pristup, mislim da bi tako bilo bolje. Također, osim “get” metoda, treba staviti i “set” metode da se može mijenjati po potrebi.

Moguce rjesenje:

```
class Database{
      std::vector<std::string> queryResult;

public:
      Database(std::string path);
      std::vector<std::string> executeQuery(std::string query);
};

class DatabaseReader {
  private:
      Database& baza;
      std::vector<std::string> keys;
      std::vector<std::string> attributes;
      std::vector<std::string> queryResult;
   public:
      DatabaseReader(Database &db):baza(db){}

      void query (std::string STLquery) {
            this->queryResults = db.executeQuery(STLquery);
      };

      std::string getAttribute(  std::string key,
                                 std::string attribute);

      std::vector<std::string> getKeys(){};
      std::vector<std::string> getAtrtibutes(){};
};
```

7. Koje načelo oblikovanja je prekršeno u prikazanom odsječku? Kako bismo mogli poboljšati (navedi barem dva načina)?

ORIGINAL:

```
class Animal{
      virtual char const* sing() =0;
};

class RescueDog:public Animal{
      virtual char const* sing(){
            return "woof";
      }
      virtual void trains() {
            // ....
      }
};
```

```
class PetOwner {
      virtual void adopt(Animal* a)  = 0;
};

class RescueDogOwner: public PetOwner {
      virtual void adopt(Animal* a) {
            //PRECONDITION: a is a RescueDog
            ((RescueDog*) a) -> train();
      }
};
```

SOLUTION:

- prekršeno je LNS ako se ne varam

1. Način: Slide 51: bacamo exception i obradujemo ga na mjestu pozivanja metode

```
class RescueDogOwner: public PetOwner {
      virtual void adopt(Animal* a) {
            //PRECONDITION: a is a RescueDog
            if (typeid(a) != typeid(RescueDog)
                  throw Exception();
            ((RescueDog*) a) -> train();
      }
};
```

2. Način - dvije funkcije adopt

```
class RescueDogOwner: public PetOwner {
      // Ova metoda se može izostaviti.
      virtual void adopt(Animal* a) {
            base::adopt(a);
      }
      virtual void adopt(RescueDog* a) {
            a -> train();
      }
};
```

3. Način - raskidamo rodbinsku vezu

```
class RescueDogOwner {
      virtual void adopt(RescueDog* a) {
            a -> train();
      }
};
```