医学信息集成技术实验报告

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **学 号：** | **1519640532** | **姓 名：** | **戴维** | **实验成绩：** |  |
| **实验名称：** | **HL7消息处理类设计** | | | **实验日期：** | **2019-11-11** |

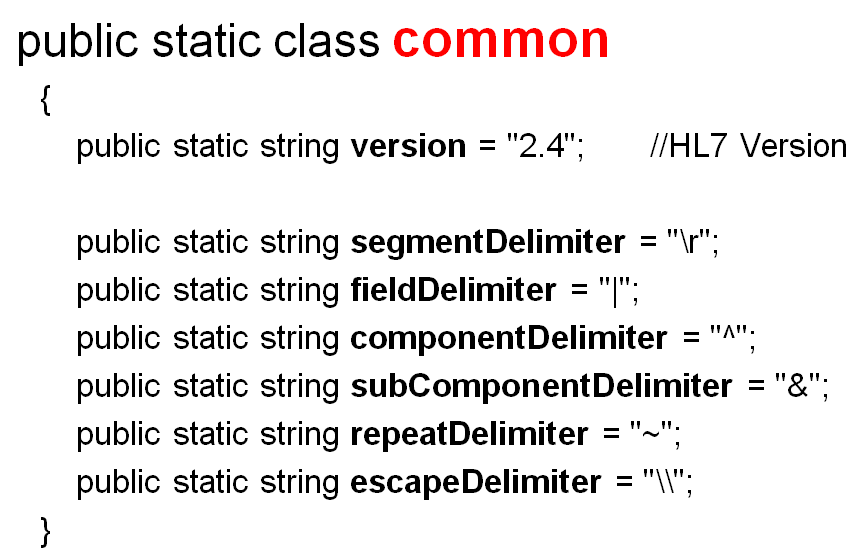
1. **实验目的：**

设计并编码HL7底层类库。

1. **实验内容：**

**一、** 新建项目，选择类库模板，解决方案名为HL7LIB。使用VS的UML类图建立HL7组合模式基类。

1. 建立common类：



1. 建立抽象构件abstractType:
   * 1. 构造函数pubic abstractType(string name), 将name保存到this.name中
     2. 保护字段name和value，均为protected string
     3. 抽象方法public abstract bool Parse(string text);
     4. 抽象方法public abstract override string ToString();
     5. 只读属性public string Name，提供对字段name的访问
     6. 只读属性public string Version，提供对common.version的访问
     7. 读写属性public string Value，其get访问器调用ToString，set访问器调用Parse(value)
2. 建立叶子构件primitiveType，继承abstractType，在类详细信息中添加：
   * 1. 构造函数public primitiveType(string name)：将name传递给基类构造函数
     2. 重写方法public override bool Parse(string text): 将text保存到value字段中
     3. 重写方法public override string ToString(): 返回value字段的值。
3. 建立容器构件compositeType： 继承abstractType，添加以下成员：
   * 1. 构造函数public compositeType(abstractType parent, string name)：为基类构造函数提供参数，并通过上层容器构件parent.delimiter字段的值确定本容器构件的分隔符
     2. 字段protected abstractType[] data: 所容纳的子组件数组
     3. 字段public string delimiter：存放本层的分隔符
     4. 重写方法public override bool Parse(string text): 将text用本容器的分隔符分割后，分别调用各个子组件的Parse方法。
     5. 重写方法public override string ToString(): 调用各子组件的ToString方法，然后加上本容器的分隔符作为返回值。
4. 进入代码视图，详细实现各个类成员，编译通过。

二、增加基类的单元测试

1. primitive类测试：在primitiveTypetest方法中增加如下代码，测试Name属性和Version属性：

primitiveType obj = new primitiveType("ID");

Assert.AreEqual("ID",obj.Name);

obj.Value = "ABC123456";

Assert.AreEqual("ABC123456", obj.Value);

右键执行”运行测试”或”调试测试”，观察测试是否成功，如不成功，修改primitiveType类的代码，直至成功。

1. 增加对其他方法的测试：

public void ParseTest()

{

primitiveType obj = new primitiveType("TS");

string ts = DateTime.Now.ToString("yyyyMMddhhmmss.fff");

obj.Parse(ts);

Assert.AreEqual(ts, obj.Value);

DateTime dt = DateTime.Now;

obj.Parse(dt.ToString("yyyy年MM月dd日hh时mm分"));

Assert.AreEqual(dt.ToString("yyyyMMddhhmm"), obj.Value);

}

public void ToStringTest()

{

……….

}

运行测试直至成功。

1. 增加对compositeType类的测试。

三、实现具体叶子构件类

1. 建立ID类，从primitiveType类派生，在构造函数中为基类构造函数传递参数，详细实现并编译通过。
2. 通过在类图中复制然后改名的方法建立其他类：IS类 NM类 ST类 DT类 TM类 TN类 TX类 FT类 TS类 SI类
3. **程序及注释：**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace HL7Lib

{

public static class Common

{

public static string version = "2.4";

public static char segmentDelimiter = '\r';

public static char fieldDelimiter = '|';

public static char componentDelimiter = '^';

public static char subComponentDelimiter = '&';

public static char repeatDelimiter = '~';

public static char escapeDelimiter = '\\';

}

public abstract class AbstractType

{

protected string name;

protected string value;

public string Name { get { return this.name; } }

public string Version { get { return Common.version;} }

public string Value

{

get { return this.ToString(); }

set { this.Parse(value); }

}

public AbstractType(string name)

{

this.name = name;

}

public abstract bool Parse(string text);

public abstract override string ToString();

}

public class PrimitiveType : AbstractType

{

public PrimitiveType(string name) : base(name)

{

base.name = name;

}

public override string ToString()

{

switch(this.name)

{

case "TS":

break;

case "ID":

break;

default:throw new Exception("Data type not found!");

}

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class CompositeType : AbstractType

{

public char delimiter;

protected AbstractType[] data;

public CompositeType(CompositeType parent, string name) : base(name)

{

base.name = name;

if (parent == null)

delimiter = Common.segmentDelimiter;

else if (parent.delimiter == Common.segmentDelimiter)

delimiter = Common.fieldDelimiter;

else if (parent.delimiter == Common.fieldDelimiter)

delimiter = Common.componentDelimiter;

else if (parent.delimiter == Common.componentDelimiter)

delimiter = Common.subComponentDelimiter;

}

public override string ToString()

{

string result = "";

foreach(AbstractType d in data)

{

result += d.ToString() + this.delimiter;

}

return result;

}

public override bool Parse(string text)

{

this.value = text;

string[] splitResult = text.Split(this.delimiter);

this.data = new AbstractType[splitResult.Length];

for(int i = 0;i<data.Length;i++)

{

if (splitResult[i] == null || splitResult[i].Length == 0) continue;

data[i] = GetSubDataType(this.name,i);

data[i].Parse(splitResult[i]);

}

return true;

}

private AbstractType GetSubDataType(string name,int idx)

{

switch(name)

{

case "EVN":

switch(idx)

{

case 0: return new PrimitiveType("ID");

case 1: return new PrimitiveType("TS");

case 2: return new PrimitiveType("TS");

case 3: return new PrimitiveType("IS");

case 4: return new PrimitiveType("XCB");

case 5: return new PrimitiveType("TS");

default: throw (new Exception("该元素未找到定义！"));

}

default: throw (new Exception("该元素未找到定义！"));

}

throw (new Exception("该元素未找到定义！"));

}

}

public class ID : PrimitiveType

{

public ID(string name):base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class SI : PrimitiveType

{

public SI(string name) : base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class TS : PrimitiveType

{

public TS(string name) : base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class FT : PrimitiveType

{

public FT(string name) : base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class TX : PrimitiveType

{

public TX(string name) : base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class TN : PrimitiveType

{

public TN(string name) : base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class TM : PrimitiveType

{

public TM(string name) : base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class DT : PrimitiveType

{

public DT(string name) : base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class ST : PrimitiveType

{

public ST(string name) : base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

return true;

}

}

public class NM : PrimitiveType

{

public NM(string name) : base(name)

{

this.name = name;

}

public override string ToString()

{

return value;

}

public override bool Parse(string text)

{

this.value = text;

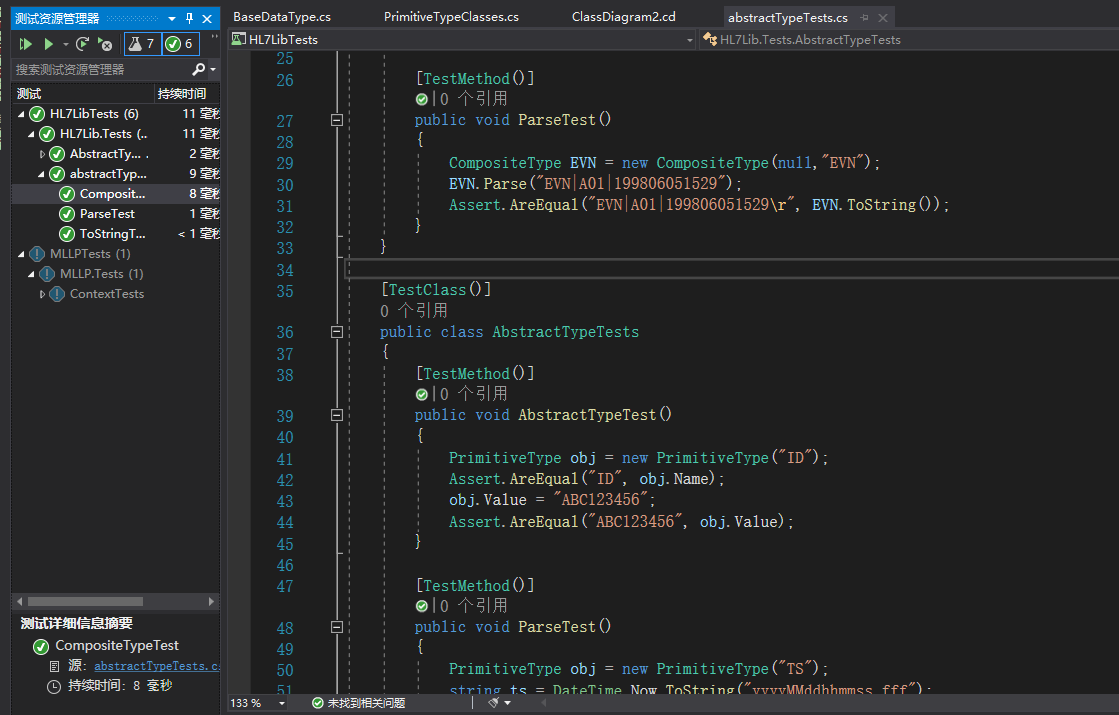
return true;

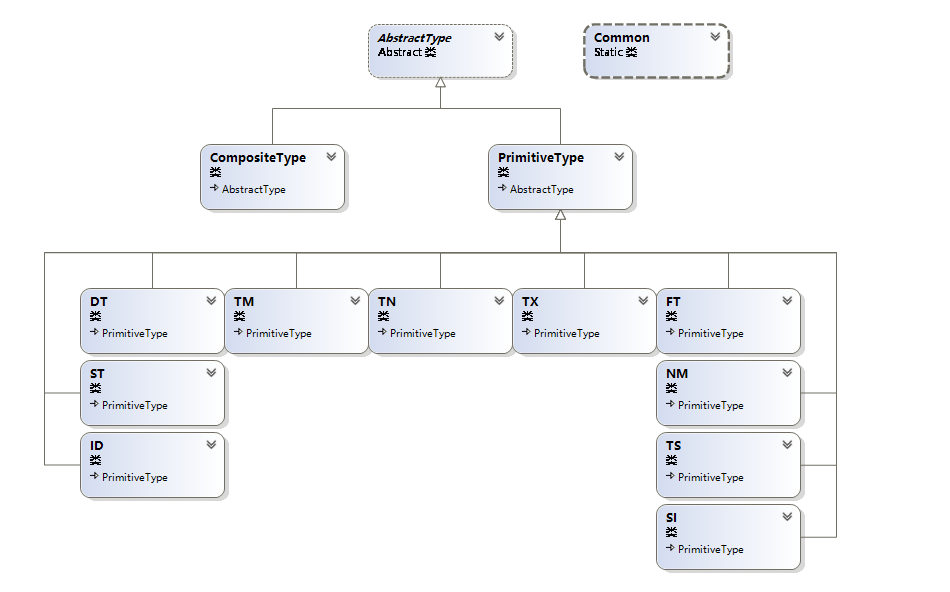
}

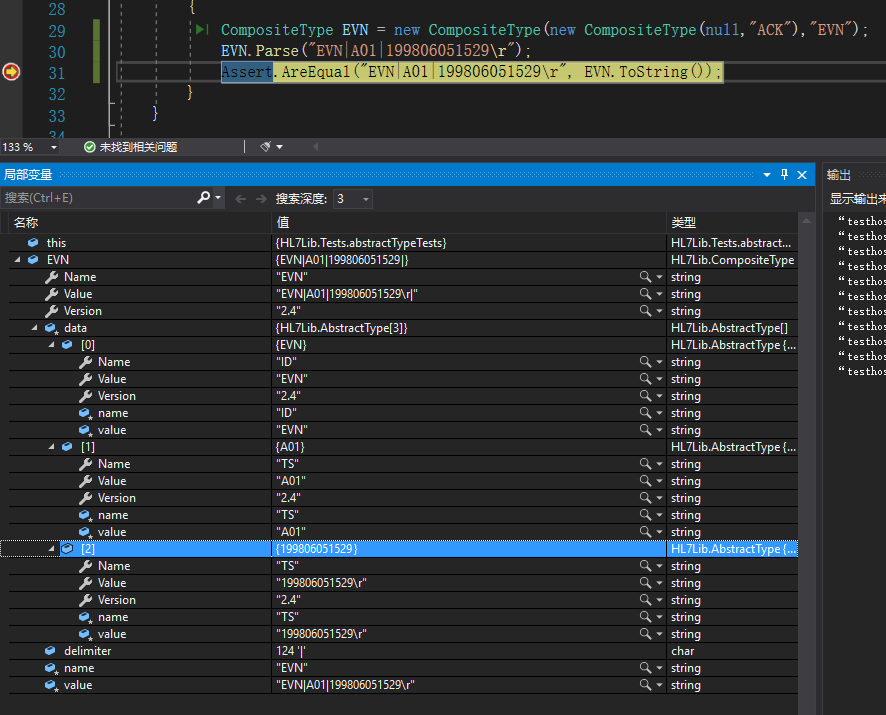
}

}

**四.运行结果**







**五.小结(不少于200字)**

本次实验中采用组合模式，将HL7中不同层级的对象进行分类，如消息（复杂类型）、段（复杂类型）、字段（复杂或简单类型）、组件（复杂或简单类型）、子组件（简单类型）等。其中，只分了复杂和简单两种类型，通过段定义表进行查询，对相应index下的数据分配相应类型、名称等详细信息。若分配为复杂类型，则在其中部署一个列表存储其子类型；若为简单类型（叶子组件），则设置好基本属性后，调用其Parse方法解析其value。

简单类型的代码可用上一次实验的代码生成器完成。

每个类型内部的Parse和根据段定义表查询子类型可以用正则表达式筛选来完成。