„Bitter Java “, Bruce A. Tate (poglavlje 9)

Gorka higijena

Aleksandra Đurić 1079/2015

Sadržaj prezentacije:

- **Zašto treba učiti higijenu programiranja?**
  - Ekstremno programiranje zahteva dobru higijenu
  - Standardi kodiranja štite od antipaterna
- **Mini-antipaterni: Nečitljiv kod**
  - Značenja imena su važna
  - Standardi za imena
  - Zagrade i uvlačenja
  - Komentari
  - Tabovi protiv razmaka
  - Editori
- **Mini-antipaterni: Organizacija i vidljivost**
- **Mini-antipaterni: Struktura**
  - Osnovna objektno-orijentisana filozofija
  - Razmatranje dizajna niskog nivoa
  - Izuzeci
- **Mini-antipaterni: Curenje i performanse**
- **Konvencije za testiranje**
- **Pravljenje vodiča za dobar stil**
  - Kupiti, pozajmiti ili ukrasti?
  - Primer jednostavnog vodiča iz Contextual, Inc.

- U mnogo slučajeva se loša Java povezuje direktno sa lošom formom.
- Dobra higijena programiranja se pridržava čistog poravnavanja, što olakšava deljenje koda i omogućava efikasnije refaktorisanje.
- U okviru higijene programiranja važno je razmotriti konvencije kodiranja i najčešće pravljene greške koje doprinose lošoj higijeni.

Aleksandra Đurić, 1079/2015

- XP (*ekstremno programiranje*) ima mali broj pravila koja omogućavaju moćna poboljšanja u razvojnom programiranju, dobijanju kvaliteta i uspehu samog projekta.

- U XP ceo kod je deljen i programeri moraju da prate striktne konvencije da bi svako ko je u timu mogao lako da pročita bilo koju liniju koda.

- Čitljivost je narušena ako se programeri ne pridržavaju dogovorenih konvencija.

- Zašto treba učiti higijenu programiranja?
  - Ekstremno programiranje zahteva dobru higijenu ←
  - Standardi kodiranja štite od antipaterna

Pravilo XP	Vrednost pravila XP	Zahtevi u higijeni XP
Program napisan po opštim standardima	Standardi su neophodni da bi bilo moguće refaktorisanje u timu. Pogledaj sledeći red.	Dovoljno je rečeno.
Nemilosrdno refaktorisanje	Cena održavanja i popravljanja loše dizajniranog koda smanjuje cenu refaktorisanja.	Refakorisanje često uključuje premeštanje delova koda kroz program. Stil mora da bude uniforman i mora da se pridržava opštih standarda.
Zajedničko vlasništvo nad kompletnim kodom	Pravilno rešenje može biti razvoj bez obzira na to ko je vlasnik, odnosno bez borbe za vlasništvo.	Ako mnogo različitih timova može da pristupa klasama, onda jedino korišćenje opšteg standarda može omogućiti uniformno održavanje stila.
Premeštanje članova tima ukrug	Programeri se bolje fokusiraju i imaju bolju motivaciju. Gubitak jednog člana tima ne osuđuje ceo projekat na propast.	Promena odgovornosti takođe menja skup klasa na koje tim može da utiče .
Programiranje u paru	Omogućava se sagledavanje problema iz različitih uglova i greške se rano uočavaju.	Oba programera moraju da se dogovore oko standarda.
Vrednovanje jednostavnosti	Jednostavno je obično dovoljno, a mnogo je lakše. Vreme razvoja je u tom slučaju kraće.	Mnoge konvencije kodiranja promovišu jednostavnost.

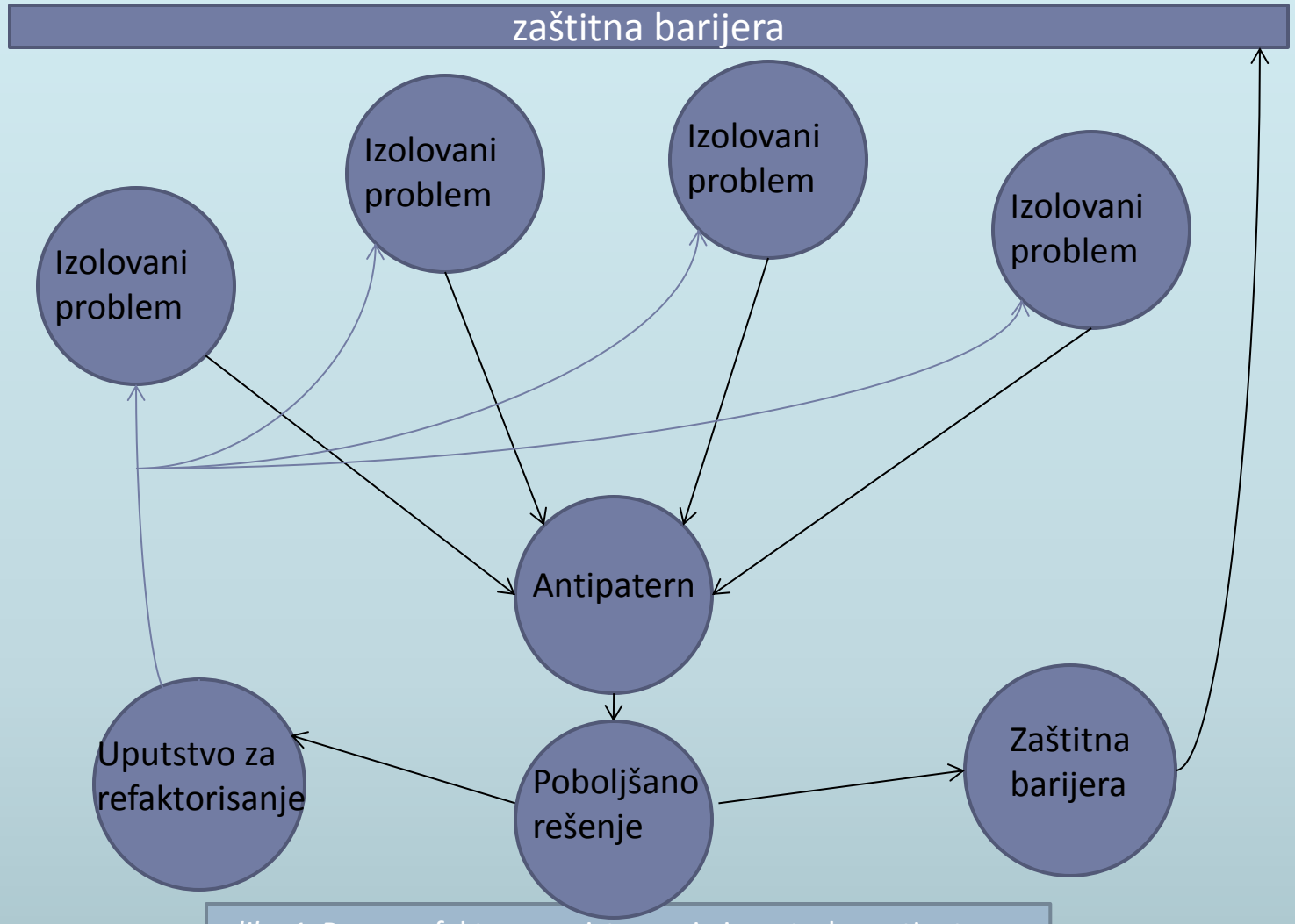
tabela 1: Pravila i zahtevi ekstremnog programiranja

- Protiv svakog pojedinačnog antipaterna se možemo boriti upotrebom adekvatnih standarda za kodiranje. Iz tog razloga se standardi za kodiranje koriste kao oružje protiv antipaterna

- Standardi pokrivaju širok raspon - od formatiranja i stilova do pravila za upotrebu ili načina strukturne organizacije.

Aleksandra Đurić, 1079/2015

- Zašto treba učiti higijenu programiranja?
  - Ekstremno programiranje zahteva dobru higijenu
  - Standardi kodiranja štite od antipaterna ←



slika 1: Proces efektnog registrovanja i upotrebe antipaterna

- Svako kod se bavio nekim značajnijim održavanjem je bio primoran da se bori sa nekonzistentnim stilom, lošim ili nedovoljnim komentarima ili komentarima koji se ne uklapaju sa kodom.

- Tačno je da je čitljivost subjektivna ali potreba za uobičajenim, konzistentnim standardima nije.

Aleksandra Đurić, 1079/2015

- Ako je aplikacija dizajnirana tako da objekti i metode imaju dobra imena dobijamo priču koja nam tačno govori šta se dešava.

Kod koji omogućava dobru komunikaciju pomoću imena

```
purchaseOrder.setCustomer(shoppingCart.getCustomer());
purchaseOrder.setItems(shoppingCart.getItems());
purchaseOrder.totalCost = purchaseOrder.addLineItems() +
                          purchaseOrder.gerTax() +
                          purchaseOrder.getShippingCost();

purchaseOrder.finalizePurchase();
```

Kod sa lošim imenima koje je teško čitati

```
po.setOther(sc.getOther());
po.setVector(sc.getVector(contents));
po.tc = po.addAll() + po.tax() + po.ship();

po.doIt();
```

←

- **Značenje imena** ima najznačajniju uloga u vodiču za imenovanje promenljivih. Nekada se dozvoljava odstupanje jer je lako razumeti šta ime označava.
- **Upotreba velikih slova** se koristi iz dva razloga:
  - za struktuiranje reči – Java koristi konvenciju koja se naziva kamilja notacija za stukturu reči. Prvo slovo svake reči, ne uključujući prvu reč počinje velikim slovom. Izuzetak su akronimi ili skraćenice gde su dozvoljena odstupanja.
  - Za struktuiranje programa

```
Customer                    //klasa
customerFirstName           // atribut
java.lang                   // paket
CUSTOMER_COLUMN             // konstanta
```

Naziv pravila	Primer korišćenja
Za polja i atribute se koriste opisne imenice uz dodatne restriktivne atribute po potrebi	StateTax shippingAddress
Korišćenje deskriptivnog prefiksa sa bulovske atribute (is, contain, has)	if(anItem.isTaxable) { // uradi nešto }

Pristupanje bulovskim poljima je potrebno napraviti kao deskriptivno testiranje (is, has, contains). Get i set nisu preporučeni	person.isPersistant()
	while(table.hasMoreRows) {
	// uradi nešto }

Za metode se koriste opisni glagoli sa dodatnim restriktivnim atributima po potrebi	command.execute() computeInterest()
Ispravno imenovanje metoda za pristupanje poljima je ime polja sa prefiksom get i set	getCustomers() setCustomerName()
Izuzetak: jedno slovo je poželjno i prihvatljivo u petljama zbog ekonomične upotrebe prostora sve dok je čitljivost očuvana	for(i=0; i<10; i++) { // uradi nešto }
Kolekcijama se daje naziv u množini da bi bilo jasnije	shoppingCart.items

tabela 2: Konvencije kodiranja za pravilno definisanje imena

←

- **Rezervisana imena** - Java dozvoljava upotrebu imena koja su rezervisana iako *Sun* strogo odvraća korisnike od ove prakse. Jako lako se može desiti da se zaboravi da je neka reč već rezervisana i da ako se upotrebi ne bude skroz jasno koju od njih je potrebno iskoristiti – onu koju smo mi definisali ili predefinisanu vrednost.
- **Mađarska notacija i domet promenljive** – unutar Java zajednice se vodi debata oko upotrebe Mađarske notacije. Ova konvencija je obično prihvaćena u C++, gde daje neke dodatne informacije o promenljivoj. Često se dodaje oznaka koja govori koji je tip te promenljive. Većina Java programera je izbegava i preferira jednostavniji stil koji teži da poboljša čitljivost koda na engleskom.

```
public class SomeClass
{
        public int value;

        public int getValue()
        {
                return value;
        }

        public void setValue(int value)
        {
                this.value  = value;
        }

        public void readValueFromDatabase()
        {
                int value = 0;
        }

        value = getFromTable(tableName);
}
```

❶ ovde je „value“ atribut

❷ ovde je „value“ lokalna promenljiva

❸ koje „value“ je korišćeno?

Doseg pravila	Primer	Unapređenje	Posledice
Dodavanje _ ispred svih atributa.	Public int _value; _value = 0;	Prevencija kolizije.	Smanjena čitljivost zbog dodavanja nereda u kod.

Dodavanje reference **this** ispred svih atributa.	// deklaracija nepromenjena
	this.value = 0;
		Prevencija kolizije.	Smanjena čitljivost zbog zauzimanja mesta.

Pristup atributima isključivo preko get i set.	// deklaracija nepromenjena setValue (0);	Prevencija kolizije. Izolovani atributi.	Smanjena čitljivost zbog zauzimanja mesta.
Upotreba imena atributa u telu metode je nedozvoljena.	// deklaracija nepromenjena // nepromenjena upotreba	Najčitljivija alternativa.	Podložna koliziji i ljudskoj grešci.

tabela 3: Konvencije kodiranja koje se koriste u prevenciji kolizije između imena atributa i lokalne promenljive

- Standardi propisuju različite načine korišćenja uvlačenja i zagrada. Ovo je oblast gde je verovatno najbolje izabrati standard i držati ga se, ali neka pravila treba da budu više ispoštovana nego druga.

```
if(purchaseOrder.isChargedForShipping())
        totalCost = totalCost + purchaseOrder.addShipping();
        if(totalCost<10)
             totalCost = totalCost+SMALL_ORDER_SUR_CHARGE;
```

- Ovakve greške se mogu izbegavaju korišćenjem **{}** u svakom slučaju kada koristimo **if, while, for**
- Još jedan način za izbegavanje ovakvih slučajnih grešaka je korišćenje dobrog editora.

Aleksandra Đurić, 1079/2015

- Koji stil korišćenja zagrada je najbolji?
  - Prvi stil koji se često koristi je:

```
if(condition) {
        // uradi nešto
}
```

On više ističe okolnosti nego jasnoću. Ovaj stil omogućava da se više linija koda prikaže u određenom trenutku i na taj način olakšava čitljivost.

  - Naredni stil koji se takođe često koristi a propagira jasnoću je:

```
if(condition)
{
        // uradi nešto
}
```

Najvažnije je izabrati jedan standard i držati se njega.

- Još jedna stvar koju treba razmotriti su jasna pravila uvlačenja. Veoma je važno da se sa sigurnošću može reći koje zagrade su uparene. Skoro svaki standard propisuje da se ovo radi pomoću uvlačenja, dok neki standardi propisuju i dodatno korišćenje komentara

```
if(i>10) {
        while(j==4) {
                        // uradi nešto
        } // end while
} // end if
```

Pošto dobri editori mogu da nađu neuparene zagrade neki programeri smatraju da je korišćenje dodatnih komentara nepotrebno.

- Komentari imaju ulogu da poboljšaju čitljivost koda ljudima.
- Dobro pravilo za komentare je da oni treba da opisuju zašto je nešto urađeno, a ne kako je urađeno.
- Za pravljenje proizvodnih aplikacija komentare koristimo da dokumentujemo greške i nejasnoće. Komentari mogu objasniti ili označiti nešto.
- Java podržava tri različita tipa komentara:
  1. blokovski komentari
  2. komentari za dokumentovanje
  3. kratki komentari koji se pišu u jednoj liniji

Aleksandra Đurić, 1079/2015

←

Stil komentara	Primer	Primenljivost
Komentari za dokumentovanje	/** * ClassName * @author: Bruce Tate * /	Koristi se za komentare koji su korisni za kod i automatski se iz njih generiše dokumentacija.
Blokovski komentari	/* step 1 * step 2 * step 3 */	Koristi se za parče koda koje ne treba da se izvršava ili za dokumentovanje dugačkog opisa algoritma za potrebe testiranja.
Kratki komentari u jednoj liniji	i = 1; // a comment	Koristi se za dodavanje anotacije na kraj linije ili za opis promenljive ili same linije koda.

tabela 4: Vrste komentara u jeziku Java

←

- Komentari su takođe podložni greškama, a najčešća je ako su zatvoreni na pogrešnom mestu ili nisu uopšte zatvoreni. Sa ovim problemom takođe pomaže dobar editor.

- Kada je odabran stil koji odgovara potrebama potrebno je odrediti koliko treba komentarisati.

- Ako je linija koda jasna nema potrebe za njenim komentarisanjem. S druge strane ako je otkrivena neka greška ali ne i otklonjena to je neophodno iskomentarisati.

- **Dokumentacija** – JavaDoc može parsirati komentari i napraviti iznenađujuće robusnu dokumentaciju od njih. Očigledna dobit je lako održavanje koda. Prateće poboljšanje je što je lakše održavati ažurnom API dokumentaciju. Treba voditi računa o tome kome je dokumentacija namenjena – programeri koji razvijaju i osobe koje čitaju kod možda imaju različite perspektive.
- **Istorija** – u mnogo slučajeva je važno čuvati prethodnu istoriju koda. Čuvanje istorije može značajno pomoći u sledećim aktivnostima refaktorisanja:
  - uklanjanje koda koji više nije potreban
  - pojednostavljivanje
  - integracija i dekompozicija

  Preporuka je čuvanje koda pomoću nekog sistema za kontrolu verzija. Ovo održava kod konciznim.

- Detalj kao što je izbor korišćenja razmaka ili tabova se čini trivijalan ali sa porastom programiranja unutar tima i deljenjem koda postaje važan. Različiti editori različito posmatraju tabove. Sledeći saveti su korisni za rad u timu:
  - Odaberite standard i pridržavajte ga se. Konzistentnost je najvažnije pravilo.
  - Ako ostale okolnosti nisu poznate preporučeno je korišćenje razmaka. Razmaci se interpretiraju univerzalno, što za tabove ne važi.
  - U editoru se preporučuje čuvanje sa „no-tabs“ opcijom, neki čak i menjaju tabove razmacima u trenutku pisanja.

  Ove male promene mogu pomoći timu u interakciji. Oni takođe omogućavaju lako ubacivanje delova koda u dokumentaciju ili njihovo objavljivanje u nekim publikacijama.

- Izbor editora je subjektivna stvar, ono što je važno je njihova uloga u higijeni kodiranja i čitljivosti koja je istaknuta do sad.

- Ono o čemu je važno razmišljati prilikom izbora editora:
  - Neka razvojna okruženja imaju ugrađene editore.
  - Bolji editori mogu podsticati upotrebu standarda koji se odnose na zagrade i uvlačenja.
  - Dobri editori daju lakši pregled strukture programa.
  - Način na koji editori rukuju tabovima i razmacima je važan.
  - Editori mogu biti integrisani sa razvojnim okruženjem.

```
public SomeClass {
    public int age;
    public SomeClass() {
        // kod koji inicijalizuje godine
    }
}
public AnotherClass {
    public void aMethod() {
        SomeClass anInstance = new SomeClass();
        Person person = new Person();
        person.age = anInstance.age;
    }
}
```

polju age se može pristupati, može se uzeti njegova vrednost, ali može se i menjati

```
public SomeClass {
    private int age;
    public int getAge() {
        return age;
    }
    public SomeClass() {
        // code to correctly initialize age
    }
}
```

bolje rešenje, polju se pristupa preko getAge() i može se jedino uzeti njegova vrednost

```
private Date birthDate;

public Date getBirthDate() {
    return birthDate;
}

public void setBirthDate(Date aBirthDate) {
    this.birthDate = aBirthDate;
}

public int getAge() {
    Date today = new Date();
    int thisYear = today.getYear();
    int birthYear = this.birthDate.getYear();
    int age = thisYear - birthYear;
    if (! hadBirthdayThisYear()) {
        age = age - 1;
    }
    return age;
}
```

atributi su privatni

implementacija za getAge() je sakrivena

```
private boolean hadBirthdayThisYear() {
    Date today = new Date();
    int thisMonth = today.getMonth();
    int birthMonth = this.birthdate.getMonth();
    int thisDay = today.getDate();
    int birthDayOfMonth = this.birthdate.getDate();
    if (birthMonth < thisMonth) {
        return true;
    } else if (birthMonth > thisMonth) {
        return false;
    } else if (birthDayOfMonth <= thisDay) {
        return true;
    }else {
        return false;
    }
}
```

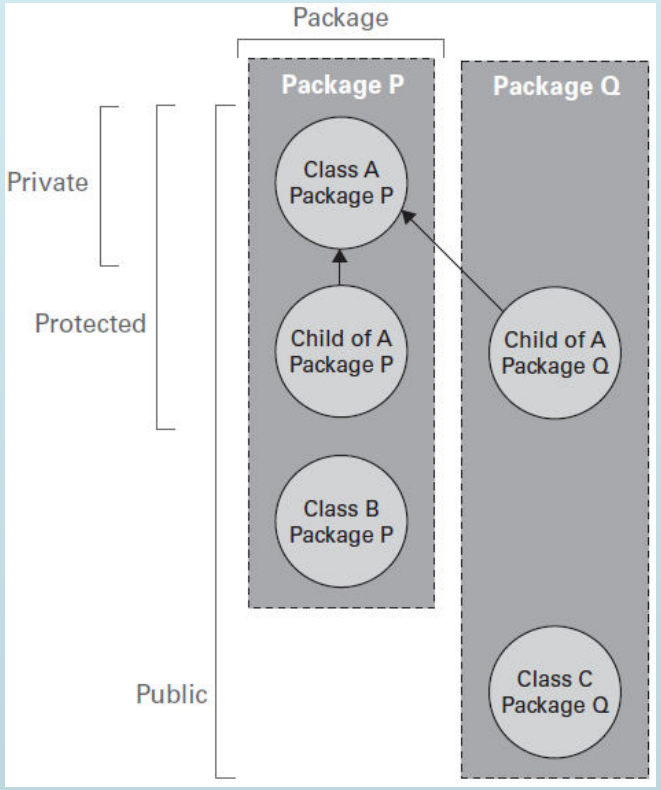
pomoćni elementi su privatni

dobro rešenje, može se po potrebi refaktorisati bez uticaja na kod koji već koristi metodu getAge()

- Razmatraju se standardi koji se tiču privatnosti

Ključna reč	Vidljivost	Upotreba
private	Vidljivost je ograničena na klasu u kojoj su metode i atributi definisani.	Atributi i metode koji se koriste za neka međuizračunavanja i važna su samo kao privremeni rezultat za druga izračunavanja.
protected	Vidljivost je ograničena na klasu u kojoj su definisani i njene potklase.	Atributi i metode koji su neophodni za buduće potklase. Generalno, treba da se promene da budu privatni. Nasleđivanje često koristi pristup na mnogo različitih načina koje je nemoguće unapred predvideti.
public	Javne metode nemaju ograničenje vidljivosti.	Samo metode treba da budu javne. Metode koje su u sastavu interfejsa moraju biti javne.
nedefinisano	Vidljivost je ograničena na paket u kome se nalazi.	Ako se kordiniše upotrebom samo jednog paketa onda je ovakva vidljivost opravdana.

tabela 5: Modifikatori pristupa u jeziku Java



slika 2: Primer koji ilustruje vidljivost u zavisnosti od iskorišćenog modifikatora pristupa

- Loš objektno orijentisan kod dolazi u mnogo različitih oblika.
- Početnici prave univerzalnu grešku – pokušavaju da napišu kod na poznat način.
- Jezik je efektno sredstvo koje olakšava prelazak.
- Dobar objektno orijantisan dizajn zahteva praćenje uputstava koji poboljšavaju jednostavnost koda, čitljivost i omogućavaju ponovnu upotrebljivost.

- Iako standardi ne mogu dovesti do dobrog dizajna njihova upotreba može sprečiti loš dizajn. Smernice za razvoj fleksibilnog dizajna:
  - Generalno, klase treba da budu stvari. Zvuči kao nešto elementarno, ali često se dešava da se pokušava enkapsulacija procesa. U tom slučaju lako se cela funkcionalnost smešta u jednu celinu umesto da bude faktorisana kroz objekte.
  - Metodi treba da budu jedna akcija. Metodi treba da rade jednu jedinu stvar i pri tome da imaju ime koje opisuje njihovu ulogu.
  - Metodi treba da budu kratki i laki za razumevanje.
  - Potrebno je izabrati najjednostavniji način implementacije koji radi. Ovo je tehnika koja doprinosi čitljivosti. Često mnogo više doprinosi brzina njenog razvoja od nekoliko bajtova više koji su joj neophodni.

- Dizajn niskog nivoa karakterišu odluke koje nisu često najjasnije u fazi implementacije.
- Sledi lista najčešćih nejasnoća i smernice za donošenje odluka:

1. **Interfejsi ili apstraktne klase**
   - Interfejsi predstavljaju dogovor o tipovima između pozivaoca i objekta. Apstraktne klase se koriste za obezbeđivanje mehanizama koji su deljeni i zajednički za nekoliko klasa.
   - Interfejsi treba da budu korišćeni svuda gde se može promeniti implementacija.
   - Interfejsi treba da se koriste za opis dodatnih mogućnosti (Printable, Serializable, Cloneable).
   - Interfejsi se koriste kada ne postoji uobičajena implementacija.
   - Apstraktne klase treba da budu korišćene kada je moguće koristiti parcijalnu implementaciju.
   - Nekada se mogu kombinovati interfejsi i apstraktne klase. Ovo nudi mogućnosti parcijalne implementacije sa fleksibilnim interfejsom.

2. **Razmatranje klonabilnog interfejsa**
   - Ako naši objekti treba da imaju mogućnost kloniranja, potrebno je da implementiraju Cloneable interfejs. Ovaj interfejs koristi nekoliko uzoraka za projektovanje. Bolje je uvek ispočetka implementirati kloniranje nego uzeti podrazumevano ponašanje klase Object.
3. **Ekvivalentnosti i heš kodovi**
   - Ako predefinišemo Object.equals, trebalo bi i da predefinišemo Object.hashCodes i obrnuto. Ekvivalentnost je jači test nego heš kodovi pa su zahtevi kod kojih je jedno potrebno a drugo nije retki.
4. **Finalni modifikator**
   - Korišćenje ovog modifikatora zabranjuje ponovnu upotrebu klase, atributa ili metoda i potrebno je posebno voditi računa prilikom njegovog korišćenja. Konvencije savetuju da se final koristi samo za konstante. Alternativno, dozvoljava se njegova pažljiva upotreba samo u slučajevima kada nadklasa definiše interfejs za sve metode.

- Java zahteva staranje o izuzecima koji se mogu pojaviti u programu ali ne postoji način da zahteva i dobro rukovanje izuzecima koji su se pojavili.

- Nekoliko korisnih saveta:
  - Izuzeci treba da pruže logično i predvidivo ponašanje (na primer pogrešno napisano ime fajla ne treba da izazove dramatično pucanje steka jer ovakvo ponašanje nije ni logično ni predvidivo)
  - Izuzecima treba rukovati na nivou gde je sve razloženo i jasno
  - Loša praksa je hvatanje neobrađenih izuzetaka

- **Generalno**
  - Optimizacija se radi kasnije. Treba da bude urađena na samom kraju razvojnog ciklusa. Ako je jasno koji deo koda predstavlja usko grlo on se optimizuje prvi.
- **Curenje i rezervne kopije**
  - Preporučuje se uparivanje alokacije i oslobađanja resursa i implementacija tih metoda što je međusobno bliže moguće
  - U kolekcijama treba osloboditi i iskorišćene objekte. Ovde je takođe dobra praksa ove metode implementirati što je bliže moguće
- **Petlje i pozivanje u krug**
  - Treba obezbediti da kolekcije ne pozivaju jedna drugu u krug
  - Izračunavanja je potrebno čuvati van petlji. Ovo, na primer, nije preporučljivo:

```
for( int i = 0; i<someCollectionSize(); i++)
```

  - Kada se prave stringovi poželjnije je korišćenje string bafera nego nadovezivanje niski.

- **Sinhronizacija**
  - Potrebno je izabrati strategiju i držati se nje.
  - Briga o sinhronizovanosti je velika. Potrebno je koristiti je kada je neophodno, ali nikada se ne sme dodavati sinchronized svim metodama samo za svaki slučaj.
  - Potrebno je razumeti kako Java rukuje sinhronizacijom. Zaštita se radi na nivou objekta a ne na nivou bloka. Atomične funkcije takođe treba da imaju zaštitu jer su u stvarnosti samo mala premeštanja bajtova atomična.
- **Blokiranje**
  - Kad god je moguće dobro je izbegavati blokiranje.
  - Neke aplikacije zahtevaju deljeni pristup za čitanje i ekskluzivni pristup za pisanje. U ovim slučajevima je potrebno koristiti katance za čitanje i pisanje.

- **Saveti:**
  - **Pravljenje testova jedinica koda na početku -** XP prepoznaje značaj ovog pravila. Kodiranje testova omogućava razmatranje izuzetaka u toku dizajna.
  - **Uključivanje main-a prilikom testiranja jedinica koda -** ovo može pomoći sa proverom da li klase odgovaraju zadatoj specifikaciji.
  - **Organizovanje test-slučajeva -** neke organizacije žele testove unutar klasa a neke žele spoljne testove.

Kombinovanjem dobrog dizajna, efektnog organizovanja testova i disciplinom postiže se da klase rade kada i kako treba.

- Nekoliko saveta koje treba razmotriti prilkom pisanja uputstva za stilizovanje:
  - Uputstvo treba da bude kratko.
  - Svako ko ima dodira sa kodom mora da ih pročita.
  - Ponašanje prema upuststvu treba da bude isto kao prema kodu – kada se pokvari potrebno ga je popraviti
  - Potrebno je pridržavati ih se ali postoje opravdani razlozi za odstupanje od njih.
  - Potrebno je napisati ih u skladu sa osobinama i veštinama tima i konkretnim alatom.

- Mnogi konsultanti se bave pravljenjem i prodajom uputstava. Takođe, moguće je naći uputstva besplatno na internetu. Uspešni timovi koriste sledeće načine prilikom pravljenja uputstva:
  - Pravljenje uputstva od nule.
  - Prilagođavanje uputstva koje već postoji – ovo je najčešće korišćen pristup.
  - Kupovina stila i njegova izmena – teško je doneti uputstva o stilizovanju za tim koji se sastoji od programera različitih osobenosti. Preporučuje se izbacivanje neodgovarajućih uputstava pre nego uklapanje uputstava iz različitih izvora.

- **Neka od pravila koja se koriste u Contextual, Inc:**
  - Smeštanje tela za for, if i while unutar vitičastih zagrada čak i kada telo ima samo jednu liniju
  - Davanje opisnih imena promenljivama i prilikom toga nikada ne treba koristiti skraćenice
  - Ne koristiti notaciju sa _ već uvek koristiti kamilju notaciju
  - Kod treba da ima maksimalno 80 karaktera u jednoj liniji
  - Kastovanje treba da izgleda ovako:

    ```
    (Foo) foo
    ```

  - Koristiti sintaksu !!! za komentarisanje dela kome je neophodna poseta kasnije
  - Deklarisanje promenljivih tamo gde su prvi put korišćene, a ne na početku funkcije