Міністерство освіти і науки України
Тернопільський національний економічний університет
Факультет комп'ютерних інформаційних технологій

Піговський Юрій Романович

Інженерія кросплатформного програмного забезпечення

Навчальний посібник

Напрям підготовки — 6.050103 "Програмна інженерія"

Тернопіль — 2014

Посібник підготовано в Тернопільському національному економічному університеті

Автор:	кандидат технічних наук, доцент Піговський Юрій Романович
Рецензенти:	доктор технічних наук, професор Саченко Анатолій Олексійович
	кандидат технічних наук, доцент Васильків Василь Васильович

Затверджено на засіданні науково-методичної комісії за напрямом підготовки 6.050103 “Програмна інженерія” при Факультеті комп’ютерних інформаційних технологій Тернопільського національного економічного університету, протокол №x від DD.MM.2014 р.

Вступ

Комп'ютерні технології застосовуються в численних предметних областях. Зокрема в засобах зв'язку, навігації, вбудованих в транспортні засоби системах управління, побутовій техніці, біомедичних приладах і т.д. Усі ці комп'ютерні системи є гетерогенними, тобто мають різну конфігурацію апаратного забезпечення та операційних систем.

Наприклад, більшість мобільних пристроїв (телефони, планшети, букрідери, засоби GPS навігації) використовують доволі потужні центральні процесори, достатньо великі об'єми оперативної пам'яті, дисплеї великої роздільної здатності та ОС Android, iOS чи Windows (подано в алфавітному порядку). Тоді як маршрутизатори фірми Cisco використовують енергоекономічні, низькочастотні центральні процесори, малий об'єм оперативної пам'яті, не обладнані дисплеєм та працюють під керуванням Internet Operating System.

Комбінацію апаратного забезпечення та операційної системи надалі будемо називати *платформою*.

Використання апаратних ресурсів платформи вимагає розробки програмного забезпечення (ПЗ). Розробку нового ПЗ доречно виконувати на основі модулів повторного використання (reusable units): фреймворків, бібліотек класів, засобів розробки (SDK) та інтегрованих середовищ розробки (IDE).

Досвід, отриманий на одній платформі чи предметній області може бути використаний при розробці ПЗ для іншої платформи чи предметної області. Тому ПЗ та модулі повторного використання проектують в такий спосіб, щоб їх можна було без модифікацій зібрати (build) і використовувати на кількох платформах одночасно, що й називається *кросплатформністю*.

На думку автора, комплексна дисципліна "Інженерія кросплатформного програмного забезпечення" складається з трьох складових дисциплін: "Конструювання ПЗ", "Технологія Java" та "Технологія .NET".

Дисципліна "Конструювання ПЗ" стосується таких питань як

- конвенції щодо оформлення коду на різних мовах програмування (відступи, дужки, іменування): для Java — [`http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`], для C#.NET — [`http://msdn.microsoft.com/en-us/library/ff926074.aspx`]
- користування засобами автоматизованого документування коду на основі анотацій: javadoc [`http://www.oracle.com/technetwork/java/javase/documentation/index-137868.`

html] для Java; XML Documentation [https://www.simple-talk.com/dotnet/.net-tools/taming-sandcastle-a-.net-programmers-guide-to-documenting-your-code/] та NDoc для .NET,

- теоретичні засоби мінімізації складності за допомогою грамотної інкапсуляції та дизайн паттернів (на основі фундаментальної книги Стівена Макконнелла про конструювання ПЗ),
- засоби модульного (unit) тестування: JUnit для Java; NUnit чи MSUnit для .NET.
- засоби автоматизації функціонального (інтеграційного) тестування, або, іншими словами, UI тестування, наприклад, сервіс http://testfairy.com та SDK засіб uiautomator для Java-Android [http://developer.android.com/tools/help/uiautomator/index.html].

Дисципліна "Технологія Java" описує основні питання платформи Java [http://www.oracle.com/technetwork/topics/newtojava/java-technology-concept-map-150250.pdf], а саме:

- особливості Java як мови програмування: структура проекту, пакети як аналог простору імен в C++ та C#, інтерфейс Enumerable, узагальнені класи, успадкування, область видимості, анонімна реалізація інтерфейсів як аналог lambda операторів в C#, порівняння об'єктів та значень, переозначення методів equals та clone, обчислювальні потоки чи нитки, засоби синхронізації на основі моніторів, ввід-вивід, користування командним рядком java, javac та змінними середовища CLASSPATH, JAVA_HOME, ANDROID_HOME,
- засоби автоматизації збирання, тестування і публікування Ant та gradle, репозиторії бібліотек maven,
- інтегроване середовище розробки InelliJ Idea чи Android Studio (засоби інспекції коду та рефакторингу),
- основи розробки для J2SE (файловий ввід-вивід, серіалізація),
- основи розробки для ОС Android: життєвий цикл Activity, робота з ресурсами (стрічки, зображення, звук), форматування layout, обробка подій, робота зі списками ListView, перенесення даних з однієї активності в іншу, особливості проектування застосунків для Андроід (синглтон застосунку).

В рамках дисципліни “Технологія .NET” будуть вивчатися:

- особливості мови C#: структура solution, простори імен, lambda вирази, інтерфейс IEnumerable, події, делегати, робота з обчислювальними потоками (нитками), засоби синхронізації, робота з командним рядком і файловою системою
- робота з Windows Forms: форматування UI, обробка подій, робота з графікою,
- факультативно — розробка кросплатформних (для ОС Android, iOS, Windows) мобільних додатків за допомогою Xamarin SDK і фреймворка MvvmCross на мові програмування C# та бібліотек класів .NET.

1 Конструювання ПЗ

Дисципліна "Конструювання ПЗ" описує аспекти створення і юніт-тестування діючого і корисного ПЗ, а тому є невідємним етапом в життєвому циклі розробки ПЗ за будь-якою моделлю.

Проектування ПЗ можна в той чи в інший спосіб обійти, провівши його подумки, однак обійти процес написання коду і отримати при цьому діюче ПЗ — неможливо. Саме процесом підготовки, удосконалення і тестування коду і займається дисципліна "Конструювання ПЗ" [Стівен Макконнелл. Конструювання ПЗ].

1.1 Конвенції кодування на мові Java

public class має бути першим у сирцевому файлі.

Сирцевий файл складається в такому порядку: Початкові коментарі, пакетний та імпортуючі оператори, оголошення класів та інтерфейсів.

Початковий коментар:

```
/*
 * Classname
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

На початку класу йдуть статичні, потім звичайні поля у порядку спадання видимості: First the public variables, then the protected, then package level (no access modifier), and then the private.

Потім йдуть конструктори, а тоді методи.These methods should be grouped by functionality rather than by scope or accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

Four spaces should be used as the unit of indentation.

Avoid lines longer than 80 characters.

Here are some examples of breaking method calls (indent 8 spaces):

someMethod(longExpression1, longExpression2, longExpression3, longExpression4, longExpression5);

var = someMethod1(longExpression1, someMethod2(longExpression2, longExpression3));

Дужки краще тримати в тій самій лінії:

```
longName1 = longName2 * (longName3 + longName4 - longName5)
            + 4 * longname6; // PREFER

longName1 = longName2 * (longName3 + longName4
                         - longName5) + 4 * longname6; // AVOID

//CONVENTIONAL INDENTATION
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
    ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
    ...
}

//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();               //MAKE THIS LINE EASY TO MIS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        ||!(condition5 && condition6)) {
```

```
    doSomethingAboutIt();
}
```

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

Comments should not be enclosed in large boxes drawn with asterisk
Comments should never include special characters such as form-feed

A single-line comment should be preceded by a blank line. Here's

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

Very short comments can appear on the same line as the code they

Here's an example of a trailing comment in Java code:

```
if (a == 2) {
    return TRUE;            /* special case */
} else {
    return isPrime(a);      /* works only for odd a */
}
```

Each doc comment is set inside the comment delimiters /**...*/, w

```
/**
 * The Example class provides ...
```

```
 */
public class Example { ...
```

Doc comments should not be positioned inside a method or construct

One declaration per line is recommended since it encourages comme

```
int level; // indentation level
int size;  // size of table
```

is preferred over

```
int level, size;
```

Do not put different types on the same line. Example:

```
    int foo,  fooarray[]; //WRONG!
```

Note: The examples above use one space between the type and the id

```
int      level;           // indentation level
int      size;             // size of table
Object   currentEntry;     // currently selected table entry
```

initialize local variables where they're declared.

Put declarations only at the beginning of blocks. (A block is any

```
void myMethod() {
    int int1 = 0;          // beginning of method block

    if (condition) {
        int int2 = 0;      // beginning of "if" block
        ...
    }
}
```

The one exception to the rule is indexes of **for** loops, which in Ja

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels.

```
int count;
...
myMethod() {
    if (condition) {
        int count = 0;      // AVOID!
        ...
    }
    ...
}
```

No space between a method name and the parenthesis "(" starting it
Open brace "{" appears at the end of the same line as the declarat
Closing brace "}" starts a line by itself indented to match its co
Methods are separated by a blank line

Braces are used around all statements, even single statements, whe

```
if (condition) {
    statements;
}
```

```
if (condition) {
    statements;
} else {
    statements;
}
```

```
if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

When using the comma operator in the initialization or update clau

A **do–while** statement should have the following form:

```
do {
```

```
    statements;
} while (condition);
```

A **switch** statement should have the following form:

```
switch (condition) {
case ABC:
    statements;
    /* falls through */
case DEF:
    statements;
    break;
case XYZ:
    statements;
    break;
default:
    statements;
    break;
}
```

Every time a **case** falls through (doesn't␣include␣a␣break␣statement

Every␣switch␣statement␣should␣include␣a␣default␣case.␣The␣break␣in

A␣try-catch␣statement␣should␣have␣the␣following␣format:

```
try␣{
␣␣␣␣statements;
}␣catch␣(ExceptionClass␣e)␣{
␣␣␣␣␣statements;
}
```

A␣try-catch␣statement␣may␣also␣be␣followed␣by␣finally,␣which␣execu

```
try␣{
␣␣␣␣statements;
}␣catch␣(ExceptionClass␣e)␣{
␣␣␣␣statements;
}␣finally␣{
```

```
    statements;
}

Two blank lines should always be used in the following circumstanc
Between sections of a source file
Between class and interface definitions

One blank line should always be used in the following circumstance
Between methods
Between the local variables in a method and its first statement
Before a block (see section 5.1.1) or single-line (see section 5.1
Between logical sections inside a method to improve readability

A keyword followed by a parenthesis should be separated by a space
        while (true) {
            ...
        }

Note that a blank space should not be used between a method name a

A blank space should appear after commas in argument lists.
All binary operators except . should be separated from their opera
    a += c + d;
    a = (a + b) / (c * d);

    while (d++ = s++) {
        n++;
    }
    printSize("size is " + foo + "\n");



The expressions in a for statement should be separated by blank sp
    for (expr1; expr2; expr3)

Casts should be followed by a blank space. Examples:
    myMethod((byte) aNum, (Object) x);
    myMethod((int) (cp + 5), ((int) (i + 3))
                                + 1);
```

Variable␣names␣should␣not␣start␣with␣underscore␣_␣or␣dollar␣sign␣$

One–character␣variable␣names␣should␣be␣avoided␣except␣for␣temporar

The␣names␣of␣variables␣declared␣class␣constants␣and␣of␣ANSI␣consta

```
static␣final␣int␣MIN_WIDTH␣=␣4;
static␣final␣int␣MAX_WIDTH␣=␣999;
static␣final␣int␣GET_THE_CPU␣=␣1;
```

Don't make any instance or **class** variable **public** without good reas

One example of appropriate **public** instance variables is the **case** v

Avoid using an object to access a **class** (**static**) variable or metho

```
classMethod();              //OK
AClass.classMethod();       //OK
anObject.classMethod();     //AVOID!
```

Numerical constants (literals) should not be coded directly, excep

Avoid assigning several variables to the same value in a single st

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

It is generally a good idea to use parentheses liberally in expres

```
if␣(a␣==␣b␣&&␣c␣==␣d)␣␣␣␣␣//␣AVOID!
if␣((a␣==␣b)␣&&␣(c␣==␣d))␣//␣RIGHT
```

Similarly,

```
if␣(condition)␣{
␣␣␣␣return␣x;
}
return␣y;
should␣be␣written␣as

return␣(condition␣?␣x␣:␣y);
```

```
If␣an␣expression␣containing␣a␣binary␣operator␣appears␣before␣the␣
(x␣>=␣0)␣?␣x␣:␣-x;

Use␣XXX␣in␣a␣comment␣to␣flag␣something␣that␣is␣bogus␣but␣works.␣U

A␣doc␣comment␣is␣written␣in␣HTML␣and␣must␣precede␣a␣class,␣field,␣

/**
␣*␣Returns␣an␣Image␣object␣that␣can␣then␣be␣painted␣on␣the␣screen
␣*␣The␣url␣argument␣must␣specify␣an␣absolute␣{@link␣URL}.␣The␣name
␣*␣argument␣is␣a␣specifier␣that␣is␣relative␣to␣the␣url␣argument.
␣*␣<p>
␣*␣This␣method␣always␣returns␣immediately,␣whether␣or␣not␣the
␣*␣image␣exists.␣When␣this␣applet␣attempts␣to␣draw␣the␣image␣on
␣*␣the␣screen,␣the␣data␣will␣be␣loaded.␣The␣graphics␣primitives
␣*␣that␣draw␣the␣image␣will␣incrementally␣paint␣on␣the␣screen.
␣*
␣*␣@param␣␣url␣␣an␣absolute␣URL␣giving␣the␣base␣location␣of␣the␣im
␣*␣@param␣␣name␣the␣location␣of␣the␣image,␣relative␣to␣the␣url␣arg
␣*␣@return␣␣␣␣␣␣the␣image␣at␣the␣specified␣URL
␣*␣@see␣␣␣␣␣␣␣␣␣Image
␣*/
␣public␣Image␣getImage(URL␣url,␣String␣name)␣{
␣␣␣␣␣␣␣␣try␣{
␣␣␣␣␣␣␣␣␣␣␣␣return␣getImage(new␣URL(url,␣name));
␣␣␣␣␣␣␣␣}␣catch␣(MalformedURLException␣e)␣{
␣␣␣␣␣␣␣␣␣␣␣␣return␣null;
␣␣␣␣␣␣␣␣}
␣}

␣Insert␣a␣blank␣comment␣line␣between␣the␣description␣and␣the␣list␣
The␣first␣line␣that␣begins␣with␣an␣"@"␣character␣ends␣the␣descript

The␣first␣sentence␣of␣each␣doc␣comment␣should␣be␣a␣summary␣sentenc
This␣sentence␣ends␣at␣the␣first␣period␣that␣is␣followed␣by␣a␣blank

␣␣␣/**
␣␣␣␣*␣This␣is␣a␣simulation␣of␣Prof.␣Knuth's MIX computer.
```

```
     */
```
However, you can work around **this** by typing an HTML meta-character

```
    /**
     * This is a simulation of Prof. Knuth's MIX computer.
     */
```
Ideally, make description complete enough **for** conforming implemen

If you must document implementation-specific behavior, please docu

On Windows systems, the path search behavior of the loadLibrary m

The␣use␣of␣"On␣Windows"␣at␣the␣beginning␣of␣the␣sentence␣makes␣it␣

Use␣<code>␣style␣for␣keywords␣and␣names.
Keywords␣and␣names␣are␣offset␣by␣<code>...</code>␣when␣mentioned␣i
Java␣keywords
package␣names
class␣names
method␣names
interface␣names
field␣names
argument␣names
code␣examples

adding␣a␣link␣to␣an␣API␣name␣if:

The␣user␣might␣actually␣want␣to␣click␣on␣it␣for␣more␣information␣(
Only␣for␣the␣first␣occurrence␣of␣each␣API␣name␣in␣the␣doc␣comment␣

When referring to a method or constructor that has multiple forms,

The add(**int**, Object) method adds an item at a specified position i

However, **if** referring to both forms of the method, omit the paren
The add method enables you to insert items. (preferred)
The add() method enables you to insert items. (avoid when you mean

A method **implements** an operation, so it usually starts with a verb

Gets the **label** of **this** button. (preferred)
This method gets the **label** of **this** button. (avoid)

Class/**interface**/field descriptions can omit the subject and simply
These API often describe things rather than actions or behaviors:
A button **label**. (preferred)
This field is a button **label**. (avoid)

Use "this" instead of "the" when referring to an object created fr
For example, the description of the getToolkit method should read
Gets the toolkit **for this** component. (preferred)
Gets the toolkit **for** the component. (avoid)

Avoid – The description below says nothing beyond what you know fr

```
/**
 * Sets the tool tip text.
 *
 * @param text  the text of the tool tip
 */
public void setToolTipText(String text) {
```

Preferred – This description more completely defines what a tool

```
/**
 * Registers the text to display in a tool tip.  The text
 * displays when the cursor lingers over the component.
 *
 * @param text  the string to display.  If the text is null,
 *              the tool tip is turned off for this component.
 */
public void setToolTipText(String text) {
```

Avoid Latin
use "also␣known␣as" instead of "aka", use "that␣is" or "to␣be␣spe

Include tags in the following order:

@author (classes and interfaces only, required)

```
@version (classes and interfaces only, required. See footnote 1)
@param (methods and constructors only)
@return (methods only)
@exception (@throws is a synonym added in Javadoc 1.2)
@see
@since
@serial (or @serialField or @serialData)
@deprecated (see How and When To Deprecate APIs)

If desired, groups of tags, such as multiple @see tags, can be sep

Multiple @author tags should be listed in chronological order, wit

Multiple @param tags should be listed in argument-declaration orde

Multiple @throws tags (also known as @exception) should be listed

@see #field
@see #Constructor(Type, Type...)
@see #Constructor(Type id, Type id...)
@see #method(Type, Type,...)
@see #method(Type id, Type, id...)
@see Class
@see Class#field
@see Class#Constructor(Type, Type...)
@see Class#Constructor(Type id, Type id)
@see Class#method(Type, Type,...)
@see Class#method(Type id, Type id,...)
@see package.Class
@see package.Class#field
@see package.Class#Constructor(Type, Type...)
@see package.Class#Constructor(Type id, Type id)
@see package.Class#method(Type, Type,...)
@see package.Class#method(Type id, Type, id)
@see package

An @param tag is "required" (by convention) for every parameter, e
```

Література

1. *Фамилия И.О., ...* Название доклада. // Тезисы докладов Название конференции — Город-организатор: Институт-организатор, год конфы. — С. xx.

2. *Фамилия И.О., ...* Название статьи. // Название журнала, Год. — Том XX. — С. xxx–xxx.