# Softvér ako skladačka a kvalita komponentov

BC. MICHAL DOBIŠ

*Slovenská technická univerzita*
*Fakulta informatiky a informačných technológií*
*Ilkovičova 3, 842 16 Bratislava*
dobis03@student.fiit.stuba.sk

**Abstrakt.** Vývoj v oblasti Softvérového inžinierstva bezpochyby smeruje k tvorbe systémov skladaním z relatívne samostatných komponentov. V dnešnej dobe už nie je problém na internete nájsť voľne dostupné triedy a knižnice tried, ktorých použitie môže značne urýchliť a tým zlacniť vývoj softvéru. Práca poukazuje, že tento zdanlivý liek na mnoho každodenných problémov môže v konečnom dôsledku vyústiť do nepredvídateľných problémov. Definuje vhodnosť komponentu z hľadiska jeho kvality, pričom do popredia kladie testovanie ako základný a najpoužívanejší spôsob jej overovania. Dokument upozorňuje na potrebu zaisťovania kvality každej súčiastky systému už v procese jej návrhu a implementácie. Pre znovupoužiteľný softvér je typickou objektovo orientovaná paradigma. Preto esej upriamuje pozornosť najmä na testovanie tried a na možnosti zlepšovania pokrytia kódu testami s využitím automatizovanej softvérovej podpory.

## Úvod

Prirodzenou snahou každého človeka je maximálne si zjednodušiť prácu. Je samozrejmé, že mnohé činnosti sa často opakujú. Vo vedeckých a technických disciplínach je tento jav označovaný aj ako „znovuobjavovanie kolesa“. Softvérové inžinierstvo nie je výnimkou a neraz sa stane, že sa plytvá drahocenným časom a prostriedkami na tvorbu už existujúcich riešení.

Prax už dávno ukázala, že sa skutočne oplatí využívať vysokú abstrakciu poskytovanú modernými programovacími jazykmi a ich štandardnými knižnicami. Firmy zistili, že mnohé časti nimi vyvíjaných systémov sa navzájom podobajú a dajú sa realizovať rovnako. Logickým vyústením takéhoto zistenia je vytváranie vlastných, viac krát použiteľných komponentov. Konečným výsledkom uvedomenia si opakovania činností je, že veľa ľudí sa rozhoduje zdieľať svoje diela so všetkými záujemcami na verejných portáloch v internete.

*Manažment v softvérovom inžinierstve, október 2006, s. 1-9.*

Dobrý manažér si musí uvedomiť možnosti poskytované rozsiahlymi zoznamami voľne dostupných funkcií, tried a knižníc. Veď prečo by mal prideľovať ľudí na vývoj niečoho, čo už vymyslel niekto iný? Odpoveď je však nečakane jednoduchá. Stačí sa zamyslieť nad kvalitou takýchto komponentov. Nebude nasadenie cudzieho kódu spôsobovať práve naopak zníženie kvality produkovaného systému? Nespôsobí v praxi jeho zdraženie z dôvodu slabej predajnosti?

Ak chcem riadiť úspešný projekt, musím byť schopný umožniť urýchlenie vývoja aj použitím cudzích komponentov. Musím však dbať na kvalitu každého z nich, je nevyhnutné každý riadne otestovať. Situácia sa nemení ani keď plánujem sám vytvárať nové súčiastky, kedy je zaistenie kvality prvoradou úlohou. Snažím sa preto v tejto eseji definovať čo je to kvalita softvéru pomocou základných metrík. Uvádzam možnosť jej overovania testovaním komponentov a popisujem možnosti automatizácie tejto činnosti. Pritom vychádzam z objektovo orientovanej paradigmy a za základný prvok softvéru považujem triedu.

## Kvalita softvérových komponentov

Pred zavedením súčiastky do vytváraného systému, si musí byť manažér istý jej schopnosťou plniť funkcie, na ktoré má byť v konkrétnom prostredí použitá. Aby rozhodovanie o použití komponentu bolo relevantné, je nevyhnutné byť schopný navzájom porovnať rôzne komponenty. Treba zistiť, či je vhodné použiť vybranú súčiastku v závislosti od jej kvality.

Kvalita sa nám na prvý pohľad javí ako samozrejmý pojem. Každý vie povedať, či je daný produkt kvalitný alebo nie. Na základe čoho tak ale usudzujeme? Pre bežného používateľa počítača môže byť kvalitný systém daný ľahkou ovládateľnosťou a príjemným vzhľadom. Skúsenejší používateľ ocení nadštandardné funkcie. Nakoniec programátor bude spokojný, ak je zdrojový kód ľahko čitateľný a má dobre definovanú vnútornú štruktúru. Je zjavné, že kvalita je nejasný pojem. Ako sa ale dá hovoriť o jej overovaní, keď vlastne nevieme čo to je? Aby sa tento pojem stal v praxi použiteľným, treba ho exaktne definovať z rôznych uhlov pohľadu.

Všeobecne prijímané je definovanie kvality ako množiny atribútov softvérového produktu, pomocou ktorých je popísaná a vyhodnocovaná [3]. K jednotlivým atribútom bývajú priraďované metriky, pomocou ktorých sa určuje ich naplnenie posudzovaným produktom. Najbežnejšie metriky určujúce kvalitu komponentu sú nasledovné:

*Prispôsobivosť*

Prispôsobivosť je miera, do akej sa komponent môže použiť bez modifikácie v iných aplikáciách alebo prostrediach, než pre ktoré bol pôvodne vytvorený.

Typicky býva ohodnocovaná ako nezávislosť na veľkosti pamäti a výkone procesora. Pre jednotlivé moduly v takom prípade postačuje experimentálne zistiť percentuálne ohodnotenie (využitie výkonu)/(celková výkonnosť).

Pre prax zaujímavejším odhadom je podľa mňa schopnosť prispôsobiť sa novým podmienkam, najmä podobe zbavenia sa obmedzení na veľkosti dátových štruktúr.

Vhodnejším údajom je preto zrejme podiel (počet modulov s obmedzeniami veľkostí zapísanými priamo v kóde - napríklad veľkosť poľa)/(celkový počet modulov).

*Úplnosť*

Úplnosť je miera, do akej komponent implementuje všetky požadované schopnosti.

Pre číselné vyjadrenie sa používajú príčiny a efekty dané na základe špecifikácie (Cause and effect graphing). Podmienky zo špecifikácie so žiadnym efektom a efekty bez žiadnych podmienok sú identifikované ako nejasnosti. Výsledkom je vyjadrenie počtu zostávajúcich nejasností a neúplností podľa vzorca (1).

$$CE[\%] := 100 * (1 - \frac{A_{exist}}{A_{tot}}) \quad (1)$$

kde $A_{exist}$ predstavuje počet zostávajúcich nejasností a $A_{tot}$ celkový počet nejasností. 100% udáva, že neexistujú žiadne nejasnosti.

*Správnosť*

Správnosť sa definuje ako miera, do akej komponent:

- neobsahuje chyby v špecifikácií, návrhu a implementácií,
- spĺňa požiadavky a používateľské potreby,
- je schopný poskytovať očakávané výstupy pre dané vstupy.

Číselným vyjadrením bývajú počty správ o chybe za daný časový interval a počet otvorených problémov.

*Efektívnosť*

Efektívnosť je miera, do akej je komponent schopný realizovať navrhnuté funkcie s minimálnym využitím zdrojov.

Časová a pamäťová zložitosť modulu sa dá počítať priamo z návrhu použitých algoritmov. Medzi experimentálne získavané údaje patrí percentuálne využitie CPU alebo maximálne využitie operačnej pamäte.

*Všeobecnosť*

Všeobecnosť komponentu býva určená ako škála úloh, pre ktoré je komponent použiteľný. Jej číselným vyjadrením môže byť asi dosť ťažko určiteľný počet aplikačných domén, pre ktoré je vhodná.

*Udržovateľnosť*

Úsilie, ktoré treba vynaložiť na zmenu komponentu s cieľom odstrániť chyby, zvýšiť výkonnosť alebo iné parametre, prípadne prispôsobiť komponent novým podmienkam je označované ako udržovateľnosť.

Jej odhadom býva počet riadkov kódu, alebo počet rozhodovaní v module. Alternatívnym vyhodnotením býva počet vstupno-výstupných premenných.

*Modularita*

Modularita je spôsob, akým je komponent rozdelený na sub-komponenty.

Definovaná býva súdržnosťou a zviazanosťou komponentov. Súdržnosť je potom vzťah medzi časťami komponentu a počíta sa pomocou vstupov a výstupov do jednotlivých funkcií komponentu (2). Zviazanosť je daná počtom prepojení medzi rôznymi komponentmi.

$$Zviazanost := \sqrt{X^2 + Y^2} \tag{2}$$

kde X udáva obrátený počet priradení v module a Y je podiel výstupov a odlišných vstupov z a do funkcií.

*Prenosnosť*

Prenosnosť je úsilie, ktoré treba na prenos výrobku z jednej platformy na inú.

Býva určovaná ako počet použitých vlastností špecifických pre implementačný jazyk a operačný systém.

*Spoľahlivosť*

Spoľahlivosť predstavuje schopnosť komponentu vykonávať svoje funkcie v časovom intervale pri daných podmienkach.

Ako jej odhad sa používa časový interval medzi výskytmi chýb počas vykonávania.

*Zrozumiteľnosť*

Zrozumiteľnosť sa definuje ako miera, do akej je zmysel komponentu jasný jeho používateľovi. Zaisťuje sa nízkou zložitosťou a dokumentáciou.

Jasným vyjadrením zrozumiteľnosti býva počet strán a gramatických chýb v dokumentácií, počet funkcií a modulov v programe alebo počet nejasných referencií v zdrojovom kóde.

## Testovanie komponentov

Overovanie kvality softvéru je zaiste ťažká úloha, ak nie dokonca nemožná. Je totiž takmer nemožné byť dokonalý v každom smere, keďže niektoré metriky sa v praxi v podstate vylučujú. Zlepšovanie modularity niekedy znižuje zrozumiteľnosť, prenosnosť často znižuje efektivitu, a znižovanie časovej zložitosti takmer vždy vedie k zvyšovaniu pamäťovej zložitosti. Vzhľadom na to je samozrejmé, že kvalitu softvérového komponentu musí zabezpečovať v prvom rade samotný vývojár. Už v procese návrhu predurčuje mnohé základné vlastnosti vrátane modularity, všeobecnosti a prenosnosti. Implementácia, testovanie a dokumentácia potom dotvára mozaiku vlastností produktu. Ako tvorca mnohých programov som presvedčený, že

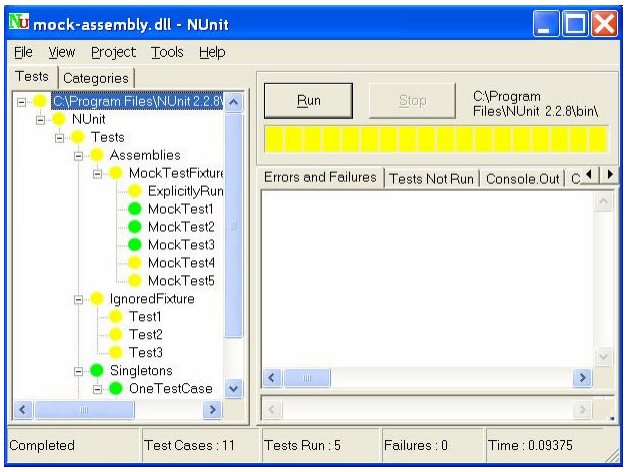
dobrý softvérový inžinier by mal paralelne s písaním samotného programu vytvárať aj množinu testov na overenie správnosti vytváraných komponentov.

Testovanie je v súčasnosti často zanedbávanou, medzi softvérovými inžiniermi neveľmi obľúbenou oblasťou. Veľa programátorov k nemu pristupuje ako k nutnému zlu a neplánuje testy dopredu. Často len skúsia zaradiť komponent do existujúcej časti systému a vykonajú niekoľko skúšok s bežnými, alebo s krajnými hodnotami parametrov. Pre aktuálne nasadenie komponentu môže takýto test postačovať, no pri zmene prostredia sa môžu vyskytnúť neočakávané chyby. Takéto „okresávanie testov na minimum“ potom často vedie k neočakávaným problémom pri integrácií systému alebo v horšom prípade k výpadkom u zákazníka.

Domnievam sa, že príčinou je absencia budovania testovacích návykov už počas prvých kontaktov nádejného budúceho informatika s modernými programovacími jazykmi. Tieto sa neskôr prenášajú ďalej do praxe aj preto, že väčšina z nás má nedostatok znalostí ako si testovanie uľahčiť.

Z osobných skúseností viem, že dôkladné testovanie komponentu vyžaduje postupnosti volaní metód objektu, ktoré v aktuálnom systéme nie sú typické. Vytvárať kvôli tomu ďalší program so samostatným spustiteľným súborom je neefektívne a preto sa takmer vôbec nerealizuje. Cieľom však je, aby sa otestovala čo najväčšia časť komponentu, pričom navrhnuté testy je dobré uschovať pre ich opätovné použitie pri zmenách na zdrojovom kóde.

Elegantným riešením problému je zviazať vždy testovanú triedu s triedou na jej testovanie. Programátor môže súbežne s tvorbou komponentu písať aj statické testovacie metódy umožňujúce opakované spúšťanie identických akceptačných testov ako sekvencií volaní metód testovanej triedy. Chýbajúcim krokom je možnosť spúšťania testov nezávisle na pôvodne vytváranom systéme. V tomto smere sú veľmi užitočné prostriedky, medzi ktoré patrí JUnit pre programovací jazyk Java a NUnit pre jazyky .NET [1], ktoré umožňujú z grafického prostredia spúšťať vytvorené testy na triedach a vyhodnocujú úspešnosť testu (pozri Obr. 1).




**Obr. 1.** Grafické používateľské rozhranie pre spúšťanie a vyhodnocovanie testov s NUnit

Základnou myšlienkou softvérových prostriedkov pre podporu testovania je zaiste zvýšiť úroveň pokrytia kódu testami. Zrejme jediný efektívny spôsob ako to dosiahnuť je maximálne zjednodušiť celú činnosť programátora plánujúceho vykonať testy. Jednotlivé prostriedky bývajú optimalizované pre ich nenáročné, používateľsky orientované využívanie. V praxi to znamená, že typicky dovoľujú písať zdrojový kód testov priamo do súboru, kde sa nachádza testovaná trieda. Skrátenie písania testovacích tried býva realizované buď ich odvodením od základných tried poskytovaných príslušným testovacím rámcom, alebo bývajú odlíšené pomocou atribútov. Druhý spomenutý spôsob využíva napríklad vyššie uvedený NUnit. Obrázok číslo 2 zobrazuje triedu, ktorá je atribútmi jazyka C# predurčená na testovanie. Príklad slúži na jednoduché overenie komponentu účet (Account).

```
namespace bank
{
  using NUnit.Framework;

  [TestFixture]
  public class AccountTest
  {
    [Test]
    public void TransferFunds()
    {
      Account source = new Account();
      source.Deposit(200.00F);
      Account destination = new Account();
      destination.Deposit(150.00F);

      source.TransferFunds(destination, 100.00F);
      Assert.AreEqual(250.00F, destination.Balance);
      Assert.AreEqual(100.00F, source.Balance);

    }
  }
}
```

**Obr. 2.** Príklad jedného testu pomocou NUnit

Je zrejmé, že cestou ako dosiahnuť lepšie testovanie je minimalizovať nadbytočné úsilie, ktoré so sebou táto úloha prináša. Myslím si, že lenivosť je tou najprirodzenejšou ľudskou vlastnosťou hneď po schopnostiach zdedených po zvieracích predkoch. Preto sa okamžite ponúka aj ďalšia otázka: dá sa testovanie automatizovať úplne? Podľa mojich znalostí sa v praxi zatiaľ plne automatizované testovanie v oblasti objektovo orientovaného softvéru neujalo, no výskum poukazuje na dva základné smery vývoja.

### Automatické generovanie testov na základe presnej špecifikácie

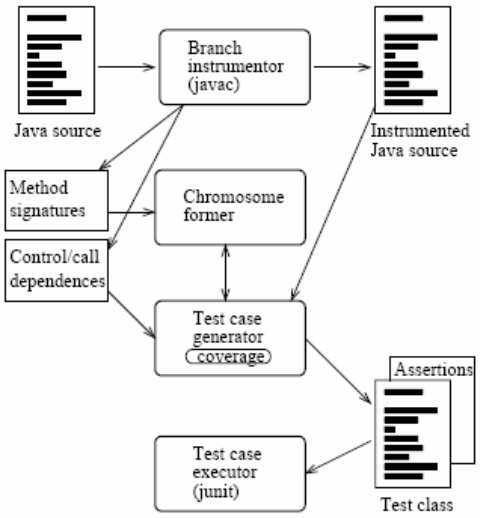
Slovo „automatické“ predznamenáva, že niečo bude generované počítačom. Oživenie každej nám tak známej kopy drôtov a plošných spojov je realizované implementovaním algoritmov, ktoré majú vždy nejaký vstup a výstup. Výstupom je v našom prípade celkom prirodzene množina testov pozostávajúcich zo sekvencií volaní metód testovanej triedy. Jednotlivé metódy sa navzájom líšia najmä tým, čo považujú za vstup.

Prvá metóda, ktorá ma zaujala berie za vstup uzavretú špecifikáciu [2] testovaného komponentu. Vývojár, ktorý chce využiť služieb takéhoto generátora testov musí byť schopný zapísať triedu a jej metódy v jazyku udávajúcom vstupné a výstupné podmienky metód, spolu s očakávanými zmenami vnútorného stavu objektu.

Testovací systém vďaka známej špecifikácií presne vie, čo je úlohou komponentu a jeho jednotlivých metód. Na základe toho je schopný generovať testy, spúšťať ich a vyhodnotiť ich úspešnosť. Získame tak veľa testov, ktorých výsledky môžu byť plne automaticky vyhodnotené podľa výstupných podmienok. Za ponúkaný luxus však zaplatíme nutnosťou detailnej špecifikácie každej triedy v niektorom nie príliš rozšírenom špecifikačnom jazyku. Preto je na mieste otázka, kde sa pomýlim s väčšou pravdepodobnosťou – pri písaní uzavretej špecifikácie alebo pri písaní samotného programu?

### Automatické generovanie testov priamo zo zdrojového textu

Zbaviť sa písania ďalších riadkov udávajúcich špecifikáciu ponúka vízia generovania testov priamo zo zdrojového textu programu [4]. Elegantné riešenie problému ponúkajú objektovo orientované jazyky s reflexiou, medzi ktoré patrí napríklad Java alebo C#. Plne postačujúci algoritmus (pozri Obr. 3) pozostáva zo skompilovania zdrojového textu triedy, z ktorého sa vyžiada typ objektu obsahujúci zoznam verejných konštruktorov a metód. Následne sa evolučným algoritmom vytvárajú chromozómy definujúce jednotlivé testy ako postupnosti volaní metód s rôznymi parametrami. Evolučný algoritmus vyberie ako najlepší ten test, ktorý pokryje čo najviac vetiev programu.




**Obr. 3.** Spôsob práce generátora testov zo zdrojového kódu pre jazyk Java. [4]

Popísaný algoritmus má za vstup len zdrojový kód, ktorý môže byť chybný. Preto je zrejmé, že kontroly na očakávané hodnoty a výsledný stav objektu musí na koniec touto metódou vybraných testov doplniť vývojár. Má však dobrý predpoklad, že vygenerované testy skutočne dobre overia správnosť komponentu.

Výstup programu realizujúceho popísané generovanie testov môže byť integrovaný s existujúcimi softvérovými prostriedkami pre podporu testovania. Softvérový inžinier tak môže pohodlne spúšťať získané testy v ním zvolenom čase rovnako dobre, ako ich smie upraviť podľa vlastného uváženia. Som presvedčený, že práve týmto smerom by sa mali uberať tvorcovia systémov pre automatizované testovanie. Aktuálne je však takýchto systémov podľa mojich vedomostí nedostatok, respektíve ich výpočtová zložitosť spôsobuje dlhé čakanie, kým na súčasných výpočtových prostriedkoch získame výsledky.

## Záver

Pri práci na softvérových projektoch sa často využívajú knižnice tried z viac, alebo menej dôveryhodných zdrojov. Komponenty od známych firiem z oblasti informačných technológií sa istotne netreba obávať zaradiť do vyvíjaného systému. Ich nevýhodou oproti ľahšie dostupným je v prvom rade cena. Preto som presvedčený, že v nekritických častiach systémov sa oplatí využívať aj zdroje súčiastok ako sú nespočetné verejné portály programátorov alebo open source projekty od najrôznejších autorov. Pri ich výbere však treba dbať na overenie ich kvality, ktoré musí prebehnúť rýchlo a efektívne, aby sa nestratila výhoda ušetreného času.

Najefektívnejším spôsobom kontroly vlastností komponentu je zaiste jeho otestovanie. Viem, že nie všetky atribúty softvéru sa dajú overovať testovaním a takisto si uvedomujem, že ním nemožno preukázať neprítomnosť chýb. Určite je však dôveryhodnejší komponent, ktorého funkcionalitu možno overiť jednoduchými testami. Ešte lepšia situácia sa naskytne, keď k posudzovanej súčiastke pripravil aspoň základné sekvencie volaní metód slúžiace na overovanie kvality už samotný autor komponentu. Nápomocné mu pritom môžu byť aj rôzne podporné softvérové prostriedky.

Neviem aká je budúcnosť automatického generovania testov. Som ale presvedčený, že minimálne automatizovanú podporu testovania umožňujúcu pohodlné písanie, spúšťanie a vyhodnocovanie jednotlivých testov by mal používať každý. Uľahčí tak život nie len sebe, ale aj každému, kto bude neskôr používať jeho dielo.

## Použitá literatúra

1. Beck, K. et al: *NUnit*, 2005 [cit. 2006-10-25].
   http://www.nunit.org/

2. Leow W.K., Khoo S.C., Sun Y.: Automated Generation of Test Programs from Closed Specifications of Classes and Test Cases. In: *26th International Conference on Software Engineering (ICSE'04)*, IEEE (2004), 96-105.
3. Salamon, W.J., Wallace, D.R.: *Quality characteristics and metrics for reusable sofware*, 1994 [cit. 2006-10-25].
   http://hissa.ncsl.nist.gov/HHRFdata/Artifacts/ITLdoc/5459/metrics.html
4. Tonella, P.: Evolutionary testing of classes. In: *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ACM Press, New York (2004), 119-128.

## Annotation

*Software composition and quality of components*

No doubt that evolution in software engineering tends to produce new systems by composing separate components. Nowadays, it is not a problem to find classes and class libraries in the Internet, freely available to everyone who wants to enhance the speed and decrease the costs of his project development. Author realizes that this medicine for many everyday problems might be illusionary and can result in unpredictable problems. The component quality as the measurement of its feasibility is described briefly. Document draws attention to need of the quality ensuring during whole design and implementation phase, while the focus is placed on unit testing as the most common and the most efficient way to do it. As the object oriented paradigm is considered typical for reusable software development, the automation possibilities of class testing and the code coverage is discussed in this paper.