PhoneMe

软件设计模式文档

kde9

孔祥欣	**Kfirst**
胡玮玮	**deepsolo**
卿培	**edwardtoday**

2009 年 5 月

TABLE OF CONTENTS

1 引言 4

1.1 编写目的 4

1.2 项目背景 4

1.2.1 项目编号 4

1.2.2 软件名称 4

1.2.3 任务提出者 4

1.2.4 开发者 4

1.3 参考文献 4

2 设计模式说明 5

2.1 Factory 模式 5

2.1.1 ControllerFactory 5

2.1.2 ModelFactory 5

2.2 Singleton 模式 6

2.2.1 ConfigFactory 6

2.3 Façade 模式 7

2.3.1 ReadFile 7

2.3.2 WriteFile 9

2.4 Proxy 模式 12

2.4.1 Save 13

3 文件系统说明 15

3.1 allname 15

3.2 group 15

3.3 card.. 15

1 引言

1.1 编写目的

本文档对 PhoneMe 软件的设计模式进行简要说明，并确定了软件使用的文件系统，使项目设计和评审人员能够对项目有比较深的认识和了解。

1.2 项目背景

1.2.1 项目编号

200903001

1.2.2 软件名称

PhoneMe 基于 Java 语言的单机版个人通讯录

1.2.3 任务提出者

《软件工程》课程

1.2.4 开发者

kde9 开发小组

1.3 参考文献

1.3.1 PhoneMe 需求文档_kde9.pdf

1.3.2 PhoneMe 设计文档_kde9.pdf

1.3.3 PhoneMe 需求文档_kde93_r2

1.3.4 PhoneMe 设计文档_kde9_rev2

1.3.5 2005011301_177473_848803779_iWallet 设计文档 2.0_820204417.pdf

1.3.6 Roger S.Pressman 著，梅宏译，《软件工程——实践者的研究方法》，机械工业出版社，2002 年

2 设计模式说明

在设计的过程中，我们在如下几个特定的地方使用了特定的设计模式

2.1 FACTORY 模式

工厂（Factory）模式是一个比较常用的设计模式，主要用于根据所提供的数据返回某一类的一个实例。在我们的设计中有如下两处用到了工厂模式，分别为产生 controller 和 model 。

2.1.1 CONTROLLERFACTORY

在 ControllerFactory 中，有 createAllNameController、createCardController 、createGroupController 等三个静态工厂方法用来生产不同的 controller。具体代码实现如下：

```
public class ControllerFactory {

    public static AllNameController createAllNameController() {

        return new MyAllNameController();

    }


    public static CardController createCardController() {

        return new MyCardController();

    }


    public static GroupController createGroupController() {

        return new MyGroupController();

    }

}
```

2.1.2 MODELFACTORY

在 ModelFactory 中，有 createCard、createGroup、createAllName 等三个静态工厂方法用来生产不同的 model。具体代码实现如下：

```
public class ModelFactory {

    public static Card createCard(int id) {

        return new MyCard(id);

    }


    public static Group createGroup(int id) {

        return new MyGroup(id);

    }


    public static AllName createAllName() {

        return new MyAllName();

    }

}
```

2.2 SINGLETON 模式

单例（Singleton）模式也比较常见，它主要用来保证一个类有且仅有一个实例，在我们的设计中有如下一处使用了该模式。

2.2.1 CONFIGFACTORY

Configuration 是我们用来管理配置文件的一个类，而这个类的实例只需要一个，因此我们在 ConfigFactory 类中通过一个 static 变量和一个 static 方法 creatConfig 达到这一目的。具体代码实现如下：

```
public class ConfigFactory {

    static Configuration configuration = new Configuration();
```

```
    public static Configuration creatConfig() {
        return configuration;
    }
}
```

2.3 FAÇADE 模式

外观（Façade）模式可以用于将一组复杂的类打包到一个较为简单的外部接口中。在我们的设计中，在进行文件系统的 IO 操作时，我们使用了外观模式。

2.3.1 READFILE

我们将读文件的操作封装到 ReadFile 类中，这样当我们的操作需要读文件系统的时候，我们只需要产生 ReadFile 的一个实例，然后调用其中的 readLine、close 方法即可，这样就不需要在每一次读取文件的时候都通过 IO 流来操作，使得这一动作被封装到一个较为简单的接口中。具体代码实现如下：

```
public class ReadFile {
    /**
     * 要操作的文件名
     */
    String fileName;
    FileReader fr;
    BufferedReader br;

    /**
     * 通过文件名构造一个文件read流
     * @param fileName 文件名
     * @throws FileNotFoundException 要打开的文件不存在
     */
```

```
public ReadFile(String fileName)
throws FileNotFoundException {
    this.fileName = fileName;
    fr = new FileReader(fileName);
    br = new BufferedReader(fr);
}

public ReadFile(File file)
throws FileNotFoundException {
    this.fileName = file.getPath();
    fr = new FileReader(file);
    br = new BufferedReader(fr);
}

/**
 * 从文件中读取一行
 * <br><strong>
 * 使用时注意，文件读操作结束后要调用close方法关闭流。
 * </strong></br>
 * @return 包含要读取行内容的String
 * @throws IOException
 */
synchronized public String readLine()
throws IOException {
```

```
        return br.readLine();
    }

    /**
     * 关闭流
     * @throws IOException
     */
    synchronized public void close()
    throws IOException {
        br.close();
        fr.close();
    }
}
```

2.3.2 WRITEFILE

我们将写文件的操作封装到 WriteFile 类中，这样当我们的操作需要写入文件系统的时候，我们只需要产生 WriteFile 的一个实例，然后调用其中的 write 、writeLine、close 方法即可，这样就不需要在每一次写文件的时候都通过 IO 流来操作，使得这一动作被封装到一个较为简单的接口中。具体代码实现如下：

```
public class WriteFile {
    /**
     * 要操作的文件名
     */
    String fileName;
    String contant;
    FileWriter fw;
```

```
BufferedWriter bw;


/**
 * 通过文件名和标识符创建文件write流
 * @param fileName 文件名
 * @param flag 为true时表示追加，为false时表示覆盖。
 * @throws IOException
 */
public WriteFile(String fileName, boolean flag)
throws IOException {
    this.fileName = fileName;
    fw = new FileWriter(fileName, flag);
    bw = new BufferedWriter(fw);
}


/**
 * 通过文件名和标识符创建文件write流
 * @param fileName 文件名
 * @param flag 为true时表示追加，为false时表示覆盖。
 * @throws IOException
 */
public WriteFile(File file, boolean flag)
throws IOException {
    this.fileName = file.getPath();
```

```
        fw = new FileWriter(file, flag);
        bw = new BufferedWriter(fw);
    }

    /**
     * 向文件中写入内容
     * <br><strong>
     * 使用时注意，文件写操作结束后要调用close方法关闭流。
     * </strong></br>
     * @param str 要写入的内容
     * @return
     * @throws IOException
     */
    synchronized public void write(String str)
    throws IOException {
        bw.write(str);
    }

    /**
     * 向文件中写入内容，并换行
     * <br><strong>
     * 使用时注意，文件写操作结束后要调用close方法关闭流。
     * </strong></br>
     * @param str 要写入的内容
```

```
     * @throws IOException
     */
    synchronized public void writeLine(String str)
    throws IOException {
        bw.write(
                str +
                System.getProperty("line.separator") );
    }

    /**
     * 关闭流
     * @throws IOException
     * @throws IOException
     */
    synchronized public void close()
    throws IOException {
        bw.close();
        fw.close();
    }
}
```

2.4 PROXY 模式

代理（Proxy）模式采用更简单的对象来表示那些复杂的，或者创建耗时的对象。如果创建一个对象所需的时间或者计算资源很昂贵，代理模式能将创建过程推迟到真正需要这个对象时才完成。

2.4.1 SAVE

在MyAllNameController、MyCardController、MyGroupController等类中均有save方法，而在我们的设计中如果瞬间添加很多个表项，这个时候调用save方法的时候可能会遇到资源占用过多的问题，进而导致系统卡死。因此，在这里采用代理模式可以解决这个问题。具体代码实现如下：

```
public class Save

implements Constants{

    private String pathAndName;

    private String content;

    private Thread thread;


    public Save() {

        thread = new Thread() {

            public void run() {

                try {

                    WriteFile wf = new WriteFile(pathAndName, false);

                    wf.write(content);

                    wf.close();

                } catch (IOException e) {}

            }

        };

    }


    public void init(String pathAndName, String content) {

        this.pathAndName = pathAndName;
```

```
        this.content = content;
    }

    public boolean save() {
        thread.start();
        return true;
    }
}
```

3 文件系统说明

数据文件共分为三类，allname、group 和 card。

3.1 ALLNAME

其中 allname 文件只有一个，文件名为‘Allname’，为一个检索文件，这个文件中保存了所有人的 id 与名字对。

保存的格式是，每一项占一行，每三行为一组，保存一个人的 id、firstName、lastName，文件尾会有一空行。

3.2 GROUP

group 文件可以有很多，保存每个 group 的名字和 group 成员的 id，文件名为该组的 id。

保存的格式是，第一行保存 group 的名字，以后每行保存一个 group 成员的 id。文件尾会有一空行。

3.3 CARD

card 文件可以有很多，保存每个 card 的名字，表项，关系等。文件尾会有一空行。

保存的格式是，第一行保存 firstName，第二行保存 lastName。

接下来是保存表项的部分，表项包括 key 和 value 两部分。
每个表项的第一行为表项的 key，以后若干行为表项的 value，每个 value 的结束是以一行特殊的字符串叫做‘valueSeparator’为标志的，一个 key 可以对应多个 value。
一个表项的结束是以一个特殊的字符串叫做‘itemSeparator’为标志的。

表项部分可以包括多个表项。若没有表项部分，则接下来是一行‘valueSeparator’，再一行‘itemSeparator’。再接下来是关系部分，这部分每两行为一组，保存与当前联系人有关系的联系人的 id 和关系的类型。

下面给出一个 card 文件实例（其中‘*&^%$#’为 valueSeparator，‘~!:{?’为 itemSeparator}：

```
孔
祥欣
班级
计 62
*&^%$#
爱好
乒乓球
*&^%$#
发呆
*&^%$#
~!:{?
2
朋友
3
同学
//This is an empty line
```

其中 孔 为 firstName， 祥欣 为 lastName，班级和爱好为表项的 key，计 62、乒乓球和发呆为表项的 value，该联系人与 id 为 2 的联系人为朋友关系，与 id 为 3 的联系人为同学关系。