Oblikovni obrasci u programiranju

Siniša Šegvić
Marko Čupić

Auditorne vježbe 2013. - rješenja starih ispita

Full protected by **thoc**

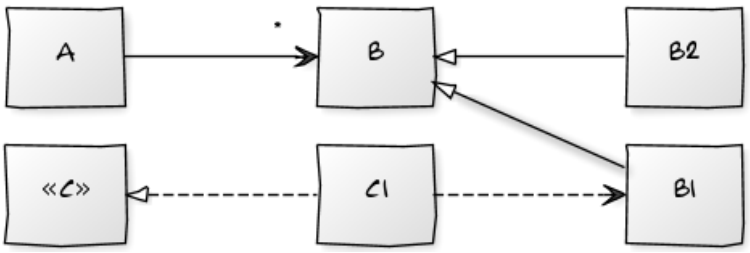
NAPOMENE

PDF je nastao kao proširena i nekim pojašnjenjima nadopunjena verzija već objavljenih auditornih vježbi održanih 14.06.2013. na kojima su rješavani na službenim stranicama predmeta objavljeni stari ispiti. Nekim vlastitim bilješkama pokušao sam, prvenstveno sebi, olakšati praćenje suhoparno prepisanog rješenja sa ploče. Ako ovo pomogne još nekome – tim bolje!

Sa istim ciljem složio sam i neke primjere koji pokrivaju zadatke u kojima se i na ispitu traži barem neka skica programskog rješenja.

Programski primjeri osposobljeni su za neko osnovno testiranje, a napisani su u Pythonu 3.3. Nadopuna su pseudokoda i konkretnih dijelova kôda predloženih od strane profesora kao osnovu za rješenja zadataka na auditornima. Oni nisu rješenja zadataka jer su konkretniji i opsežniji od zahtjeva koji su postavljeni na ispitu. Služe za demonstraciju i lakše praćenje ove tekstualne (i grafičke) nadopune spomenutih auditornih (ovog PDF-a, jel') uz puno ispisa trenutnih operacija i od kuda je što programski pozvano.

Korištena je nekakva kombinirana notacija kad su *class diagrami* (UML) u pitanju, vjerujem da će sve bit jasno, a izrađeni su na http://yuml.me/. Tamo imate i neke dodatne primjere da vam razjasne eventualne nejasnoće u notaciji.



`A` referencira više `B`-ova.
`B1` i `B2` nasljeđuju `B`.
`C` je sučelje, a implementira ga `C1`.
`C1` stvara `B1`.

Nemojte zamjerit na nekim nekonzistentnostima u ovim notacijama i korištenju naših / engleskih izraza.

Svaki prijedlog, ispravka ili nadopuna je dobrodošla, najbolje na PM ili u temi na f2.

ZAVRŠNI ISPIT 2012.

ZADATAK 1

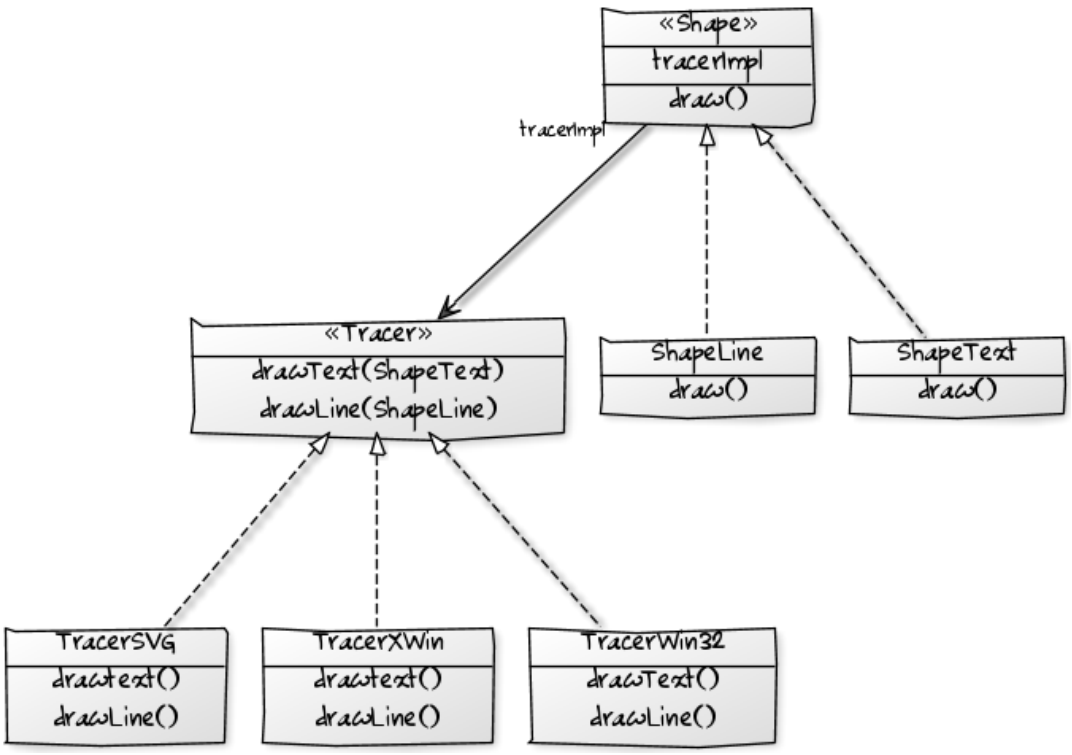
Razmatramo program za vektorsku grafiku koji treba iscrtavati geometrijske likove poput trokuta, kvadrata, pravocrtnih segmenata i tekstovnih oznaka. Iscrtavanje se treba moći transparentno provesti pod Microsoftovim Windowsima i XWindowsima, kao i u datotečni format SVG. Predloži organizaciju koja bi omogućila istovremenu mogućnost dodavanja novih geometrijskih likova te novih načina iscrtavanja. Pripazi da u rješenju minimiziraš ponavljanje.

Uočavamo 2 razdvojene osi razreda koje moramo posložiti da funkcioniraju zajedno. S jedne strane likovi, a s druge strane načini iscrtavanja. Iscrtavanje je jedina akcija koja nas se tiče za potrebe ovog zadatka, dodavanje nekih novih operacija bi znatno zakompliciralo stvari jer bi onda konkretni poziv metode ovisio o 3 stvari: koju operaciju želimo izvršiti, na koji je način želimo izvršiti i na kojem liku je želimo izvršiti. Ovako smo ograničeni na crtanje oblika ovisno o konkretnom razredu oblika i konkretnoj implementaciji načina tog iscrtavanja.

Te 2 kategorije razreda ćemo u svakom slučaju organizirati kao konkretne razrede koji implementiraju zajedničko sučelje / apstraktni bazni razred; `Shape` i `Tracer` („Ne koristimo naziv `Drawer` jer to previše asocira na ladicu“, S.Š. ☺).

v1

Rješenje uporabom Mosta znači nam razdvajanje ovih oblika i izvođača i `Tracera` koji pruža sučelje za konkretne implementacijske razrede: `TracerWin32`, `TracerXWin`, `TracerSVG`... Oblik treba imati referencu na konkretnog izvođača.



Metode `draw()` konkretnog oblika prosljeđuju zahtjev zadanom izvođaču (`Traceru`) kojeg ne smijemo zaboraviti negdje pohraniti za svaki `Shape` (bilo u konstruktoru, bilo nekim *setterom* prije prvog korištenja) pa će tako metoda `draw()` konkretnog oblika `ShapeText` izgledati:

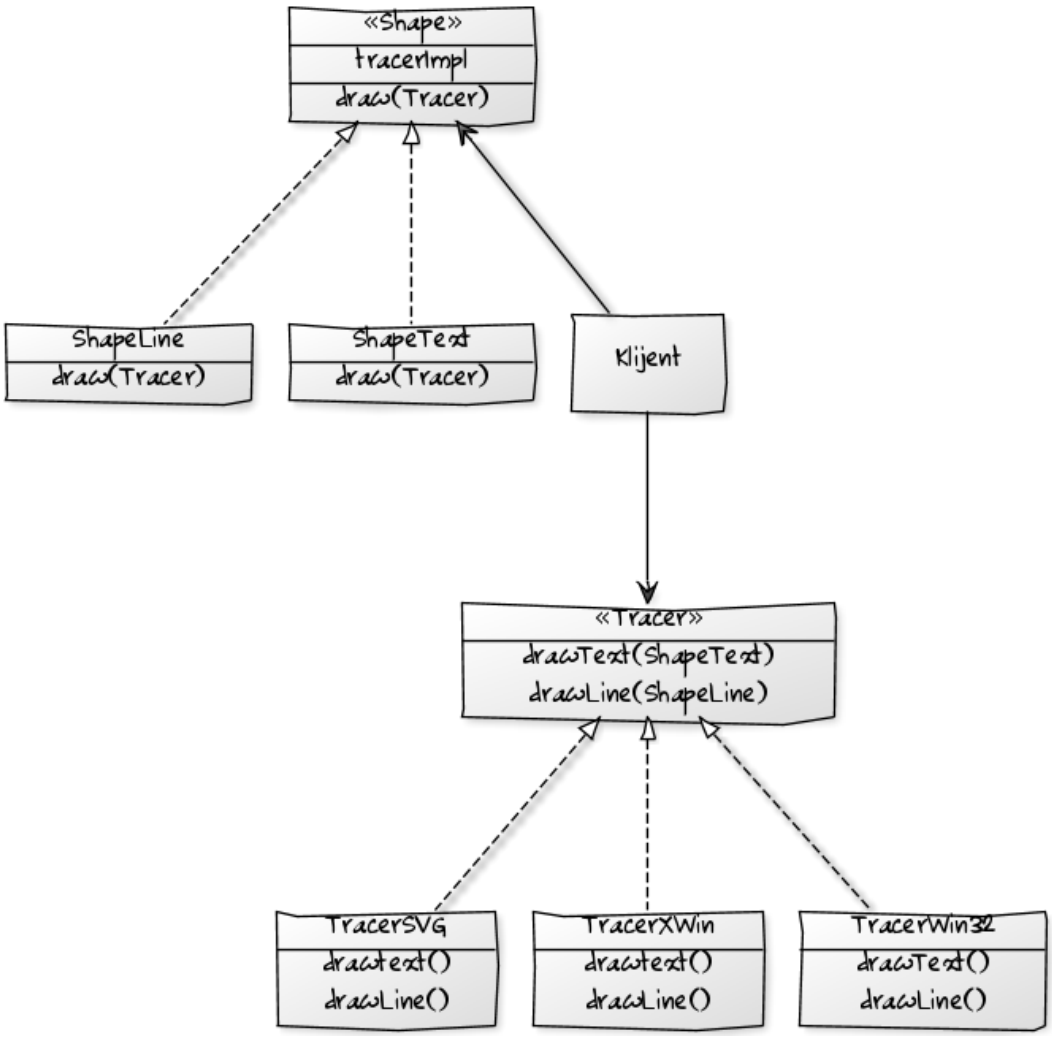
```
def draw(self):
    self.tracerImpl.drawText(self)
```

Nakon što imamo pohranjene sve stvorene oblike u nekakvoj varijabli shapes sa pridruženom referencom na konkretnu implementaciju crtanja, klijentska funkcija koja bi nad takvom jednom, recimo listom oblika izvršila iscrtavanje glasi:

```
def iscrtaj(shapes):
    for shape in shapes:
        shape.draw()
```

v2

Drugo rješenje je uporabom Visitora („Bolje rješenje“, S.Š). Ishod poziva će i dalje ovisiti o konkretnom obliku (elementu) i konkretnom načinu iscrtavanja (posjetitelju). Metode `Tracera` ponovo će dobivati reference na domenski objekt (`Shape`) na način da pozivom metode `draw` nad tim oblikom on pošalje „sebe“. `draw()` nam je sada ekvivalent metode generički znane u vidu ovog obrasca kao `accept()`.



```
class ShapeText(Shape):
    def draw(self, tracer):
        tracer.drawText(self)
class ShapeLine(Shape):
    def draw(self, tracer):
        tracer.drawLine(self)
```

Sada ta metoda prima i konkretnog `Tracera` kao argument, a njega stvara klijent koji će nad svakim objektom razreda `Shape` pozvati metodu `draw()` predajući joj referencu na Tracera.

```
def iscrtaj(shapes, tracer):
    for shape in shapes:
        shape.draw(tracer)

tracerImpWin32 = TracerWin32()
iscrtaj(shapes, tracerImpWin32)
```

Osobno mi se ne sviđa naglašavanje u zadatku *„Predloži organizaciju koja bi omogućila istovremenu mogućnost dodavanja novih geometrijskih likova..“* jer to značajno dovodi u pitanje upotrebu ova 2 obrasca koji (osobito Posjetitelj) se oslanjaju na stabilnost domenskog objekta (u našem slučaju `Shapeova`) s obzirom da dodavanjem svakog novog lika moramo mijenjati sve Visitore, odnosno Izvođače ako smo se odlučili za rješenje s Mostom.

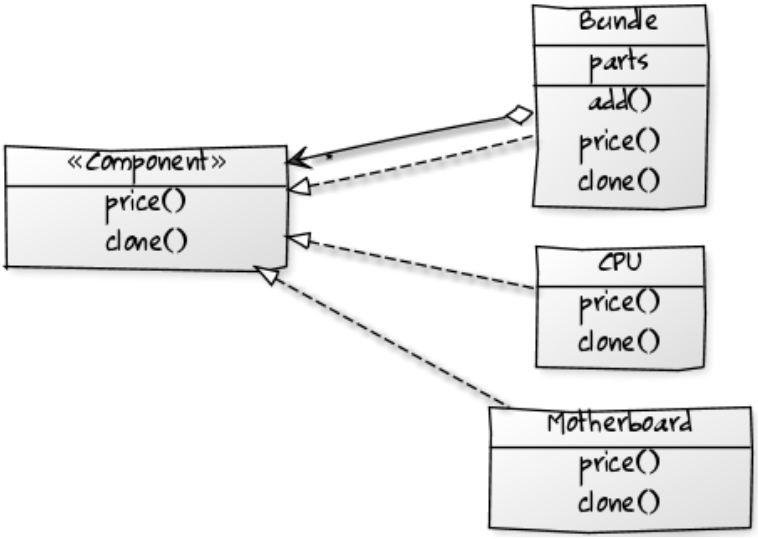
ZADATAK 2

Razmatramo oblikovanje programa za internetsku prodaju elektroničkih komponenata. Program mora omogućiti kupovinu jednostavnih artikala poput matične ploče, kao i složenih artikala koji se sastoje od većeg broja jednostavnih artikala. Složeni artikli mogu sadržavati druge složene artikle: npr. stolna konfiguracija sastoji se od računala, monitora i zvučnika, a računalo se opet sastoji od matične ploče, kućišta itd. Radi jednostavnosti, sve artikle koje je korisnik izabrao tijekom jedne kupovine također treba predstaviti složenim artiklom. Predloži rješenje koje bi omogućilo predstavljanje složenih artikala, računanje njihove ukupne cijene, kao i jednostavno stvaranje složenog artikla iz postojećeg (npr. kad kupcima želimo ponuditi jednostavan odabir unaprijed konfiguriranih složenih artikala).

Bez puno mudrosti može se uočiti nužnost za korištenje Kompozita. Naglašeno nam je postojanje jednostavnih artikala: matična ploča, CPU, zvučnik, kućište itd. kao i složenih artikala koji se sastoje od jednostavnih artikala, ali i složenih artikala koji se sastoje od drugih složenih artikala.

Artikl nam predstavlja Komponentu koju onda implementiraju i jednostavni artikli – Primitivi / Listovi; ali i složeni artikli – Kompoziti. Preko sučelja `Component` klijentu omogućavamo baratanje i složenim objektima (`Bundle`) i primitivima (`Motherboard`, `ComputerCase`, `CPU`, `Speaker`, `Monitor`…).

Takvo zajedničko sučelje odgovara nam i za ispunjavanje ovog dijela zadatka: *„jednostavno stvaranje složenog artikla iz postojećeg“* jer ćemo klijentu omogućiti operaciju kloniranja nad Prototipom i sve što on zna jest da će mu povratna vrijednost biti objekt razreda `Component`, a hoće li to biti jednostavni artikl (i koji točno jednostavni artikl) ili složeni artikl (i koji točno složeni artikl – kompozit) odlučuje se za vrijeme izvođenja.



Na ovom dijagramu prikazani su samo primjera radi `CPU` i `Motherboard` kao jednostavni artikli, za potrebe programske demonstracije napisao sam ih još nekolicinu.

Njihovu cijenu (koju recimo postavimo u konstruktoru) dohvaćajmo jednostavnom *getterskom* metodom:

```
def cijena(self):
    return self.price
```

Ovo izgleda malo glupo kad je imenovanje u pitanju jer se kod Pythona treba paziti poziva li se metoda ili se dohvaća član razreda. Nisam dodatno kompliciraO stvari zaobilazeći neka pravila i stvarajući članove razreda nasilno „privatnima“.

Kompozit ima neku svoju kolekciju u kojoj su pohranjene komponente koje ga sačinjavaju (bilo da su to drugi kompoziti ili primitivi). Programski rješeno kao lista u Pythonu. Tu je onda i metoda `add()` koja kao argument prima objekt razreda `Component` (nije naznačeno na dijagramu), a koju moramo implementirati za složene artikle i koja će omogućiti dodavanje novih komponenti u gore spomenutu kolekciju (eventualno i nekakav `remove()` za uklanjanje, za potrebe ispita sasvim sigurno nije neophodno).

Cijena jednog složenog artikla bit će zbroj cijena svih artikala koji ga sačinjavaju:

```
def cijena(self):
    rw = 0
    for part in self.parts:
        rw += part.cijena()
    return rw
```

Mogućih implementacija kloniranja složenih objekata ima više, s obzirom da nema nekih posebnih naputaka u tekstu zadatka, nećemo previše kompliciratI nego stvorimo novi `Bundle` i dodamo mu sve komponente koje sadrži `Bundle` koji kloniramo te novostvoreni objet vratimo kao povratnu vrijednost metode. Eventualnu dvojbu treba li raditi *deep copy* ili *shallow copy* također zanemarimo.

Znam da sam programski stvorio džumbus ali to je samo da bi imao neki pristojan ispis za demonstrativne svrhe. Bitno je uočiti da na kraju imamo „košaricu“ koja je razreda `Bundle`, a u njoj su i jednostavni artikli poput matične ploče, i složeni artikli poput *desktop* konfiguracije koju sačinjavaju računalo (kojeg, opet, sačinjavaju neki primitivi), sustav ozvučenja (kojeg sačinjavaju pojedinačni zvučnici i woofer – primitivi), monitor koji je primitiv ...

ZADATAK 3

Zadana je funkcija za izračunavanje Fibonaccijevih brojeva, te ispitni program.

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
def test():
    print([ fib(n) for n in range(5)])
```

Koliko puta će se pozvati funkcija `fib` pri evaluiranju ispitnog programa? Kako bismo taj broj poziva mogli smanjiti primjenom cacheirajućeg Proxyja?

Ovaj `fib(n) for n in range(5)` poziva funkciju `fib()` nad brojevima `0`, `1`, `2`, `3` i `4` pa to dovodi do sljedećih poziva:

`fib(0)`

`fib(1)`

`fib(2)` `fib(1)` `fib(0)`

`fib(3)` `fib(2)` `fib(1)` `fib(0)` `fib(1)`

`fib(4)` `fib(3)` `fib(2)` `fib(1)` `fib(0)` `fib(1)` `fib(2)` `fib(1)` `fib(0)`

Ukupno 19 poziva. To će nam potvrditi i nadopunjena skripta `zi2012_zad3_fibonacci_v0.py`.

Lako je uočiti uzorke ponavljanja prilikom poziva funkcija i porast broja poziva funkcije za iste argumente prilikom ulaska u rekurziju za veće brojeve. Za izračunavanje vrijednosti funkcije za broj 4, program 3 puta ponavlja računanje vrijednosti funkcije za broj 1. Koja je u ovom primjeru do tada izračunata već 4 puta!!

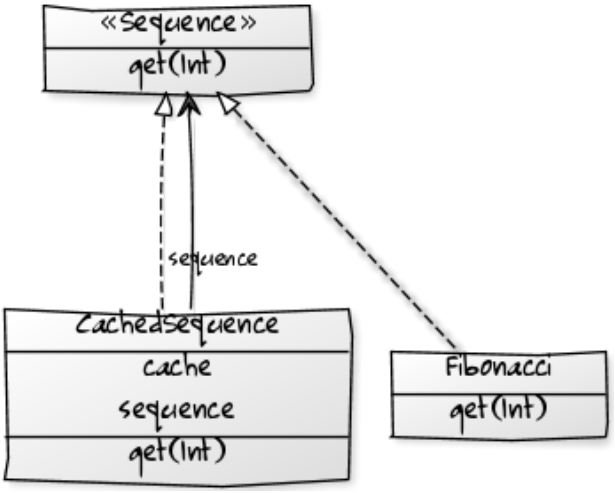
Na ovom jednostavnom primjeru Fibonaccija to možda nije kritično, ali za neku rekurzivnu funkciju koja prije konačnog izračunavanja povratne vrijednosti obavlja mali milijun drugih stvari i troši vrijeme i resurse to je situacija koju svakako želimo izbjeći. Zato bi povratne vrijednosti bilo dobro pamtiti u nekakvoj varijabli.

v1

Za takvu pohranu poslužiti će nam mapa / rječnik `cache`. Par ključ – vrijednost predstavljat će nam integer `n` i povratna vrijednost funkcije `fib()` od tog integera, tim redom.

Taj `cache` će nam biti član razreda `CachedSequence` koji implementirazajedničko sučelje `Sequence`, a kojeg također implementira i razred `Fibonacci` na kojeg `CachedSequence` ima referencu. Metoda `get()` unutar razreda `Fibonacci` ekvivalent je funkcijiji `fib()` zadanoj u tekstu zadatka.

```
class CachedSequence(Sequence):
    def __init__(self, seq):
        self.sequence = seq
        self.cache = dict() # self.cache = {}
```



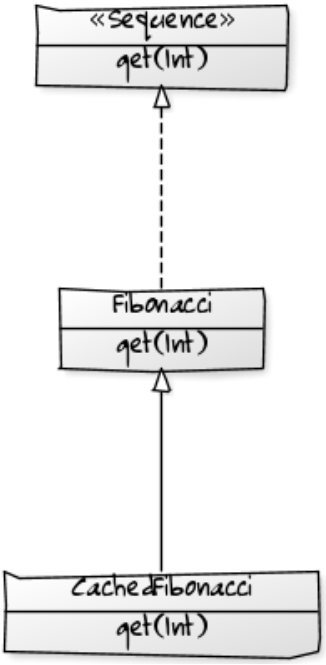
Metodu get unutar ovog razreda CachedSequence implementirat ćemo na sljedeći način:

```
def get(self, n):
    if n in self.cache.keys():
        return self.cache[n]
    else:
        fib_n = self.sequence.get(n)
        self.cache[n] = fib_n
        return fib_n
```

Problem kod jedne ovakve rekurzije jest da referencirani razred `Fibonacci` nema nikakav doticaj sa rječnikom `cached` pa ovom organizacijom za konkretni primjer nismo ništa postigli osim što smo ispunili traženo u zadatku da imamo *by the book* primjer *cacheirajućeg Proxya*.

Broj poziva metode `get()` ostaje 19! (primjer `zi2012_zad3_fibonacci_v1.py`)

v2



Promjenom organizacije gubimo koncept oblikovnog obrasca, ali kaže S.Š. da to uopće nije problem dok god je rješenje zadatka točno. Kakve sad koristi imamo da `Fibonacci` bude apstraktna klasa koja implementira zajedničko sučelje `Sequence`, a koju nasljeđuje `CachedFibonacci` postat će nam jasno kad vidimo programsku implementaciju:

```
class CachedFibonacci(Fibonacci):
    def __init__(self):
        self.cache = dict() # self.cache = {}

    def get(self, n): (1)
        if n in self.cache.keys():
            return self.cache[n]
        else:
            fib_n = super(CachedFibonacci, self).get(n)
            self.cache[n] = fib_n
            return fib_n
```

Razlika u jednom jedinom retku čini veliku razliku! Više ne pozivamo metodu `self.sequence.get(n)` na referenciranom objektu klase `Fibonacci` već metodu `get(n)` definiranu u roditeljskoj klasi (jer smo je u ovoj *overrideali*) ali sa trenutnom klasom kao objektom nad kojim se ta metoda izvodi.

Dodatno pojašnjenje: Ako prilikom izvođenja medode `get(n)` (1) definirane unutar `CachedFibonaccija` zbog (ne)ispunjenja uvjeta dođe do poziva:

`super(CachedFibonacci(3), self(4)).get(n)` (2)

zapravo će se pozvati metoda `get(n)` (2) definirana u klasi koju `CachedFibonacci` (3) nasljeđuje:

```
class Fibonacci(Sequence):
    def get(self(4), n):
        if n < 2:
            return n
        return self(4).get(n-1) + self(4).get(n-2)
```

ali ne nad tim objektom roditeljske klase, već nad objektom iz kojeg je pozvana – objektom klase `CachedFibonacci` (4)!

Zbog toga će ovi pozivi `self(4).get(n-1)` i `self(4).get(n-2)` završavati kao pozivi metoda `get()` (1) iz `CachedFibonaccija`!! Nadam se da se nisam pogubio u ovom šarenilu...

Python sintaksa za isvesti ovo je opisana: `super(CachedFibonacci, self).get(n)`
C++: `Fibonacci::get(n)`
C#: `base.get(n)`
Java: `super.get(n)`

Broj poziva sada je reduciran na samo 5! (primjeri `zi2012_zad3_fibonacci_v2.py`, `zi2012_zad3_fibonacci_v2.cs`)

ZADATAK 4

Razmatramo oblikovanje biblioteke za olakšavanje razvoja jednostavnih prilagođenih HTTP poslužitelja. Biblioteka treba pružiti funkcionalnost za prihvaćanje spajanja mrežnih klijenata, te omogućiti klijentima da konkretnu obradu zahtjeva GET i POST definiraju vlastitim kôdom. Pretpostavljamo da sve zahtjeve treba obraditi isti kôd neovisno o zatraženoj stazi. Predloži organizaciju koja bi klijentima omogućila postizanje opisane funkcionalnosti nasljeđivanjem, i uz što manju količinu vlastitog kôda.

Nemamo nikakve detalje o obradi zahtjeva niti o specifičnostima implementacija metoda koji te zahtjeve obrađuju, ali naglašeno nam je:

1) da obradu vrši jedan te isti kôd neovisno o stazi – Samim time svi klijenti moraju taj kôd naslijediti od zajedničke apstraktne klase
2) da klijentima treba omogućiti definiranje vlastitog kôda za obradu zahtjeva GET i POST – Očito da svi klijenti trebaju implementaciju ovoga pa ćemo ove metode samo deklarirati (kao *pure virtual*) u roditeljskoj klasi, a onda svakoj konkretnoj klasi koja ju nasljeđuje ostavljamo i konkretnu implementaciju.

Možda je do mene, ali nisam na prvu uspio iz zadatka iščitati da se zapravo očekuje korištenje zajedničkog kôda za glavnu obradu, unutar kojeg onda ovisno o konkretnoj implementaciji klase dolazi do pozivanja specifičnosti vezanih za obradu zahtjeva POST i GET.

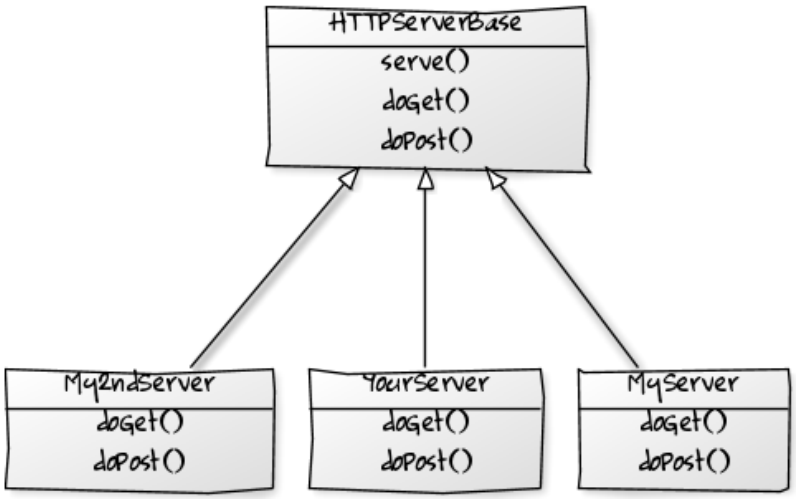
Ring a bell! Imamo kostur kôda koji je zajednički, unutar kojeg treba smjestiti konkretnosti za ove POST i GET dijelove obrade.

```
'''
...
nekakav niz naredbi za koje ne znamo konkretnosti ali su zajedničke za
sve klijente
...
...
obrada Get zahtjeva specifična za svakog klijenta posebno
...
novi komad zajedničkog kôda
...
obrada Post zahtjeva specifična za svakog klijenta posebno
...
'''
```

Okvirna metoda iliti metoda predloška tj. Template method! Nazovimo ju `serve()` i definirajmo u roditeljskoj klasi tako da ju svaki klijent koji tu klasu izvodi naslijedi:

```
class HTTPServerBase:
    def serve(self):
        ''' nekakav niz naredbi za koje ne znamo konkretnosti '''

        print("Iz okvirne metode pozivam metodu \'doGet()\'")
        self.doGet() #!!!

        ''' nekakav niz naredbi za koje ne znamo konkretnosti '''

        print("Iz okvirne metode pozivam metodu \'doPost()\'")
        self.doPost() #!!!
```

U ovoj klasi `HTTPServerBase` samo deklarirajmo metode `doGet()` i `doPost()`, a konkretnu implementaciju prepustimo klijentima kako se od nas i traži. I to je sve!



Napravili smo skicu organizacije u vidu ovog dijagrama, napravili smo programsku skicu roditeljske klase sa definiranom okvirnom metodom, a s obzirom da nemamo nikakve detalje ovi komentari ili ispisi će i više nego poslužit svrsi, jedino što bi još mogli da zaokružimo zadatak jest ponudit programsku skicu nekog od konkretnih klijenata:

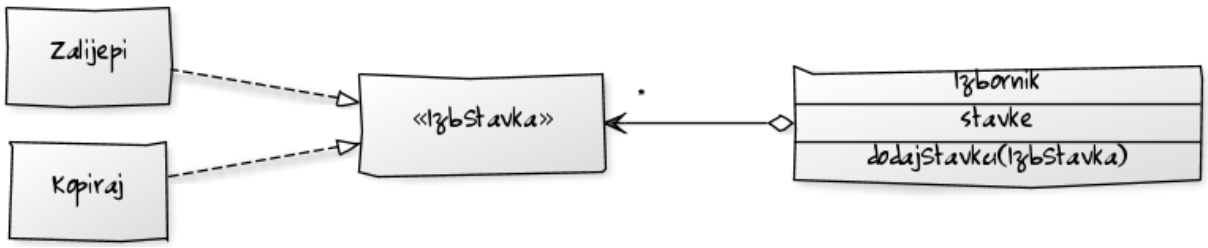
```
class MyServer(HTTPServerBase):

    def doGet(self):
        print("  - Pozvana je metoda za obradu zahtjeva GET (v1)")
        print("  - Obrada načinom definiranim u klasi MyServer -")

    def doPost(self):
        print("  - Pozvana je metoda za obradu zahtjeva POST (v1)")
        print("  - Obrada načinom definiranim u klasi MyServer -")
```

Ponavljam za kraj, Okvirnu metodu `serve()` ćemo nasljediti točno onakvu kakva je implementirana u `HTTPBaseServer`.

ZADATAK 5

Korisnik izrađuje aplikaciju s grafičkim korisničkim sučeljem. U okviru te aplikacije potrebno je podržati sustav izbornika. Korisnik međutim za to želi koristiti ugrađenu biblioteku koja treba omogućiti lagano dodavanje izborničkih stavki i kôda koji te stavke trebaju pokrenuti. Predložite organizaciju takve biblioteke. Koji je oblikovni obrazac prikladan? U vašem rješenju nacrtajte dijagram razreda i naznačite sudionike tog oblikovnog obrasca. Prikažite to na primjeru stavki: "Kopiraj", "Izreži" i "Zalijepi".

Za početak, sve ove izborničke stavke sigurno ćemo obuhvatiti zajedničkim sučeljem. Možemo ga zasada zvati `IzborničkaStavka`, a njega će implementirati `StavkaKopiraj`, `StavkaZalijepi` i sve ostale koje ćemo ikad koristiti. Izbornik koristi izborničke stavke, one ga zapravo sačinjavaju, možemo zaključiti da on ima nekakvu kolekciju sa referencama na konkretne stavke koje poznaje preko njihovog zajedničkog sučelja.



Ovako korak po korak došli smo do organizacije prikazane dijagramom gore. Izbornik preko sučelja nema nikakvog znanja o konkretnostima `Zalijepi` i `Kopiraj`(što nam sasvim odgovara), a isto tako želimo da izbornik nema ovisnosti ni o elementima nad kojima se stavke izvršavaju. Tim prije što nam je naglašeno da koristimo biblioteku pa moramo omogućiti ponovno korištenje. Ne želimo da nam Izbornik ovisi o specifičnostima aplikacije jer bi ga inače za svaku morali ponovno pisati.

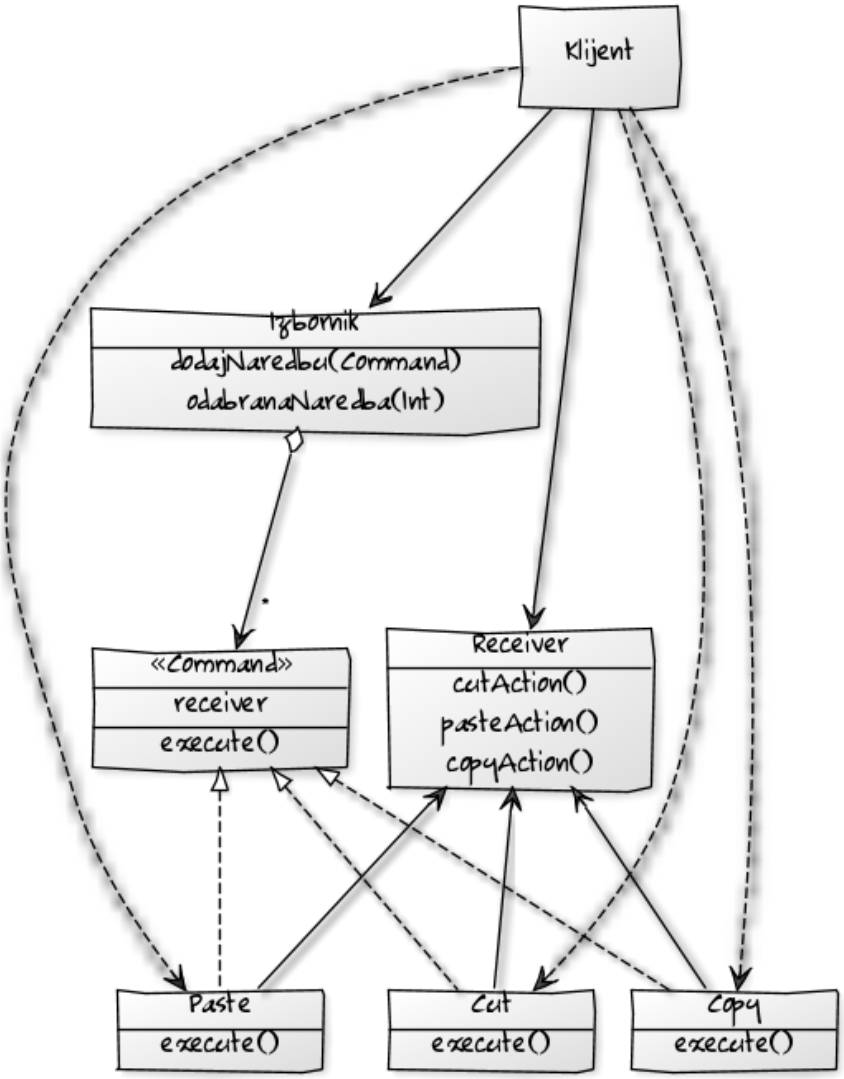
Iskoristit ćemo obrazac Naredbu jer nam organizacija koju „on“ predlaže točno odgovara zahtjevima. Izvršavanje svake od izborničkih stavki (Naredbi) delegirat ćemo elementu o kojem Izbornik nema pojma – Primatelju (*Receiveru*). Izbornik na taj način samo inicira zahtjev za izvršavanjem naredbe preko sučelja koje jedino poznaje. Primatelj zna operacije koje mora obaviti da zadovolji zahtjev. Konkretne naredbe tako odvajaju inicijatora zahtjeva od objekta koji zahtjev izvršava.

Pojam Receivera u kontekstu ovog zadatka ostao mi je malo maglovit na auditornima, tim prije što ga S.Š. nije niti uvrstio na dijagram. Pretpostavljam da nije pogrešna pretpostavka da on ima ima sve potrebne metode kojima konkretne naredbe delegiraju svoje izvršavanje. Recimo ovako nešto (uzevši u obzir da referencu na `Receivera` konkretna naredba primi u konstruktoru i pohrani):

```
class Receiver:
    def copyAction(self):
        #Kopiram
    def pasteAction(self):
        #Lijepim
    def cutAction(self):
        #Izrezujem


class CopyCommand(Command):
    def execute(self):
        self.receiver.copyAction()

class PasteCommand(Command):
    def execute(self):
        self.receiver.pasteAction()

class CutCommand(Command):
    def execute(self):
        self.receiver.cutAction()
```

Klijent kreira naredbe i registrira ih kod pozivatelja – `Izbornika`. On po potrebi poziva konkretnu naredbu preko osnovnog sučelja i ove njene metode `execute()`. Recimo da se taj poziv obavi pozivom metode `odabranaNaredba(n)` kojoj se proslijedi redni broj naredbe koju želimo izvršiti.

Izgenerirani dijagram je malo u banani od nepreglednosti ☹



Da još jednom naznačimo sudionike obrasca:

`Command` je naredba i deklarira jedinstveno sučelje za izvođenje operacija. `Paste`, `Cut` i `Copy` su konkretne naredbe koje implementiraju sučelje naredbe pozivajući primatelja. Primatelj je primatelj – `Receiver`. `Izbornik` je pozivatelj (*invoker*) koji preko sučelja naredbe inicira zahtjev za izvršavanjem. I tu je `Klijent` koji kreira konkretnu naredbu, navodi njenog primatelja i prosljeđuje ju pozivatelju.

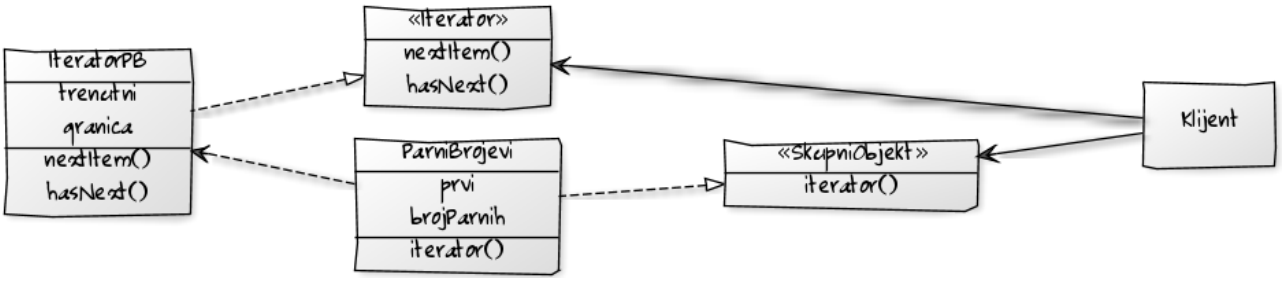
ZADATAK 6

Potrebno je uporabom oblikovnog obrasca Iterator omogućiti "posjetu" elemenata skupnog objekta ParniBrojevi koji predstavlja sortiranu kolekciju uzastopnih parnih brojeva. Neka konstruktor tog razreda prima prvi parni broj te broj uzastopnih parnih brojeva koji pripadaju toj kolekciji. Niti skupni objekt niti iterator ne smiju ni u jednom trenutku doista stvoriti u memoriji čitavu takvu kolekciju. Primjer uporabe u jeziku Java prikazan je u nastavku.

```
ParniBrojevi pb = new ParniBrojevi(2 , 100); // <== ne sprema kolekciju
Iterator <Integer> it = pb.iterator();
```

Predložite cjelokupnu programsku implementaciju svih potrebnih razreda. Iterator treba biti izveden kao korisnički razred. Ispitni program mora uporabom primjerka iteratora ispisati sve elemente skupnog objekta.

Ovaj put nećemo filozofijom odgonetavati kakav obrazac iskoristiti jer nam je to i više nego uočljivo naglašeno u zadatku. Pa napravimo onda klasičnu organizaciju po shemi oblikovnog obrasca Iteratora:



`ParniBrojevi` implementiraju sučelje `SkupnogObjekta`, a `IteratorPB` sučelje `Iteratora`. Metoda `iterator()` skupnog objekta zapravo je metoda tvornice.

```
def iterator(self):
    return IteratorPB(self.prviParni, self.brojUzastopnihParnih)
```

Konkretni iterator neće imati referencu na konkretni skupni objekt baš zbog naglaska da se ova kolekcija brojeva u ni jednom trenutku doista ne stvara u memoriji, pa nam referenca ne bi služila ničemu. Iteratoru u konstruktoru šaljemo argumente koje smo primili pri stvaranju objekta `ParniBrojevi`.

```
class IteratorPB(Iterator):
    def __init__(self, prvi, brUzastopnih):
        self.trenutni = prvi
        self.granica = prvi + 2*brUzastopnih

    def nextItem(self):
        if self.hasNext():
            rv = self.trenutni
            self.trenutni += 2
            return rv
        else:
            raise ValueError

    def hasNext(self):
        return self.trenutni < self.granica
```

ispitni programčić izgledao bi ovako:

```
def test(it):
    while(it.hasNext()):
        print(it.nextItem())

pb = ParniBrojevi(2, 100)
it = pb.iterator()
test(it)
```