

Obvezna načela u dizajnu koda

Ovdje su navedena neka elementarna načela dizajna koda. Pažljivo su odabrana ona koja je lako razumjeti i primijeniti. Stoga će se od studenta očekivati *bezuvjetno* pridržavanje tih načela, a odstupanje od njih smatrat će se ozbiljnom pogreškom i rezultatom nemara.

Načelo 1. Već od prve, nulte, ili koje god sporedne, priručne i neozbiljne verzije koda strogo se držati **pravila imenovanja** paketa, klasa, sučelja, metoda, atributa, parametara i varijabli:

- a. **dozvoljen isključivo engleski jezik**
- b. **svaki identifikator**, bio on i najmizernija pomoćna varijabla, treba biti **smislen i sadržajan**. Npr. imena poput `tmp`, `var1`, `num` ne toleriraju se.
- c. **dozvoljen skup znakova**: engleska slova i znamenke. Klasine konstante (`static final`) koriste još i podvlaku (`_`). Ostali znakovi **zabranjeni**.
- d. **paket**: isključivo mala slova; mora biti potpaket od `hr.fer.tel.ilj08`; primjer: `hr.fer.tel.jsem.statemachine`
- e. **klasa i sučelje**: `UpperCamelCase`
- f. **metoda, varijabla, parametar, atribut**: `lowerCamelCase`
- g. **klasina konstanta**: `UPPER_CASE`

Načelo 2. Primijeniti **stil oblikovanja** koda prema primjeru koji slijedi. Sva ova pravila provodi Eclipse-ovo automatsko oblikovanje koda (`Source | Format`), podrazumijevajući da ste obavili `Import Preferences`, (datoteka `ILJ-EclipsePreferences.epf` iz repozitorija).

```
public class Untitled1
    extends Object
    implements Comparable
{
    public int compareTo( Object something )
    {
        return 1;
    }

    public boolean calcStringValue( int length, String valueToTestFor,
        boolean trim )
        throws Exception
    {
        if ( trim )
        {
            if ( valueToTestFor == null )
            {
                return false;
            }
            else
            {
                valueToTestFor = valueToTestFor.trim();
            }
        }
        else
        {
            try
```

```

    {
        length = 234 / 34 + length - 4;
    }
    catch ( NumberFormatException ex )
    {
        throw (Exception) ex;
    }
}

return true;
}
}

```

Konkretno:

- a. otvorena i zatvorena **vitica** same u **vlastitom** retku
- b. konzistentno **uvlačenje** (indent); korak uvlačenja je 2 znaka
- c. **najviše jedna** izjava po retku
- d. **podređene izjave** od `if`, `for`, `while`, `do` u **retku ispod**, uvučeno. Preferira se i redovito korištenje vitica, čak i za jednu izjavu.
- e. **maksimalna duljina retka: 80** znakova. Nastavak izjave uvući dublje od početka izjave.
- f. svaki **operator** (plus, puta, jednako, itd.) okružiti **razmacima**
- g. sadržaj **oblih zagrada** odvojen od njih **razmakom**: (sadržaj)
- h. **razmak** nakon ključnih riječi `if`, `for`, `while`, `switch`, `catch`. Primjer:
`if (test)`
- i. izjave **throws**, **extends** i **implements** u **vlastitom** retku, dvostruko uvučene.

Vidjeti i Dodatno načelo 1 za daljnja pravila oblikovanja. Njih ne provodi Eclipse.

Načelo 3. U definiciji **sučelja** (`interface`) **ne koristiti** nikakve **modifikatore** metoda. Svaka metoda sučelja je nužno javna i apstraktna, bez obzira napiše li se "`public abstract`" ili ne. Navođenje podrazumijevanih modifikatora zagađuje zapis, a donosi točno nula informacije.

Pogrešno:

```

public interface Node
{
    public abstract void addNode();
}

```

Ispravno:

```

public interface Node
{
    void addNode();
}

```

Načelo 4. **Dostupnost** članova klase uvijek držati na nužnom **minimumu**. To znači:

- a. Svi **atributi** objekta **private**. Pristup atributima omogućiti putem metoda `get/set`. Čak i ako klase iz istog paketa trebaju dostup atributima, to nije opravdanje da im se omogući izravan pristup proširenjem dostupnosti.

- b. Sve **pomoćne metode** (koje pozivaju samo druge metode unutar klase) **private**. Ako metoda treba biti dostupna i u nekoj drugoj klasi istog paketa, treba povećati dostupnost na *package-private*. Posebice: ako se radi o izvedenoj klasi, ali u **istom paketu**, **nije potrebno** dostupnost širiti na *protected*!

Načelo 5. Atributima trenutnog objekta uvijek pristupati putem reference `this`. Ovo znatno poboljšava čitljivost koda, a i osigurava od konflikata s lokalnim varijablama ili parametrima.

Načelo 6. Pažljivo nadjačavati **standardne metode** klase `Object`. Dodatne detalje potražiti u skripti.

- a. `public boolean equals(Object o)`: **parametar** mora biti vrste `Object`.
- b. `public int hashCode()`: obvezno uskladiti njenu implementaciju s metodom `equals()`. Ako se koriste komponente koje se oslanjaju na ovu metodu (npr. `HashMap` i `HashSet`), one će raditi neispravno ako su metode neusklađene.
- c. `public String toString()`: metoda treba vratiti `String` s opisom trenutnog objekta. **Ne smije** ništa **ispisivati** na ekran.

Načelo 7. Korištenje bilo kakvog **tudeg koda** je **zabranjeno**. Posebice, **pokušaj prikrivanja podrijetla koda bit će kažnjen neprolaznom ocjenom**.

Preporučena načela u dizajnu koda

Ovih načela se isto tako treba pridržavati kao i gornjih. Razlika je u tome što se njihovo nepridržavanje neće automatski shvaćati kao rezultat nemara jer mogu biti teža za shvatiti.

Dodatno načelo 1. Daljnja pravila oblikovanja koda (vidi na primjeru uz Načelo 2):

- a. **Sučelje** (`interface`) ima **prednost u dodjeli imena** nad konkretnom klasom. Treba biti najčišće, bez prefiksa (npr. `IList`) ili sufiksa (npr. `ListInterface`). Ime sučelja se koristi često jer se objektima uvijek pristupa putem sučelja koje implementiraju. Ime klase koristi samo za instanciranje objekta. Primjer iz JDK-a: sučelje `List`, konkretne klase `ArrayList` i `LinkedList`.
- b. U deklaraciji polja (u nekim situacijama ga je nemoguće izbjeći) uvijek navesti sve dimenzije uz vrstu podatka, nikad desno od imena varijable.

Pogrešno:

```
int array1d[];  
int[] array2d[];
```

Ispravno:

```
int[] array1d;  
int[][] array2d;
```

Dodatno načelo 2. Brojeвне vrste podatka se smije koristiti samo za podatke sa **smislom broja**. Slijede dva učestala slučaja gdje se ovo načelo krši.

- a. Često se pojavljuje neki podatak koji može poprimiti jednu od vrijednosti iz nekog **manjeg skupa mogućnosti**. Svaka vrijednost ima posebno značenje koje **nema veze s pojmom broja**. Primjer: četiri boje u igraćim kartama: pik, karo, herc, tref. Tendencija u studenata je da za tu namjenu pogrešno koriste `int`-ove sa specijalno dodijeljenim značenjima. Alternativan, također pogrešan pristup je korištenje `String`-konstanti. **Jedini dozvoljen pristup** u takvom slučaju jest Javin *enum*. O njemu se može informirati u skripti.
- b. Čest je pogrešan pristup da se radi **identifikacije** objektu dodaje posebno atribut, najčešće `int`. Objekt je u Javi jedinstveno identificiran svojom **referencom**. Identifikacija posebnim atributom stoga je suvišna.

Pogrešan pristup:

```

public class Node
{
    private int id; // dodatna identifikacija brojem
    private List neighbours;

    public Node( int id )
    {
        this.id = id;
    }

    public void addNeighbour( int neighbourId )
    {
        // u listi susjeda čuvamo njihove id-jeve:
        this.neighbours.add( new Integer( neighbourId ) );
    }

    public boolean equals( Object o )
    {
        ... provjera instanceof ...
        Node otherNode = ( Node ) o;
        // usporedba po id-ju
        return this.id == otherNode.id;
    }
}

```

Ispravan pristup:

```

public class Node
{
    // uklonjeno: private int id;
    private List neighbours;

    // parametar id uklonjen iz konstruktora:
    public Node( )
    {
    }

    // prosljeđujemo referencu na susjeda, a ne njegov id:
    public void addNeighbour( Node neighbour )
    {
        // u listi susjeda čuvamo reference na susjede:
        this.neighbours.add( neighbour );
    }

    // čitava ova metoda je sad nepotrebna jer je ona ovako
    // već implementirana u klasi Object:
    // public boolean equals( Object o )
    // {
    //     return this == o;
    // }
}

```

Kao **dodatni mehanizam identifikacije** može se koristiti `String`-ove, a nikad `int`-ove. Slučajevi kad je ovo potrebno:

- **krajnji korisnik treba identificirati** objekt putem identifikatora (imena)
- **identitet** objekta **treba trajati** dulje od jednog pokretanja aplikacije. Npr. treba ga čuvati u trajnom spremištu (datoteci, bazi)

Vlastiti mehanizam identifikacije mora osigurati da se ne pojave dva ista identifikatora. Izvedba toga nije trivijalna.

Primjer korištenja identifikacije `String`-om, bez prikaza mehanizma za osiguravanje jedinstvenosti:

```
public class Node
{
    private String name;
    private List neighbours;

    public Node( String name )
    {
        this.name = name;
    }

    public String getName( )
    {
        return this.name;
    }
}

// klasa koja koristi klasu Node i omogućuje
// identifikaciju čvora po imenu:
public class Graph
{
    private List allNodes;

    // dohvaća čvor po imenu:
    public Node getNodeByName( String name )
    {
        for ( Iterator iter = this.allNodes.iterator(); iter.hasNext(); )
        {
            Node node = ( Node ) iterator.next();
            if ( node.getName().equals( name ) )
                return node;
        }
        return null;
    }
}
```

Dodatno načelo 3. Vaše klase (koje su dio vježbe) **ne smiju biti izvedene iz tuđih klasa**, čak ni iz JDK-jevih. Na primjer, često u vježbi treba napraviti vlastito spremište nekih objekata. Spremište ima i neke specifične mogućnosti vezane za konkretnu situaciju. Postoji tendencija da se tada napravi svoju klasu izvedenu iz npr. `ArrayList`. To je pogrešan pristup jer `ArrayList` nije dizajnirana za izvod. Pravilan pristup je ubacivanje objekta `ArrayList` u klasu i proslijeđivanje njemu poziva potrebnih metoda.

Pogrešno:

```
public class NoteList extends ArrayList
{
    // listu će se puniti ovom metodom. Dakle, naslijeđena metoda
    // boolean add( Object o ) je suvišna -- kao i mnoge druge.
    public void addNote( Note b )
    {
        this.add( b );
    }
}
```

Ispravno:

```
public class NoteList
{
    // objekt čuva svoju instancu ArrayList-a:
    private List noteList = new ArrayList();

    public void addNote( Note b )
    {
        // koristi ju pozivanjem njenih metoda:
        this.noteList.add( b );
    }
}
```

Dodatno načelo 4. Komponente trebaju biti maksimalno **neovisne** jedna o drugoj i jasno definiranih sučelja:

1. Komponente se koriste isključivo putem svojih **javnih metoda**. Naročito: kôd u nekoj klasi A ne smije izravno pristupati atributima bilo koje druge klase B. Za to treba predvidjeti metode u klasi B.
2. **Funkcionalnost** jedne komponente **ne smije** biti izvedena u nekoj drugoj komponenti.
3. **Ne smiju** postojati **cikličke ovisnosti** između komponenti. Konkretno, ako se u klasi A spominje klasa B, onda se u klasi B ne smije spominjati klasa A. Ovo pravilo uključuje i slučajeve neizravnog korištenja, npr. klasa A koristi B, a ona koristi C. Tada C ne smije koristiti A. Postoje određeni slučajevi u praksi gdje je ovo potrebno, ali studenti to najčešće rade greškom.

Dodatno načelo 5. U **natklasi** se **ne smiju** spominjati njene **potklase**. Natklasa ne smije biti ovisna niti o postojanju konkretnih potklasa, a još manje o njihovim detaljima.

Dodatno načelo 6. **Atributi** objekta služe isključivo za čuvanje njegovog **stanja** koje opstaje između poziva metoda. **Ne smiju** se koristiti kao **pomoćne varijable** za obradu jednog poziva metode. Sve takve varijable trebaju biti lokalne unutar metode. Čak i ako isti podatak treba nizu pomoćnih metoda koje se pozivaju iz glavne, treba ga uredno prosljeđivati svakoj od njih kao parametar.

Pogrešno:

```
public class Repository
{
    private NoteList noteList = new NoteList();
    private Document doc = new Document( new Element("note-list") );

    // ovi atributi očito ne čuvaju stanje objekta. Root je
    // redundantan jer se može uvijek dobiti od doc-a; iterator se
    // isključivo koristi tijekom obrade jednog poziva metode.
    private Element root = doc.getRootElement();
    private Iterator iterator;

    public void getAllNotes()
    {
        for ( this.iterator = this.noteList.iterator();
              this.iterator.hasNext(); )
        {
            Note n = ( Note ) iterator.next();
            ...
        }
    }
}
```

```

    }
    ...
}

```

Ispravno:

```

public class Repository
{
    private NoteList noteList = new NoteList();
    private Document doc = new Document( new Element("note-list") );

    public void getAllNotes()
    {
        // iter je lokalna varijabla unutar petlje for
        for ( Iterator iter = this.noteList.iterator();
              iter.hasNext(); )
        {
            Note n = ( Note ) iterator.next();
            ...
        }
    }

    // kad zatreba, korijenski element se dobavlja od dokumenta
    public Element getRoot()
    {
        return this.doc.getRootElement();
    }
    ...
}

```

Dodatno načelo 7. Općenito treba **izbjegavati preopterećenje metoda**. Ako se već koristi, paziti na zadovoljenost sljedećeg pravila:

Kod metoda s istim brojem parametara, parametri na istoj poziciji ne smiju biti u odnosu potklasa-natklasa. U slučaju kršenja ovog pravila, zbog statičkog odabira pozvane metode neće uvijek doći do poziva one metode koju razvijatelj intuitivno očekuje. O ovome se raspravlja i u skripti, u odjeljku o metodi `equals`. **Pogrešno:**

```

public class List
{
    ...
    public void add( Object o )
    {
        ...
    }
    // String je potklasa od Object
    public void add( String s )
    {
        ...
    }
    ...
}

```


Ispravno (jedna od mogućnosti):

```
public class List
{
    ...
    public void add( Object o )
    {
        ...
    }
    // izbjegavamo preopterećenje:
    public void addString( String s )
    {
        ...
    }
    ...
}
```

Dodatno načelo 8. U vježbi treba definirati **jednu iznimku** i svugdje gdje je potrebno bacati tu jednu iznimku. Osnovne komponente samo bacaju, a nikad **ne hvataju** tu iznimku. Ona treba doprijeti do onog koda koji *koristi* vaše osnovne komponente. U vašem slučaju, to je korisničko sučelje (aplikacijski kod) i testirajući kod.

Pogrešno:

```
public class Playground
{
    ...
    // metoda generira iznimku i baca ju
    private void checkPosition( int posX, int posY )
        throws GameException
    {
        if ( posX < 1 || posY < 1 )
            throw new GameException("Requested position out of bounds");
    }

    // daljnje metode pozivaju checkPosition() i hvataju njenu
    // iznimku

    public void putPlayerAt( int posX, int posY, Object player )
    {
        try
        {
            checkPosition( posX, posY );
        }
        catch ( GameException ex )
        {
            // i kaj sad s njom???
        }
        ...
    }

    public void removePlayerAt( int posX, int posY )
    {
        try
        {
            checkPosition( posX, posY );
        }
        catch ( GameException ex )
        {
            // i što sad s njom???
        }
        ...
    }
}
```

```

    }
}

```

Ispravno:

```

public class Playground
{
    ...

    // samo ova metoda generira iznimku i baca ju
    private void checkPosition( int posX, int posY )
        throws GameException
    {
        if ( posX < 1 || posY < 1 )
            throw new GameException("Requested position out of bounds");
        ... // daljnje provjere
    }

    // daljnje metode samo objavljuju iznimku
    // ako checkPosition() baci iznimku, one ju prešutno ponovo
    // bacaju bez ikakvog eksplicitnog koda koji to izjavljuje

    public void putPlayerAt( int posX, int posY, Object player )
        throws GameException
    {
        checkPosition( posX, posY );
        ...
    }

    public void removePlayerAt( int posX, int posY )
        throws GameException
    {
        checkPosition( posX, posY );
        ...
    }

    // ovdje iznimka ne nastaje čak ni u izravno
    // pozvanim metodama, već u metodi koju one pozivaju
    public void replacePlayerAt( int posX, int posY, Object player )
        throws GameException
    {
        this.removePlayerAt( posX, posY );
        this.putPlayerAt( posX, posY, player );
    }
}

```

Dodatno načelo 9. Vježba ne smije koristiti nikakve **biblioteke** (jar-ove) izvan JDK-a. Ako eventualno netko inzistira na dodatnom jar-u, treba imati jak razlog i tražiti odobrenje. Korištenje dodatnih jar-ova u pravilu je znak krivog pristupa projektu.


Skripta CMD

Prije propitkivanja asistenata informirajte se o naredbenoredačkim ("komandnolinijskim") alatima ovdje: <http://java.sun.com/j2se/1.5.0/docs/tooldocs>

Skriptu treba pokrenuti tako da je trenutni direktorij onaj u kojem se nalazi poddirektorij "src". **Obvezno provjeriti** je li sistemska varijabla **JAVA_HOME** namještena na direktorij gdje je instaliran JDK. Tipična vrijednost je `c:\j2sdk1.5.0`. Namještanje u naredbenom retku (vrijedi samo za trenutni prozor komandnog retka):

```
> set JAVA_HOME=direktorijJDK-a
```

U skriptama se **ne smije** namještat `JAVA_HOME` jer je ona različita od stroja do stroja!

Namještanje za stalno na Windowsima: Kombinacijom tipaka -Break otvori se System Properties. Ode se na Advanced, Environment Variables i doda navedena varijabla.

Kreiranje arhive JAR u Eclipse-u

1. U Package Explorer-u označimo bilo koji paket u našem projektu.
2. File | Export | JAR file...
3. U prozoru koji se pojavi glavni dio zauzima okvir s popisom kazala i datoteka. Naš projekt bi trebao već biti označen kvačicom (sivom). To je ispravno. Na desnoj strani, ako ima nekih datoteka, nijedna ne bi trebala biti označena.
4. Označiti Export generated classes and resources
5. Pod Select the export destination upisati kazalo u koje treba smjestiti JAR.
6. Next, Next
7. Pod Select the class of application entry point odabrati svoju glavnu klasu aplikacije (koja ima odgovarajuću metodu `main`)
8. Finish.

Pokretanje aplikacije, `runapp.cmd`

```
@echo off
echo Starting application
"%JAVA_HOME%\bin\java" -jar app.jar
```