Tvorba informačných systémov 11

Clean Code
Junit

Súhrn odporúčaní

Komentáre
Prostredie
Funkcie
Všeobecné
Java
Názvy
Testy

Upozornenie: ide len o odporúčania, čiže heuristiky o tom, ako pristupovať k čisteniu kódu. Nejde o nejakú úplnú ani záväznú sadu pevných pravidiel. Niektoré preto môžu vyzerať v zdanlivom protiklade, v čistom kóde vždy musíme vyvážiť rôzne priority, ale je potrebné si všetky uvedomovať. Príklady často upozorňujú na jeden aspekt a sú prevzaté z reálneho kódu, čiže môžu obsahovať aj množstvo iných nedostatkov – pokúste sa ich nájsť. :)

K1: nevhodné komentáre

Nemá obsahovať informácie, ktoré majú byť inde: napríklad v systéme pre správu zdrojového kódu, v systéme na sledovanie problémov alebo v podobnom záznamovom systéme. V komentároch by sa nemali objavovať meta-dáta (mená autorov, dátum poslednej modifikácie, číslo problému a pod.) Majú byť použité len na technické poznámky týkajúce sa kódu a jeho návrhu.

K2: zastarané komentáre

Komentár zo starého dátumu, ktorý je irelevantný alebo chybný je zastaraný. Komentáre zastarávajú veľmi rýchlo. Najlepšie je komentáre, ktoré v budúcnosti zastarajú, vôbec nepísať. Ak taký komentár nájdete, buď ho aktualizujte, alebo sa ho čo najskôr zbavte. Zastarané komentáre väčšinou odbiehajú od témy a významu kódu, ktorý kedysi popisovali. Stávajú sa plávajúcimi ostrovmi bezvýznamnosti a zlými ukazovateľmi pri prechádzaní kódu.

K3: prebytočný komentár

Komentár je prebytočný, keď popisuje niečo, čo dostatočne popisuje samé seba. Napr.

```
  i++;  // increment i
```

Podobne dokumentačné komentáre, ktoré nevravia nič viac (alebo ešte menej), ako obsahuje signatúra funkcie:

```
/**
* @param sellRequest
* @return
* @throws ManagedComponentException
*/
public SellResponse beginSellItem(SellRequest sellRequest)
throws ManagedComponentException
```

Komentáre by mali opisovať veci, ktoré samotný kód vyjadriť nemôže.

K4: Zle napísaný komentár

Ak už nejaký komentár píšeme, treba, aby bol najlepší, ako napísať vieme. Starostlivo vyberáme slová a formulácie, dbáme na gramatiku a pravopis, píšeme súvislé vety, nepíšeme o samozrejmostiach, sme čo najstručnejší.

K5: Zakomentovaný kód

O zakomentovanom kóde je niekedy ťažké povedať ako je starý, či je na niečo dobrý, nikto ho nevymaže, lebo každý predpokladá, že ho potrebuje niekto iný. Tento kód zotrváva, degeneruje a zo dňa na deň sa stáva menej relevantný. Volá funkcie, ktoré už neexistujú. Používa premenné, ktorým sa už zmenili názvy, riadi sa konvenciami, ktoré medzi tým zastarali. Zaplňuje moduly, zhoršuje čitateľnosť kódu. Ak vidíte zakomentovaný kód, zmažte ho. Ak ho ktokoľvek bude potrebovať, môže sa vrátiť k predchádzajúcej verzii v správe zdrojových súborov.

P1: Zostavenie vyžaduje viac ako jeden krok

Zozstavenie projektu (kompilácia, linkovanie...) by malo byť jednoduché a triviálne. Nie je dobre, ak musíte ručne vyberať jednotlivé časti zo systému pre správu zdrojového kódu. Nemali by ste potrebovať sekvencie tajných príkazov alebo kontextovo závislých skriptov určených na zostavenie jednotlivých prvkov. Nemalo by sa vyžadovať vyhladávanie rôznych ďalších súborov JAR, XML a iných knižníc, ktoré systém vyžaduje. Malo by byť možné program vybrať jedným príkazom a pomocou ďalšieho ho zostaviť.

```
svn get mySystemInJava
cd mySystem
ant all
```

```
svn get myCPPSystem
cd myCPPSystem
make all
```

P2: Testy vyžadujú viac ako jeden krok

Všetky jednotkové testy by sa mali dať spustiť jediným príkazom. V ideálnom prípade tieto testy spustíte z IDE jediným tlačidlom, alebo jedným jednoduchým príkazom. Schopnosť spustiť všetky testy je natoľko zásadná a dôležitá, že by to malo byť rýchle, jednoduché a samozrejmé.

F1: priveľa argumentov

Funkcia by mala mať malý počet argumentov. Najlepšia je funkcia bez argumentov, potom jeden, dva, alebo tri argumenty. Viac ako tri argumenty je velmi problematické a mali by sme sa tomu bez váhania vyhýbať.

F2: Výstupné argumenty

Výstupné argumenty nezodpovedajú ľudskej intuícii. Čitateľ programu očakáva, že argumenty budú výstupné, nie vstupné. Ak funkcia musí zmeniť nejaký stav, nech mení stav objektu, na ktorom je volaná.

F3: Logické argumenty

Logické argumenty signalizujú, že funkcia robí viac ako jednu činnosť. Sú mätúce a nemali by sa používať. (myslia sa tým argumenty typu boolean, ktoré sú len „flagy“, ktoré určujú čo má funkcia robiť).

F4: mŕtve funkcie

Metódy, ktoré nikto nevolá, by sa mali zrušiť. Udržovať nepoužívaný kód je nákladné. Nebojte sa takéto funkcie zmazať. Majte na pamäti, že váš systém pre správu zdrojového kódu ich aj tak bude niekde mať uložené.

V1: Viac jazykov v jednom zdrojovom súbore

Dnešné moderné vývojové prostredia umožňujú umiestniť do jedného zdrojového súboru úseky s viacerými jazykmi. Napríklad javovský zdrojový súbor môže obsahovat kód v XML, HTML, YAML, dokumentáciu pre JavaDoc, angličtinu, zdrojový kód v JavaScripte, atď. Iný príklad: súbor HTML a JSP môže obsahovať Javu, syntax knižničných tag-ov, atď. Je to mätúce, niekedy ľahkomyselné alebo nedbalé. Ideálne je, ak zdrojový kód obsahuje iba jeden jazyk. V praxi často musíme použiť viaceré jazyky, ale mali by sme sa snažiť ich počet a rozsah minimalizovať.

V2: Nie je implementované to, čo je samozrejmé

„Princíp najmenšieho prekvapenia“ radí, aby každá funkcia a trieda implementovala (robila) to, čo by čitateľ programu, kde je použitá, mohol očakávať. Napríklad nasledujúci riadok:

```
Day day = DayDate.StringToDay(String dayName);
```

prevádza názvy dní na konštanty. Mohli by sme predpokladať, že napr. reťazec „Monday“ bude prevádzať na Day.MONDAY. Podobne by sme mohli očakávať, že bude akceptovať používané skratky dní a nebude robiť rozdiely medzi malými a veľkými písmenami. Ak očakávaná funkčnosť nie je implementovaná, čitatelia programu sa nemôžu spoľahnúť na autora ale musia stále podrobne kontrolovať pôvodný kód.

V3: Nekorektná funkčnosť v hraničných prípadoch

Niekedy si neuvedomujeme, ako komplikované je korektné správanie sa kódu. Vývojári často píšu funkcie o ktorých si myslia, že budú fungovať, spoľahnú sa na svoju intuíciu bez toho, aby vynaložili námahu a overili si, či ich kód funguje aj v mimoriadnych situáciách a v hraničných prípadoch. Potrebné úsilie sa nedá ničím nahradiť. Každá hraničná podmienka a mimoriadny prípad, či zvláštne správanie predstavujú nebezpečenstvo, ktoré môže akokoľvek elegantný a intuitívny algoritmus znehodnotiť. Nespoliehajte sa na intuíciu. Preverte každú hraničnú podmienku a napíšte pre ňu testy.

V4: Zrušené zabezpečenia

Rušenie bezpečnostných opatrení je riskantné. Použitie ručného nastavenia serialVersionUID môže byť nevyhnutné, ale vždy bude znamenať riziko. Vypnutie varovaní prekladača (alebo dokonca všetkých) môže síce znamenať, že sa program preloží, ale za cenu nekonečného počtu ladiacich cyklov. Vypnúť kontrolné testy a veriť, že prejdú neskôr je podobné ako predstierať, že kreditná karta sú hotové peniaze.

V5: Zdvojenie

Zádada o škodlivosti zdvojeného kódu je jedna z najdôležitejších a treba ju brať vážne (DRY = „Don't repeat yourself“; zásada extrémneho programovania „Raz a len raz“).
Zdvojenie kódu pravedpodobne znamená prehliadnutie možnosti nejakej abstrakcie. Zdvojenie sa často môže zmeniť v procedúru alebo triedu. Prevedenie zdvojenia na abstrakciu rozšíri jazyk návrhu. Vytvorené abstraktné prostriedky môžu využiť ďalší programátori, kódovanie bude rýchlejšie a menej chybové, lebo sa zvýšila úroveň abstrakcie. Príklady: úplne identický kód, opakujúce sa podmienky v swich/case alebo if/else, podobné algoritmy, ale odlišný kód – dá sa riešiť pomocou templates, alebo návrhových vzorov; vo svete databáz: možnosť využiť normálne formy.

V6: Kód na zlej úrovni abstrakcie

Abstrakcia oddeľuje kód na vyššej úrovni od detailov nižšej úrovne. Napr. abstraktné triedy obsahujú pojmy vyššej úrovne a odvodené triedy obsahujú pojmy nižšej úrovne. Pri tomto rozdelení by sme mali byť dôslední a všetky pojmy vyššej úrovne by mali byť v základných a všetky pojmy nižšej úrovne v odvodených triedach. Konštanty, premenné a pomocné funkcie, ktoré patria k detailnej implementácii by nemali byť v základnej triede. Základná trieda by o nich nemala nič vedieť. Toto pravidlo platí nielen pre triedy, ale aj pre zdrojové súbory, komponenty a moduly. Separácia má byť čo najúplnejšia.

V6: Kód na zlej úrovni abstrakcie

Príklad:

```
public interface Stack {
  Object pop() throws EmptyException;
  void push(Object o) throws FullException;
  double percentFull();
  class EmptyException extends Exception { }
  class FullException extends Exception { }
}
```

Metóda `percentFull()` je na nesprávnej úrovni abstrakcie. Hoci existujú rôzne implementácie rozhrania `Stack`, kde pojem *fullness* má zmysel, existujú iné, ktoré jednoducho nemajú žiadnu informáciu o tom ako sú zaplnené. Preto by bolo lepšie umiestniť ju do odvodeného rozhrania, napr. `BoundedStack`.

Poznámka: ak by v neobmedzenom zásobníku táto metóda vracala napr. 0, tak by niektoré programy jednoducho zlyhali a nefungovali...

V7: Základné triedy závisia na odvodených triedach

Všeobecným dôvodom na rozdelenie pojmov medzi základné a oddelené triedy je, aby pojmy vyššej úrovne v základnej triede nezáviseli na pojmoch nižšej úrovne v odvodených triedach. Čiže ak základné triedy obsahujú názvy svojich odvodených tried, pravdepodobne niečo nie je v poriadku. Základné triedy by nemali o svojich odvodených triedach nič vedieť. Rozdelenie by malo platiť aj v knižničných súboroch jar. Základné triedy môžu byť v jednom a odvodené v druhom. Druhý – s odvodenými triedami – sa potom môže ladiť a meniť bez toho, aby sa menil prvý z nich.

V8: Priveľa informácií

Dobre definované moduly majú malé rozhrania, ktoré umožnia „za málo peňazí veľa muziky“. Zle definované rozhrania majú rozsiahle a zložité rozhrania, ktoré nútia používať na jednoduché veci množstvo krokov. Dobre definované rozhrania ponúkajú len zopár funkcií, takže počet väzieb je nízky. Čím menej má trieda metód, tým lepšie, čím menej má trieda premenných, tým lepšie. Skrývajte dáta, pomocné funkcie, konštatny a dočasné premenné. Nevytvárajte triedy s množstvom metód alebo premenných. Snažte sa obmedziť počet „protected“ premenných a metód. Snažte sa o kompaktné rozhrania.

V9: Mŕtvy kód

Mrtvý kód je taký kód, ktorý sa nevykonáva. Napr. v tele podmieneného príkazu s podmienkou, ktorá nikdy nemôže byť splnená, alebo v bloku catch príkazu try, v ktorého tele nie je žiaden príkaz throw. Pomocné metódy, ktoré sa nikdy nevolajú. Mŕtvy kód sa stáva nositeľom skrytých problémov. Čím je starší, tým horšie. Pri zmene návrhu sa môže zabudnúť na jeho aktualizáciu. Vznikal v dobe, keď bol systém iný. Keď nájdete mŕtvy kód, vymažte ho.

V10: vertikálne oddelovanie

Premenné a funkcie by mali byť definované blízko miesta, kde sa používajú. Lokálne premenné by mali byť deklarované tesne pred prvým použitím a mali by sa používať v malom vertikálnom rozsahu. Nechceme deklarovať lokálne premenné stovky riadkov ďaleko od miesta, kde sa používajú. Súkromné funkcie by mali byť definované hneď za svojím prvým použitím. Patria do rozsahu celej triedy, ale aj tak je správne obmedziť vertikálnu vzdialenosť medzi definíciou funkcie a jej volaním. Nájdenie súkromnej funkcie by malo byť len otázkou letmého pohľadu smerom dole od prvého použitia.

V11: Nekonzistentnosť

Ak niečo robíte jedným spôsobom, robte tak aj všetko ostatné. Súhlasí to s princípom najmenšieho prekvapenia. Konvencie si vyberajte opatrne, ale potom sa nimi starostlivo riaďte. Napr. ak hodnotu typu `HttpServletResponse` ukladáte do premennej nazvanej response, konzistentne používajte premennú s rovnakým názvom aj na ďalších takýchto miestach. Ak sa metóda volá `processVerificationRequest`, tak podobná metóda, ktorá robí niečo súvisiace by sa mala volať podobne, napr. `processDeletionRequest`.

V12: Neporiadok

Implicitný konštruktor, ktorý nie je implementovaný, alebo premenné, ktoré sa nepoužívajú, funkcie, ktoré sa nevolajú, poznámky, ktoré neposkytujú žiadne informácie a podobne, to všetko len prídava k neporiadku v kóde a predstavuje odpadky, ktoré by sa mali odstrániť. Udržujte kód čistý, dobre organizovaný a bez odpadkov.

V13: Umelé väzby

Nevytvárajte umelé väzby medzi entitami, ktoré od seba nazávisia. Napríklad všeobecné vymenované typy by nemali byť v špecifickejších triedach, lebo to núti celú aplikáciu k tomu, aby tieto triedy používala. Rovnako sú nevhodné statické funkcie na všeobecné použitie, ktoré sú deklarované v špecifických triedach.
Umelá väzba je väzba medzi dvoma modulmi, ktoré neplnia nejaký priamy účel, je to výsledok ukladania premenných, konštánt, alebo funkcií na miesto, ktoré je vhodné len dočasne, je to výsledok lenivosti a nedbalosti. Venujte čas zváženiu, kde sa majú funkcie, konštanty a premenné deklarovať.

V14: Chýbajúce schopnosti

V situácii, keď jedna metóda používa prístupové metódy a mutátory iných objektov, aby mohla pracovať s ich dátami je zrejme umiestnená nesprávne a mala by sa skôr nachádzať vnútri v danom objekte a mala priamy prístup k týmto premenným. Inak sa zbytočne komplikuje rozhranie a vystavujú sa dáta objektu, ktoré by mohli ostať skryté. Niekedy však máme iné dôvody, pre ktoré aj takýto návrh môže byť najvhodnejším možným, „chýbajúce schopnosti“ objektov treba však cieľavedome minimalizovať.

V15: Prepínače v argumentoch

O argumentoch true/false na konci volania nejakej funkcie už bola reč – zo zápisu volania nie je vôbec jasné, čo daný argument znamená, program je tým pádom nečitateľný. Naviac takto volaná funkcia nerobí jednu vec, ale viac rôznych vecí. Takéto funkcie treba rozdeliť na viacero menších.

V16: Nejasný zámer

Kód by mal byť čo najexpresívnejší (čo najlepšie vyjadrovať, čo robí). Výrazy bez medzier, maďarská notácia názvov, magické čísla, to všetko autorov zámer zatemňuje. Príklad nevhodnej funkcie:

```
public int m_otCalc() {
  return iThsWkd * ithsRte +
    (int) Math.round(0.5 * iThsRte *
      Math.max(0, iThsWkd - 400)
    );
}
```

Ekvivalentný, dlhší, ale prehľadnejší kód môže vyzerať takto:

```
public int straightPay() {
  return getTenthsWorked() * getTenthRate();
}
public int overTimePay() {
  int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
  int overTimePay = overTimeBonus(overTimeTenths);
  return straightPay() + overTimePay;
}
private int overTimeBonus(int overTimeTenths) {
  double bonus = 0.5 * getTenthRate() * overTimeTenths;
  return (int) Math.round(bonus);
}
```

V17: Zle umiestnená zodpovednosť

Najdôležitejšími rozhodnutiami, ktoré softvérový vývojár robí, sú rozhodnutia, kam umiestniť kód. Typický príklad: má byť konštanta PI umiestnená v triede Math, Trigonometry, alebo Circle?

Kód by sa mal umiestňovať tam, kde ho čitateľ programu bude najskôr očakávať, nie nutne tam, kde sa nám to (zdanlivo) najlepšie hodí.

V18: Nevhodný modifikátor static

V prípade, že nikdy nebudeme potrebovať, aby metóda bola polymorfná a ak nie je dôvod, aby metóda pracovala nad nejakou inštanciou, môže byť statická. Typický príklad: Math.max(double a, double b). Bolo by nezmyselné, ak by sme ju museli volať takto: new Math90.max(a,b). V prípade, že funkcia získava údaje len zo svojich argumentov, ale v budúcnosti by sa nám mohlo hodiť, aby bola polymorfná, je vhodnejšie, aby nebola statická. Vo všeobecnosti by sa mali nestatciké metódy uprednostňovať pred statickými.

V19: Používajte vysvetľujúce premenné

Program sa stane prehľadnejší, ak sa výpočet rozdelí na niekoľko medzivýsledkov, ktoré sa uložia do premenných s vysvetľujúcimi názvami. Tu je príklad vhodného rozdelenia (namiesto toho, aby sa priamo do volania metódy put() použili nepomenované medzivýsledky:

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
  String key = match.group(1);
  String value = match.group(2);
  headers.put(key.toLowerCase(), value);
}
```

V20: Názvy funkcií by mali vravieť, čo robia

Príklad nesprávneho pomenovania:

```
Date newDate = date.add(5);
```

Z tohto kódu nie je jasné, či funkcia k dátumu pripočíta 5 týždňov, hodín, či dní. Nie je jasné, či metóda vráti objekt, na ktorom sa volá a zmení ho, alebo vráti novo vytvorený objekt bez zmeny pôvodného. Na základe tohto volanie nevieme povedať, čo táto funkcia robí. Vhodnejšie názvy by boli `addDaysTo` alebo `increaseByDays`. Ak sa treba pozrieť do implementácie funkcie, aby sme zistili, čo robí, treba jej vymyslieť lepšie meno, alebo zmeniť jej funkcionalitu tak, aby sa dala lepšie pomenovať.

V21: Pochopte algoritmus

Pred tým, ako si začnete myslieť, že je funkcia hotová, overte si, že *rozumiete* tomu, ako funguje. Nestačí, že funkcia prejde všetkými testami. Musíte *vedieť*, že riešenie je korektné.

V22: Urobte z logických závislostí fyzické

Ak jeden modul závisí na inom, táto závislosť by mala byť fyzická, nie len logická. Závislý modul by nemal mať nároky na modul, na ktorom závisí – nemal by vytvárať logické závislosti. Namiesto toho by tento modul mal explicitne žiadať o všetky informácie, na ktorých závisí. V knižke je uvedený príklad s dvoma triedami. Jedna trieda O vytvára obsah pre výstupnú zostavu, druhá trieda F zodpovedá za výstupné naformátovanie strany (napr. pridanie hlavičky a päty stránky a pod.). Spor je o konštante PAGE_SIZE. Ak je umiestnená v triede O, ktorá vytvára obsah a po danom počte riadkov zavolá naformátovanie stránky cez metódu triedy F, vytvára sa takto logická závislosť. Trieda O nemôže poznať obmedzenia na veľkosť stránky, o tom vie iba trieda F. Trieda O by sa teda mala opýtať triedy F na vhodnú dĺžku stránky.

V23: Voľte radšej polymorfizmus ako príkazy if/else, switch/case

Ak sa rovnako členený príkaz switch vyskytuje na viacerých miestach v programe a jednotlivé vetvy case príkazu switch môžu vytvoriť polymorfné objekty, mali by sme dať prednosť polymorfizmu pred použítim príkazov switch, resp. if-else.

V24: Dodržujte standardné konvencie

Každý tím by mal dodžiavať normy na písanie kódu. Štandard by mal špecifikovať kde deklarovať inštančné premenné, ako pomenovať triedy, metódy, premenné, kde umiestňovať zátvorky atď. Tím by mal vystačiť so samotným kódom, ktorý je príkladom štandardu aj bez explicitnej dokumentácie. Každý člen tímu by konvencie mal dodržiavať.

V25: Nahraďte magické čísla pomenovanými konštantami

V ktorých prípadoch používať číselné (alebo iné) konštanty a kedy ich pomenovať? Napr. vo výrazoch

```
double milesWalked = feetWalked / 5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

Je význam konštánt 5280, 8 a 2 jednoznačný a zrozumiteľný a ich pomenovaním by sa čitateľnosť kódu nezlepšila. Naopak, dlhé a zložité konštanty ako je číslo PI treba pomenovať, lebo sa v nich ľahko spravia chyby a chceme, aby sa vo všetkých častiach programu používalo toto číslo s rovnakou presnosťou. Ak niekde používame špeciálne konštanty, ktoré majú určitý skrytý význam, treba ich tiež pomenovať (magické konštanty) a meno by malo vyjadrovať tento význam (napr. referenčný záznam v dátabáze).

V26: Buďte presní

Snažme sa v každej situácii použiť najvhodnejší nástroj. Napr. prvý záznam zoznamu nemusí byť jediný záznam, vyhýbanie sa uzamykaniu a transakciám len preto, že si myslíme, že nemôže dôjsť k súbežnej aktualizácii je lenivosť a je to aj nebezpečné. Použiť premennú typu ArrayList na mieste, kde by stačil typ List je priveľmi obmedzujúce. Označovať všetky premenné ako protected je zasa priveľmi otvorené. Každé rozhodnutie v kóde by malo byť adekvátne danej situácii. Dbajte na detaily. Napr. ak voláte funkciu, ktorá môže vrátiť null, vždy skontrolujte návratovú hodnotu. Vyhnite sa dvojznačnosti a nepresnostiam.

V27: Štruktúra je viac ako konvencia

Konvencie na tvorbu názvov sú dôležité, ale nie sú až tak dôležité, aby mali prednosť pred dôvodmi vyplývajúcimi z návrhu a vhodnej štruktúry programu. Konvencie pomenovania sú významnejšie na vyšších úrovniach abstrakcie – ako sú abstraktné metódy, základné triedy, menej už na nízkej úrovni v jednotlivých príkazoch napr. switch/case.

V28: Zapúzdrite podmienky

Kód s logickými výrazmi je ťažšie zrozumiteľný, zvlášť ak ich musíte sledovať v ich špecifickom kontexte. Zložité podmienky schovajte do funkcií. Napríklad:

```
if (shouldBeDeleted(timer))
```

je lepšie ako

```
if (timer.hasExpired() && !timer.isRecurrent())
```

V29: Vyhýbajte sa negatívnym podmienenému výrazom

Negatívne podmienené výrazy sú menej zrozumiteľné ako pozitívne. Použite a vytvárajte ich vždy, keď je to možné. Napríklad:

```
if (buffer.shouldCompact())
```

je lepšie ako

```
if (!buffer.shouldNotCompact())
```

V30: Funkcie by mali robiť len 1 vec

Niekedy je lákavé vytvoriť funkcie, ktoré robia viac operácií.
Funkcie tohto druhu robia viac ako jednu vec, mali by sa rozdeliť.
Príklad:

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
        e.deliverPay(pay);
    }
  }
}
```

možno rozdeliť takto:

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}
private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}
private void calcucateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
    e.deliverPay(pay);
}
```

V31: Skrytá časová väzba

Ak je časová postupnosť volaní funkcií nevyhnutná, nemala by byť skrytá. Napríklad v nasledujúcom kóde nie je postupnosť dosť názorná:

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;
  public void dive(String reason) {
    SaturateGradient();
    RetuclateSplines();
    diveForMoog(reason);
  }
  …
}
```

namiesto toho použime radšej:

```
  …
    Gradient gradient = saturateGradient();
    List<Spline> splines = reticulateSplines(gradient);
    diveForMoog(splines, reason);
  }
  …
```

V32: Nepodliehajte ľubovôli

Zdôvodnite si štruktúru svojho kódu a presvedčte sa, že tieto dôvody v jeho štruktúre naozaj vidno. Ak štruktúra kódu vyzerá ako niečo ľubovoľné, ostatní programátori budú mať nutkanie naďalej ju ľubovoľne meniť. Ak v celom programe vyzerá štruktúra konzistentne, budú ju používať i ostatní a budú zachovávať konvencie.

V33: Zapúzdrite podmienky hraničných prípadov

Podmienky hraničných prípadov sa veľmi ťažko sledujú. Kód, ktorý ich spracováva patrí na jedno miesto. Nepotrebujeme množstvo plus/mínus jednotiek roztrúsených po celom kóde. Príklad:

```
if (level + 1 < tags.length)
{
  parts = new Parse(body, tags, level + 1, offset + endTag);
  body = null;
}
```

Výraz `level + 1` je podmienka hraničného prípadu, ktorú môžeme zapúzdriť do premennej:

```
int nextLevel = level + 1;
if (nextLevel < tags.length)
{
  parts = new Parse(body, tags, nextLevel, offset + endTag);
  body = null;
}
```

V34: Funkcie by mali zostupovať len o jednu úroveň abstrakcie nižšie

Všetky príkazy vo funkcii by mali byť na rovnakej úrovni abstrakcie, ktorá by mala byť o jednu úroveň nižšie ako operácia, ktorú popisuje názov funkcie. Toto je na pohľad jednoduchá myšlienka, ale ťažko sa dôsledne dodržiava. Príklad:

```
public String render() throws Exception
{
  StringBuffer html = new StringBuffer("<hr");
  if (size > 0)
    html.append(" size=\"").append(size + 1).append("\"");
  html.append(">");
  return html.toString();
}
```

Môžeme prepísať takto:

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (extraDashes > 0)
    hr.addAttribute("size", hrSize(extraDashes));
  return hr.html();
}
private String hrSize(int height)
{
  int hrSize = height + 1;
  return String.format("%d", hrSize);
}
```

V35: Používajte konfiguračné dáta na vysokých úrovniach abstrakcie

Konštanty a konfiguračné hodnoty, ktoré sú známe a očakávame, že budú vo vyšších úrovniach abstracie neskrývajme v nižších úrovniach. Typicky sa takéto hodnoty posielajú v argumentoch z funkcie na vyššej úrovni do funkcie na nižšej úrovni, napr.

```
public static void main(String[] args) throws Exception
{
 Arguments arguments = parseCommandLine(args);
 …
}
public class Arguments
{
  public static final String DEFAULT_PATH = ".";
  public static final String DEFAULT_ROOT = "XXX";
  public static final int DEFAULT_PORT = 80;
  …
}
```

V36: Vyhýbajte sa tranzitívnym odkazom

Moduly by nemali mať priveľa informácií o svojich spolupracovníkoch. Ak modul A spolupracuje s modulom B a modul B spolupracuje s C, nechceme, aby moduly, ktoré používajú A vedeli o module C. Nechceme napr. volať metódu takto:

```
a.getB().getC().doSomething();
```

V takom prípade bude napr. komplikované neskôr vložiť modul Q medzi moduly B a C. Namiesto toho by spolupracujúce moduly mali poskytovať všetky služby, ktoré potrebujeme:

```
myCollaborator.doSomething();
```

J1: Vyhýbajte sa dlhým zoznamom a používajte wildcards

Ak importujeme viacero tried z jedného balíka, neuvádzajme zbytočne množstvo riadkov importujúcich po jednej triede, ale importujte celú skupinu tried použitím napr. import package.*;

J2: Vyhýbajte sa dedeniu konštánt

Niekedy by sme mali chuť použiť mechanizmus dedenia na zjednodušenie prístupu ku konštantám, tak, aby sme ich nemuseli špecifikovať aj menom triedy. Napríklad tak, že ich umiestnime do interfejsu, ktorý implementujú jednotlivé triedy, ktoré ich používajú. To však nerobí kód prehľadnejším ani čitateľnejším a nie je to dobrá programátorská technika.

J3: Konštanty vs. Vymenovaný typ

Od verzie 5 obsahuje Java aj vymenované typy. Tie dokonca môžu obsahovať nielen hodnoty ale aj motódy. To z nich robí pomerne silný nástroj. Príklad:

```
public enum HourlyPayGrade {
  APPRENTICE {
    public double rate() {
      return 1.0;
    }
  },
  MASTER {
    public double rate() {
      return 2.0;
    }
  };
  public abstract double rate();
}
```

N1: Vyberajte popisujúce názvy

Významy názvov sa počas vývoja softvéru menia, preto by sme sa nemali báť ich vždy, keď je to vhodné, premenovať. Sú to predovšetkým názvy, ktoré robia kód čitateľným alebo nečitateľným, preto im treba venovať dostatok pozornosti. Názvy sú priveľmi dôležité na to, aby sme s nimi mohli zaobchádzať neopatrne.

N2: Vyberajte názvy na adekvátnej úrovni abstrakcie

Voľte také názvy, ktoré odrážajú úroveň abstrakcie triedy alebo funkcie, s ktorou pracujete. Nepoužívajte názvy, ktoré informujú o implementácii alebo technickom detaile. Príklad:

```
public interface Modem {
  boolean dial(String phoneNumber);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedPhoneNumber();
}
```

Rozhranie vyzerá dobre, ale v prípade, že existujú aj modemy, ktoré sa spájajú iným spôsobom, ako vytočením nejakého čísla, je vhodnejšie použiť iný názov, napr:

```
boolean connect(String connectionLocator);
String getConnectedLocator();
```

N3: Používajte štandardné názvoslovie všade, kde je to možné

Názvy sú zrozumiteľnejšie vtedy, keď sú vytvorené na základe existujúcich pravidiel alebo predchádzajúcej praxe. Napr. pri použití návrhového vzoru Decorator na vytvorenie triedy, ktorá dekoruje triedu `Modem` s možnosťou automatického prerušenia spojenia na konci komunikácie môžeme triedu nazvať `AutoHangupModemDecorator`.

Iný príklad: funkcie, ktoré prevádzajú objekty na reťazec znakov sa v Jave vždy volajú `toString`().

Jednotlivé projekty vytvárajú svoje štandardy pre tvorbu názvov a takto vzniká špecifický jazyk daného projektu. Kód by mal tento jazyk používať v maximálnej možnej miere.

N4: Jednoznačné názvy

Voľte také názvy, ktoré jednoznačne popisujú činnosť funkcie alebo premennej. Príklad:

```
private String doRename() throws Exception
{
  if (refactorReferences)
    renameReferences();
  renamePage();
  pathToRename.removeNameFromEnd();
  pathToRename.addNameToEnd(newName);
  return PathParser.render(pathToRename);
}
```

Pomenovanie `doRename()` nie je dostatočne jednoznačné, lepšie meno by bolo `renamePageAndOptionallyAllReferences()`. Je to síce dlhý názov, ale aj tak sa používa iba na jednom mieste v module, takže dĺžka názvu je oprávnená.

N5: Pre velké rozsahy používajte dlhé názvy

Dĺžka názvu by mala nejak závisieť od veľkosti rozsahu premennej alebo metódy. Premenné s krátkym rozsahom môžu mať veľmi krátke názvy, premenné s dlhými rozsahmi by mali mať aj dlhšie mená. Napr. premenné s názvami i, j sú vhodné, ak ich rozsah má dĺžku najviac asi piatich riadkov:

```
private void rollMany(int n, int pins)
{
  for (int i = 0; i < n; i++)
  g.roll(pins);
}
```

N6: Vyhýbajte sa kódovaniu názvov

Názvy by nemali obsahovať kryptické kódy s informáciami o ich type alebo rozsahu. Predpony ako `m_` alebo `f` (na vyjadrenie, že ide o „členskú“ premennú objektu (object member field), alebo typ premennej je `float`) sú v dnešných vývojových nástrojoch zbytočné. Podobne kódy obsahujúce informácie o projekte alebo subsystéme, ako napr. `vis_` (označuje vizuálny podsystém) sú nadbytočné a rozptylujú pozornosť. Dnešné vývojové nástroje tieto informácie spravidla poskytujú priamo. Nepoužívajte „maďarskú notáciu“.

N7: Názvy by mali popisovať vedľajšie efekty

Názov by mal popisovať všetko, čím funkcia, premenná, alebo trieda je alebo čo robí. Neskrývajte za názov vedľajšie efekty. Nepoužívajte jednoduché slovesá na popísanie funkcie, ktorá robí viac ako jednoduchú činnosť. Príklad:

```
public ObjectOutputStream getOos() throws IOException {
  if (m_oos == null) {
    m_oos = new ObjectOutputStream(m_socket.getOutputStream());
  }
  return m_oos;
}
```

Funkcia robí o čosi viac, ako len získanie oos. Funkcia oos vytvorí, ak ešte nie je vytvorený. Preto by vhodnejším názvom bol napr. `createOrReturnOos()`.

T1: Nedostatočné testy

Koľko testov by malo byť v testovacej sade? Bohužiaľ, miera, ktorú používa väčšina programátorov je: „Toto by mohlo stačiť“. Testovacia sada by mala otestovať všetko, čo by prípadne nemuselo fungovať. Testy nie sú úplné, ak niekde existujú podmienky, ktoré doteraz týmito testami neboli otestované, alebo neprešli ich kontrolou.

T2: Používajte nástroje pre pokrytie kódu

Nástroje pre pokrytie kódu testami sú integrované do pokročilejších vývojových prostredí. Fungujú napr. tak, že zelenou farbou označia riadky, ktoré sú pokryté testami a červenou tie, ktoré pokryté nie sú. Je to prvý rýchly spôsob ako nájsť príkazy `if` alebo `catch`, ktorých telá ešte neprešli kontrolou.

T3: Nepreskakujte triviálne testy

Sú veľmi jednoduché a ich **dokumentačná hodnota** je vyššia ako ich zriaďovacie náklady.

T4: Vynechaný test je otázka nejednoznačnosti

Niekedy si nie sme istí ako má nejaká čiastková funkčnosť vyzerať, pretože zadanie nie je celkom jasné. Pochybnosti o zadaní môžeme vyjadriť pomocou zakomentovaných testov, alebo testov označených anotáciou `@ignore`. Voľbu musíme urobiť tak, aby tieto nejasnosti nespôsobili problémy s prekladom, ale v každom prípade ich treba čo najskôr vyjasniť.

T5: Testujte ohraničujúce podmienky

Venujte zvláštnu pozornosť testovaniu ohraničujúcich podmienok. Jadro algoritmu je často správne, ale ohraničujúce podmienky sú zle zhodnotené.

T6: Dôkladne testujte okolie chýb programu

Programové chyby majú tendenciu k tomu, aby sa zhlukovali. Ak vo funkcii nájdete chybu, je rozumné túto funkciu podrobiť dôkladnému testovaniu. Je pravdepodobné, že zístíte, že táto chyba nebola jediná.

T7: Zákonitosti v zlyhaniach testov odhalujú chyby

Niekedy sa dá problém nájsť tak, že si všimneme aké druhy testov zlyhávajú a nájdeme v nich zákonitosť (je to zároveň o dôvod viac, prečo robiť čo najúplnejšie testy). Napríklad, si niekto všimne, že všetky testy zlyhali, ak je vstupný reťazec dlhší ako päť riadkov, alebo že zlyhal každý test, ktorý odovzdal v druhom argumente nejakej funkcie záporné číslo. Niekedy stačí pohľad na „červené“ riadky kódu a riešenie problému je priamo viditeľné.

T8: Pokrytie kódu testami môže odhaliť chyby

Niekedy možno nájsť stopy vedúce k príčinám, prečo niektoré testy zhlyhávajú v pokrytí kódu testami. Ak sa pozrieme na časť kódu, ktorá nie je pokrytá, tieto stopy často odhalíme.

T9: Testy by mali byť rýchle

Pomalé testy v praktickej situácii v reálnom živote budú prvé, ktoré budú vynechané z testovacej sady. Urobte všetko potrebné pre to, aby boli testy čo najrýchlejšie.

JUnit

Štandardizovaný mechanizmus jazyka Java na písanie testov. Každý javový programátor by mal vedieť o čo ide a mal by byť schopný napísať jednoduché testy pomocou JUnit. Materiál je mimo rozsah nášho predmetu, avšak odporúčame preštudovať si napr. nasledujúci tutoriál:

http://www.vogella.de/articles/JUnit/article.html