第11章　软 件 实 现

11.1　程序设计语言

程序设计语言一直在不断地演化和演变，其发展经历了从机器语言到高级语言的过程。

计算机问世初期，程序设计语言是与计算机硬件紧密相关的机器语言和汇编语言，编写这种语言程序难度大、效率低，不易于理解且难以调试。

11.1.1　程序设计语言的特性

特定的程序设计语言有一些特定的限制，它们影响着程序员描述和处理问题的方式。程序设计语言应着重考虑程序员易学易用、不易出错，因此程序设计语言须考虑下列特性：

(1) 一致性(Uniforminy)。

(2) 二义性(Ambiguity)。

(3) 紧致性(Compactness)。

(4) 局部性(Locality)。

(5) 线性(Linearity)。

11.1.2　程序设计语言的选择

总的来说，程序设计语言的选择需要结合具体问题进行分析评价，下面给出一些可供参考的实用标准：

(1) 系统用户的要求。

(2) 程序员的知识。

(3) 软件可移植性要求。

(4) 软件的应用领域。

目前面向对象方法是软件开发的主流方法，因此面向对象语言的选择问题更受关注。开发人员在选择面向对象语言时，应该着重考虑以下一些实际因素：

(1) 将来能否占主导地位。

(2) 可复用性。

(3) 类库和开发环境。

(4) 其他因素。

11.2 编 码 风 格

11.2.1 命名

程序设计过程要涉及到对变量、常量、函数、类、对象等编程元素进行命名。一个变量的作用域越大，它的名字所携带的信息就应该越多。

下面是一些通用的规则：

(1) 标识符的命名应当直观，可以望文知义，最好采用英文单词或其组合。

(2) 标识符的长度应当符合“最小长度下的最大信息”原则，过长的英文单词应该采用一些通用而合理的缩写或者应用领域专业术语的缩写。

(3) 程序中不要出现仅依靠大小写来区分的相似标识符。

(4) 程序中不要出现局部变量和全局变量同名的现象，以免引起误解。

(5) 变量名应当使用“名词”或者“形容词 + 名词”的形式。

(6) 函数名应当使用“动词”或者“动词 + 名词”的形式。

例11.1 Java命名实例。

```
package org.jr.jzj.editor;


import java.awt.*;
import javax.swing.*;
```

```
public class LineNumber extends JComponent {
    private final static Color DEFAULT_BACKGROUND =
Color.white;
    private final static Color DEFAULT_FOREGROUND = new
Color(153, 153, 204);
    private final static Color DEFAULT_LINECLR = new
Color(192, 192, 192);
    private final static Font DEFAULT_FONT = new
Font("SansSerif", Font.PLAIN, 12);
    private final static int HEIGHT = Integer.MAX_VALUE -
1000000;
    private final static int MARGIN = 5;
```

```
 private FontMetrics fontMetrics;
private int lineHeight;
private int currentRowWidth;
private JComponent component;
private int componentFontHeight;
private int componentFontAscent;
```

```
public LineNumber(JComponent component) {
    if (component == null) {
        setBackground(DEFAULT_BACKGROUND);
        setForeground(DEFAULT_FOREGROUND);
        setFont(DEFAULT_FONT);
        this.component = this;
    }
```

```
else {
        setBackground(DEFAULT_BACKGROUND);
        setForeground(component.getForeground());
        setFont(component.getFont());
        this.component = component;
    }
```

```
    componentFontHeight =
component.getFontMetrics(component.getFont()).getHeight();
     componentFontAscent =
component.getFontMetrics(component.getFont()).getAscent();
     setPreferredWidth(9999);

this.setBorder(BorderFactory.createLineBorder(DEFAULT_LINEC
LR, 1));
   }
```

```
public void setPreferredWidth(int row) {
        int width = fontMetrics.stringWidth(String.valueOf(row));
        if (currentRowWidth < width) {
            currentRowWidth = width;
            setPreferredSize(new Dimension(2 * MARGIN + width,
HEIGHT));
        }
    }
```

```
public void setFont(Font font) {
        super.setFont(font);
        fontMetrics = getFontMetrics(getFont());
    }

    public int getLineHeight() {
        if (lineHeight == 0) {
            return componentFontHeight;
        }
        else {
            return lineHeight;
        }
}
```

```
public void setLineHeight(int lineHeight) {
        if (lineHeight > 0) {
            this.lineHeight = lineHeight;
        }
    }

    public int getStartOffset() {
        return component.getInsets().top + componentFontAscent;
    }
```

```
 public void paintComponent(Graphics g) {
        int lineHeight = getLineHeight();
        int startOffset = getStartOffset();
        Rectangle drawHere = g.getClipBounds();
        g.setColor(getBackground());
        g.fillRect(drawHere.x, drawHere.y, drawHere.width,
drawHere.height);
        g.setColor(getForeground());
```

```
        int startLineNumber = (drawHere.y / lineHeight) + 1;
        int endLineNumber = startLineNumber + (drawHere.height /
lineHeight);
        int start = (drawHere.y / lineHeight) * lineHeight +
startOffset;
        for (int i = startLineNumber; i <= endLineNumber; i++) {
          String lineNumber = String.valueOf(i);
          int width = fontMetrics.stringWidth(lineNumber);
```

```
            g.drawString(lineNumber, MARGIN + currentRowWidth -
width, start);
             start += lineHeight;
          }
          setPreferredWidth(endLineNumber);
      }
}
```

11.2.2　注释

注释是帮助阅读和理解程序的有效手段，用自然语言或伪码描述。注释说明了程序的功能，特别是在维护阶段，对理解程序提供了明确的指导。

书写注释应该注意以下问题：

(1) 程序中的注释不宜过多，否则会使人眼花缭乱。

(2) 不必要注释含义已经十分清楚的代码。

(3) 修改代码时应该同时修改注释，以保证代码和注释的一致性。

(4) 注释应该准确易懂，防止出现二义性，错误的注释不但无益而且有害。

(5) 注释的位置应该与被描述的代码相邻，应该写在程序代码的上方并且和代码左对齐。

(6) 变量定义和分支语句必须写注释，因为这些语句往往是程序某一特定功能的关键。

例11.2 Java注释实例。

```
package com.lowagie.text;
import java.net.MalformedURLException;
/**
 * A <CODE>Watermark</CODE> is a graphic element (GIF or JPEG)
 * that is shown on a certain position on each page.
 *
 * @see                 Element
 * @see                 Jpeg
 * @see                 Gif
 * @see                 Png
 */
```

```
public class Watermark extends Image implements Element {

    // member variables

/** This is the offset in x-direction of the Watermark. */
    private float offsetX = 0;

/** This is the offset in y-direction of the Watermark. */
    private float offsetY = 0;
```

```
        // Constructors


/**
 * Constructs a <CODE>Watermark</CODE>-object, using an
<CODE>Image</CODE>.
 *
 * @param      image          an <CODE>Image</CODE>-object
 * @param      offsetX        the offset in x-direction
 * @param      offsetY        the offset in y-direction
 */
```

```
 public Watermark(Image image, float offsetX, float offsetY)
throws MalformedURLException {
        super(image);
        this.offsetX = offsetX;
        this.offsetY = offsetY;
    }

    // implementation of the Element interface
```

```
/**
 * Gets the type of the text element.
 *
 * @return       a type
 */

   public int type() {
       return type;
   }

   // methods to retrieve information
```

```
/**
 * Returns the offset in x direction.
 *
 * @return                  an offset
 */


   public float offsetX() {
       return offsetX;
   }
```

```
/**
 * Returns the offset in y direction.
 *
 * @return                  an offset
 */


    public float offsetY() {
        return offsetY;
    }
}
```

11.2.3　源代码版式

1．适当的空行

在源代码中适度地使用空行可以使程序的结构更加清晰，空行分隔一般出现在：

- 源文件中的各个节之间。
- 源文件中的类与类、类与接口之间。
- 方法定义之间。
- 局部变量定义和第一个语句之间。
- 注释块或注释行之前。
- 语句的不同逻辑分段之间。

2．代码行及行内空格

● 一行代码只写一条语句，避免使用复杂的语句行。

● 行内空格一般出现在关键字和括号之间、参数列表中逗号之后、二元运算符之间、语句中的表达式之间以及赋值符号之后。

例11.3 代码行风格实例及其修改结果。

(1) 变量定义。

避免使用：

```
int width, height;                    //宽度和高度
```

应该采用：

```
int width = 10;
int height = 10;
```

(2) 运算表达式。

避免使用：

```
x = a + b; a += c + d;
```

应该采用：

```
x = a + b;
a += c + d;
```

(3) 其他语句。

避免使用：

```
prints(" size is " +foo+" \n");
```

应该采用：

```
prints(" size is " + foo +" \n");                    //增加了空格
```