Some Approaches to Inheritance-Based Class Interface Extension

Ivan S. Veličković[1] and Marija D. Cvetković[2]

Abstract – **A brief discussion of inheritance-based class interface extension, its properties and applicability will be given in this paper. Two alternatives will be analysed: an approach based on multiple inheritance and approach based on nested classes. These approaches will be compared and possible application issues will be considered.**

Keywords – **Design Patterns, Inheritance, Nested Classes**

I. INTRODUCTION

The object-oriented approach is today's most exploited model for development of large software products. Simple reuse by means of inheritance and limitations over relations between entities (objects and classes) imposed by the encapsulation concept make this method suitable for simple and reliable integration and debugging of independent software components developed by different teams of developers. But a problem remains: following the rules imposed by object-oriented model doesn't necessarily lead to a good, reusable and decoupled solution. For this reason most modern software development techniques, faced with growing market and rising customer requirements, depend on reuse of well-known, well-tested and extendable but comprehensible object-oriented design solutions, known as Design Patterns [1].

In the development process of a complex application, user interactions (use cases) are usually grouped into separate and independent functional units (components, applications). These units operate over the same data and offer to a user another set of possible activities. From a developers point of view this means the implementation of a different interface for each independent unit. This burden of unit dependent interface methods, if placed in the same data implementation class, could make such a class difficult to maintain and modify. The solution is to separate universal, unit independent, methods of the data implementation class from, unit dependant methods. Different approaches could be used in order to achieve such goal. This problem is partially addressed by the *Bridge* design pattern [1]. *Bridge* offers a delegation-based solution for interface separation that is easy to extend, maintain and modify. Unfortunately, its application is difficult over data implementation classes (*Concrete Implementers*) that are, though inherited from the same base class, very different conceptually and architecturally. Another approach could be based on *Adapter* design pattern. Essentially, "adapting" one interface to another is what we are trying to achieve. On the other hand, introduction of a unit dependant interface can vary not just class external behaviour, but also its inner implementation. Probably the most intuitive choice could be the *Visitor* pattern. From a point of view, suggested solutions can be treated as an inheritance-based approaches to *Visitor* pattern realisation. Both approaches suggested in this paper offer a possible solution to this problem that cannot be elegantly solved by direct application of mentioned related design patterns.

This paper is organised as follows. Next section discusses a malicious behaviour of software reuse through inheritance. A method for encapsulation breach by means of friend class declaration of an inherited class is presented. Section III contains a brief description of proposed solutions. Finally the Section IV gives the application example of proposed methods.

Code samples are given in C++ language and UML diagrams follow Rational Unified notation.

II. INHERITANCE AND ENCAPSULATION DECAY

Inheritance, polymorphism and encapsulation are considered as base concepts of object-oriented programming. These concepts, if properly utilized, are means of safe software code reuse. The problem of good and safe object-oriented design relies on thin balance between these concepts. For the sake of simplicity lets consider the following example.

```
class CBaseClass
{
public:
     int GetPrivate() {return m_nPrivate;};
protected:
     int m_nProtected;
private:
     int m_nPrivate;
};

class CInheritedClass: public CBaseClass
{
friend class CUserClass; // encapsulation breach
...
};
class CUserClass
{
protected:
     CInheritedClass* pData;
...
};
```

Code Sample 1. An illustration of encapsulation breach with inheritance

By means of public inheritance access to all protected and public attributes of the parent class is given to all descendants. However, private members remain class-local, and can be

[1] Ivan S. Veličković is with Faculty of Electronics Engineering Aleksandra Medvedeva 14, 18000 Nis, Serbia and Montenegro, E-mail: ivanv@elfak.ni.ac.yu

[2] Marija D. Cvetković is with Faculty of Electronics Engineering Aleksandra Medvedeva 14, 18000 Nis, Serbia and Montenegro, E-mail: cveleglg@bankerinter.net

accessed only through public methods of the parent class, if such methods are available. In previous example such privileges are given to `CInheritedClass`. This means that by using this method of inheritance, children classes could gain less restrictive access to parent class implementation, which may result in encapsulation decay [2].

Notice a code line in the previous example that gives a friend privileges to `CUserClass`. This way the access to protected attributes of `CBaseClass` is given to a class that doesn't belong to the same family of classes. Should be mentioned that private members are still invisible to the friend-privileged class. This effect partially contradict well-known phrase that friendship is given, not inherited.

Another infamous inheritance topic is multiple inheritance. It is a method that can provide privileged access to a class implementation to another class that doesn't belong to the same family of classes. Because of its cumbersome behaviour this method of code reuse is often discouraged or forbidden by programming language syntax.

Regardless of the previous discussion proper use of mentioned techniques can represent a powerful tool.

III. SUGGESTED SOLUTIONS

Figure1 represents an UML class diagram of the multiple-inheritance-based solution. Abstract class *`CAbstractImp`* with pure virtual methods *`Operation1()`* and *`Operation2()`* defines unit (application or component) independent interface for implementation classes. Each inherited class implements this interface. On the other hand, unit specific interface is specified with abstract class *`CAppSpecificItfExt`*. Finally, unit specific concrete implementation is realized by means of multiple inheritance from unit specific interface definition class *`CAppSpecificItfExt`* and a corresponding unit independent concrete implementation class.

This solution can be realized in C++ through utilization of abstract classes and multiple inheritance, or in Java with interfaces and interface implementation. A problem arises if some additional *`CAppSpecificItfExt`* attributes and related methods need to be implemented. This would not be possible in Java and it is strongly discouraged in C++.

Figure 2 represents an UML class diagram of nested class based approach. The implementation class family is the same as in the previous discussion. Changes are made in unit specific interface extension classes. A pure virtual *`Bind()`* method is added to the interface of the abstract base class *`CAppSpecificItfExt`*. Purpose of this method is to enable *Visitor*-pattern-like biding between interface extension instance and corresponding implementation class instance. Implementation of the *`Bind()`* method, and a definition of a nested class inherited from the appropriate implementation class (classes with "X" prefixed names) are left to *`CAppSpecificItfExt`* child classes. Purpose of those nested classes is to provide a privileged reference (pointer) marked as `m_pItem`. Code Sample 2 describes this idea in detail. Additional implications are given through code comments.

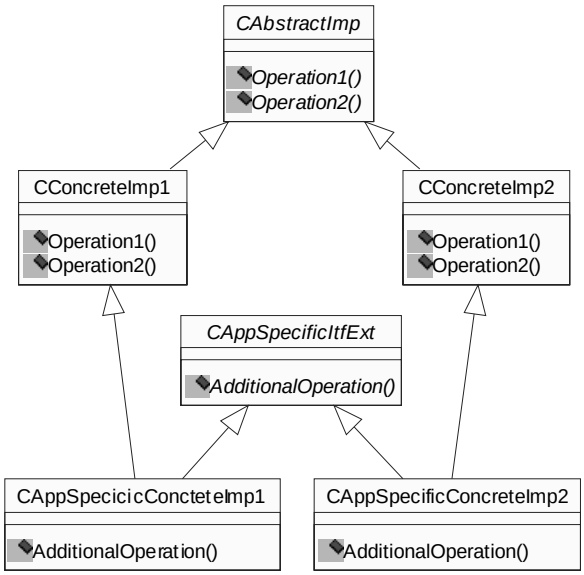

Figure 1. First solution based on multiple inheritance

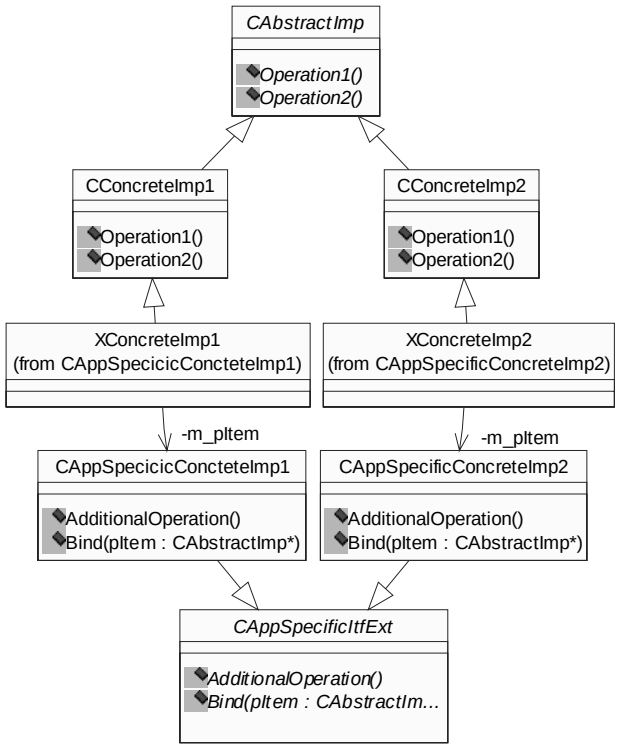

Figure 2. Second solution based on nested friend class definition

```
////////////////////////////////////////////////
class CAbstractImp
{
private:
    int m_nParentPrivate;
protected:
```

```
    int m_nParentProtected;
public:
    virtual enum eType GetType() = 0;
    // unit independent interface declaration
    virtual void Operation1() = 0;
    virtual void Operation2() = 0;
};

////////////////////////////////////////////////
class CConcreteImp1: public CAbstractImp
{
private:
    int m_nChildPrivate;
protected:
    int m_nChildProtected;
public:
    enum eType GetType() {
       return TYPE_CONCRETE_IMP_1;};
    // unit independent interface implementation
    virtual void Operation1();
    virtual void Operation2();
};

////////////////////////////////////////////////
class CAppSpecificItfExt
{
public:
    // unit dependant interface declaration
    virtual bool Bind(CAbstractImp* pItem) = 0;
    virtual void AditionalOperation() = 0;
};

////////////////////////////////////////////////
class CAppSpecificConcreteImp1: public
CAppSpecificItfExt
{
    class XConcreteImp1: public CConcreteImp1
    {
       friend class CAppSpecificConcreteImp1;
    } m_pItem;
    ...
public:
    bool Bind(CAbstractImp* pItem)
    {
       if(pItem->GetType() == TYPE_CONCRETE_IMP_1){
              m_pItem = (XConctreteImp*)pItem;
              return true;   // bind success
       }
       m_pItem = NULL;
       return false;         // bind failure
    }; // end method Bind()
    void AditionalOperation()
    {
       ...
       // somewhere in the code
       if(m_pItem){
              m_pItem->m_nParentProtected++;
              m_pItem->m_nChildProtected++;

              // following lines would not compile
              // m_pItem->m_nParentPrivate++;
              // m_pItem->m_nChildPrivate++;
       }
    }; // end method AditionalApplication
}; // end class CAppSpecificConcreteImp1
```

Code Sample 2. Code illustration of the second approach

Method described in Code Sample 1 is utilized by this approach to gain privileged access to protected members of implementation classes.

This approach can be implemented in C++ and Java without significant conceptual changes. Greatest disadvantage of this method is its complexity. Before manipulation over an instance of corresponding `CConcreteImp` an interface extension class instance should be bond with it. To reduce this drawback the *`CAppSpecificItfExt`* class can be used to maintain a *Flyweight*-pattern-like instance pool of interface extension objects [1]. Code Sample 3 illustrates this idea in detail. `UniversalBind()` static method is used to make binding process more transparent.

```
class CAppSpecificItfExt
{
    ...
private:
    static CAppSpecificItfExt*
       sm_arrPool[NUM_APP_SPECIFIC_IMPS];
public:
    static CAppSpecificItfExt*
       UniversalBind(CAbstractImp* pItem)
    {
       int index = (int)pItem->GetType();
       sm_arrPool[index]->Bind(pItem);
       return sm_arrPool[index];
    };

    static void CreatePool()
    {
       sm_arrPool[TYPE_CONCRETE_IMP_1] =
              new CAppSpecificConctreteImp1;

       sm_arrPool[TYPE_CONCRETE_IMP_1] =
              new CAppSpecificConctreteImp1;

    };
}
```

Code Sample 3. Flyweight-like implementation of unit specific interface object management.

Another possible simplification could be parameterisation of the *`CAppSpecificItfExt`* class. By passing appropriate X-prefixed class as a type parameter complete binding process would be placed into *`CAppSpecificItfExt`* class. This approach requires that global scope should be given to X-prefixed classes, which may lead to class count explosion and increased ambiguity.

IV. An Application Example

Both suggested solutions are tested with simulator of bipedal robot [3]. The simulator is constructed of several independent components: dynamics simulator, motion planer, 3D environment visualisator, 3D image interpreter and 3D environment and robot editor. Many of those are designed as independent applications and some of them communicate over network.

A coarse classification to editors and consumers can be adopted over these components. Editors provide user interface for creation and manipulation over relevant entities and consumers use these entities to perform required simulation tasks. Both type of components share the same entities definitions in form of concrete implementation classes. These definitions are centralized as independent library, which simplify debugging, and modification. Extensions of such definitions are component local. Editors are more user interface oriented. Their extensions can contain entity specific user interface objects like dialogs and property pages, and methods for user interaction events interpretation. On the other hand, consumer components are more simulation task oriented. Consumer components extensions are simulation specific and these extensions can contain simulation helper methods, methods for cross process/network boundary communication etc.

Nested classes approach appeared to be more appropriate for editor components. It is slower, more complex but more

intuitive in applications which require heterogeneous and massive interface extension. Also it is harder to maintain because some of its repetitive code segments are scattered over application specific interface extension classes.

Multiple inheritance approach appeared to be faster and easier to implement. No additional coding and binding is required, which makes it appropriate interface extension method for consumer components. However it suffers some limitation imposed by multiple inheritance.

As a result of immediate binding, both methods are sensitive to application independent classes modifications. This could be a great drawback in the early phases of application development, and great obstacle for further upgrades. However, because of great architectural differences no universal interface that would be used to separate class behaviour and its implementation can be defined.

V. CONCLUSION

This paper discussed two inheritance-based methods for class interface extension. Some of their advantages and drawbacks are exposed. Commented code examples are given to support this review. Both approaches are resulted from a practical problem proposed by a specific application request. Although application specific proposed solutions may help to overcome some general limitations imposed by object-oriented design process.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, 1995.

[2] A. Snyder, "Encapsulation and inheritance in object-oriented languages", *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland, Oregon, 1986.

[3] G. S. Đorđević, N. Vukić, I. Veličković, I. Jovanović, M. Rašić, M. Vukobratović, "Visually Interactive Robot Simulator", *TELSIKS 2003 Conference Proce*edings, Niš, Serbia and Montenegro, 2003.