Mixin Classes and Layered Refinement

Sander van Loo

Department of Mathematics and Computer Science
Faculty of Sciences
Vrije Universiteit
Amsterdam, the Netherlands

sander@marketxs.com

January 22, 2003

Abstract

Mixins make it possible to introduce a new unit of modularity in software engineering, namely that of a collaboration. I explain what a mixin is and how mixin layers can be used to implement collaborations. Collaboration-based design is a new approach to software engineering promoting code re-use and streamlining large scale refinements. I will also show that layered programming is not limited to languages that provide native support for the mixin construct but, with a few tricks, can also be extended to languages such as Java.

Introduction

With the development of increasingly complex software systems, techniques for reducing this complexity by promoting modularity and reuse are becoming more important. As features are added, removed or modified unanticipated interactions and dependencies decrease the modularity of the software system and make the reuse of individual modules more and more difficult.

To address these issues a variety of modularization techniques have been developed that allow applications to be constructed from off-the-shelf components. Typical object-oriented approaches include single and multiple inheritance. However, the kind of modularity needed is often not that of a function, entire class or that of a package. In contrast we are seeking a unit of modularity that encapsulates fragments of multiple classes and thus fragments of multiple functions. For example a product line with related applications with slightly different feature sets may also require slightly different module implementations. This makes software development and maintenance a time consuming and difficult process.

A new approach to this problem is to use a construct that encapsulates each feature in a separate layer. Each layer contains the code for a single design feature, and layers can be composed using type equations. This allows programmers to refine applications by adding one layer at a time, a process called stepwise refinement.

Single Inheritance

Single inheritance is very common technique to implement stepwise refinement. Each subclass can add or change the behaviour of a single parent class. The new subclass usually offers a more specific service than its base class. Subclasses can both refine and extend their parent class. Refinement can be achieved by replacing an existing method entirely or by calling the method of the parent class from the new implementation of that method in the subclass. Examples of languages that support single inheritance are Smalltalk-80 and Java[1].

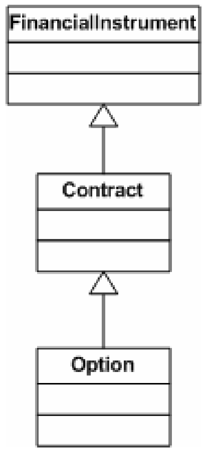

Figure 1: An example of single inheritance.

Consider a software product that provides services related to stock market data. One of the main components of this software product is an object model representing financial instruments. The FinancialInstrument class implements behaviour that is common for all types of financial instruments. However, numerous types of financial instruments exist, varying from straightforward equity stock to complex derivatives or contracts. A contract is a specialized from of a financial instrument, more concrete it has the property to expire at some time in the future. A contract can thus be implemented as a subclass of the FinancialInstrument class. The Contract class will implement the behaviour that is special to this type of financial instrument as well as inherit any behaviour that was already implemented in the FinancialInstrument class.

Multiple Inheritance

Multiple inheritance is a technique that allows a class to inherited behaviour from two or more classes. The term multiple inheritance is somewhat ambiguous. A distinction should be made between multiple implementation inheritance and multiple interface inheritance. Although multiple interface inheritance is nowadays considered legal, a large group of programmers still do not extend that view to multiple implementation inheritance. However, there are a number of cases where multiple implementation inheritance is a valuable mechanism that obviates the need for programmers to devise their own proprietary solutions, one of the most common being multiple specialization. Multiple specialization is needed when an object is conceptually a specialization of two (or

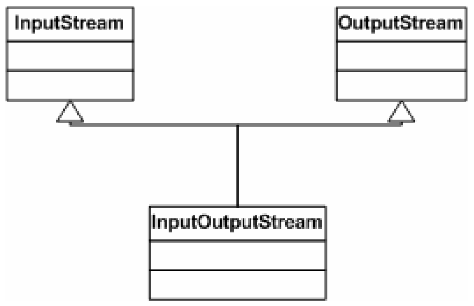

Figure 2: The canonical example of multiple implementation inheritance, the InputOutputStream.

[1] Although Java supports only single implementation inheritance it does support multiple interface inheritance, by allowing multiple interfaces to be implemented by an actual class.

more) different objects. The canonical example here is the InputOutputStream that is a specialization of an InputStream and an OutputStream.

Multiple inheritance is somewhat controversial because it poses problems if the functionality of one parent class partly overlaps with the functionality of one of the other parent classes[2]. Some kind of name resolution mechanism is needed if parent classes have identical methods. These mechanisms can be implicit (the language resolves them with an arbitrary rule), explicit (the programmer explicitly resolves the conflict in code) or the language simply does not allow conflicts at all. Although various solutions exist to resolve such conflicts there is no single best strategy. Examples of languages that support multiple inheritance are Eiffel and C++.

Collaboration-based Design

In object-oriented design, objects combine data and operations on that data in a single entity. However, objects are rarely self-sufficient. Although an object is responsible for maintaining the data that it encapsulates, objects often need to cooperate to accomplish a task. The objective of collaboration-based design is to decompose an application into objects and collaborations. A collaboration is a set of objects and a protocol for their interaction. In a collaboration all dependencies between classes that are specific to a particular service or feature are encapsulated. Thus collaborations can be viewed as reusable components that form the building blocks of an application.

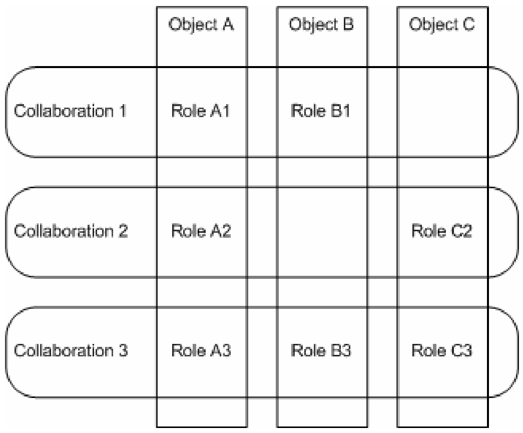

Figure 3: Collaboration-based design.

Mixin Classes

The inheritance mechanisms described above are, however, not symmetric. A superclass can be defined without defining its subclasses, but when a subclass is defined it must have its parent classes defined. If we want to extend a large number of classes in an existing application with the same functionality we need to copy the same code over and over again as we implement each new subclass. This is generally not considered good coding practice as it prevents code-sharing and abstraction. This problem is illustrated in figure 4.

[2] This is also known as the diamond problem. If the subclass C extends class A and class B and both A and B implement the method printName(), which name will be printed when printName() is called on C? In Java only the interfaces can be inherited, but not the possibly conflicting implementation.

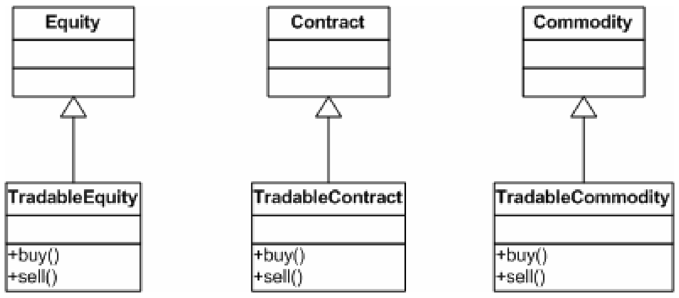

Figure 4: Back to the stock market. Imagine a terminal application that is originally designed for display purposes only. Now we want to extend the application to support active trading as well. Every entity that we want to make available for trading has to be extended with a layer of code that implements the required behaviour.

Mixins can solve this problem by allowing us to apply the same code to many classes that have the same interface. A mixin can be viewed as an abstract subclass that may be used to specialize behaviour of a variety of parent classes. It does this the same way as with normal inheritance, by adding new methods and data members or by overriding existing methods of its superclass thus allowing us to express large scale refinements.

Mixins can be implemented using parameterized inheritance. C++ is a language that provides parameterized inheritance by means of templates. The Tradable mixin that we where looking for in figure 4 would look like this:

```
template <class T>
class Tradable : public T
{
  public:
    void buy (long quantity);
    void sell (long quantity);
};
```

Applying this to the problem of the terminal application that needs to be extended with trading functionality will result in the following diagram.

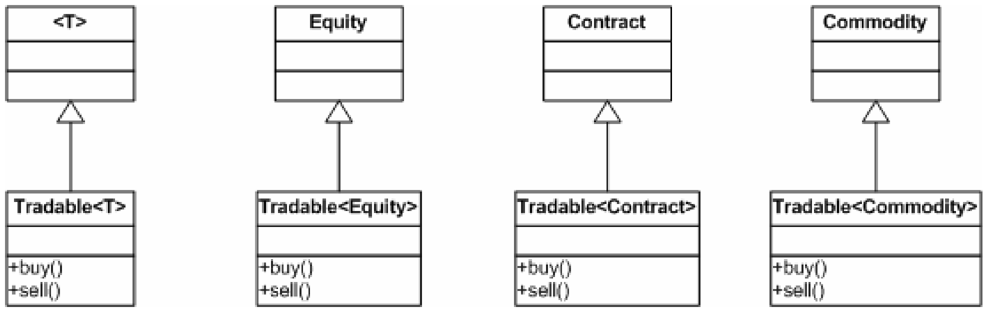

Figure 5: By using mixins the replication of code can be avoided.

A disadvantage of the use of C++ templates it that they are type insecure. Mixins generally require that their superclasses adhere to a fixed interface. In C++ there is no way to enforce this. Examples of other languages that support mixins are the Common Lisp Object System (CLOS) and Beta.

Mixin Layers

Mixin layers are mixins that encapsulate other mixins. The encapsulated mixin is called an inner mixin, the mixin that encapsulates the inner mixin is called the outer mixin. A key property of mixin layers is that the parameter of the outer mixin encapsulates all parameters of inner mixins. Mixins layers are an efficient way to implement collaborations. Collaborations can be composed by instantiating one mixin layer with another as its superclass. When composing an application with collaborations of course this innermost class cannot be a mixin, it must be a concrete instantiable class.

```
template <class T>
class Collaboration3 : public T
{
  public:
    class ObjectA : public T::ObjectA { … };
    class ObjectB : public T::ObjectB { … };
    class ObjectC : public T::ObjectC { … };
    …
};
```

The composition in figure 3 can be achieved with the following code:

```
Collaboration3 < Collaboration2 < Collaboration1 < ConcreteClass > > > composition;
```

A parallel can be drawn between collaboration-based design (using mixin layers and the ISO OSI model. In principle both systems describe a layered design where only the interfaces between layers need to be standardized. Layers can be added, removed or swapped as required as long as they adhere to the specified interface. Collaboration-based design provides the same encapsulation/decapsulation that has proven successful in development of data communication networks.

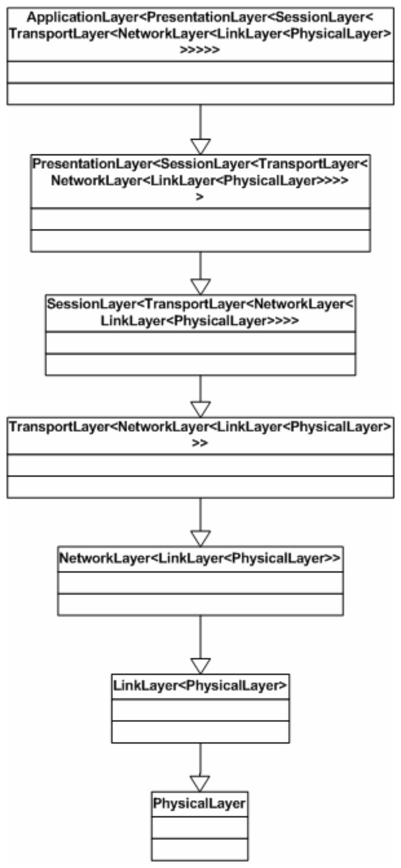
ApplicationLayer<PresentationLayer<SessionLayer<
TransportLayer<NetworkLayer<LinkLayer<PhysicalLayer>
>>>>>
PresentationLayer<SessionLayer<TransportLayer<
NetworkLayer<LinkLayer<PhysicalLayer>>>>
>
SessionLayer<TransportLayer<NetworkLayer<
LinkLayer<PhysicalLayer>>>>
TransportLayer<NetworkLayer<LinkLayer<PhysicalLayer>
>>
NetworkLayer<LinkLayer<PhysicalLayer>>
LinkLayer<PhysicalLayer>
PhysicalLayer

Figure 6: The OSI model implemented as collaborations.

Java Language Support for Mixin Classes and Layers

One of the most prominent players in the field of object-oriented programming languages is Java. Since its initial release in 1995 Java has conquered the hearts of many programmers and is now used on a world wide scale for large commercial software projects. However, the collaboration-based design paradigm is cannot be easily applied to Java. Java supports only single implementation inheritance with multiple interface inheritance. The core language does not include mixins or parameterized inheritance that allows for an implementation of collaborations using mixin layers.

Extending Java with mixins has been the topic of active research over the last few years. Numerous extensions have been developed both binary and in the form of source-to-source compilers. Source-to-source compilers have the benefit from the fact that they do not depend on a specific version of Java. While Jam, a binary extension, depends on JDK 1.1, the Java Layers source-to-source compiler provides a more flexible way of mixin support that is also compatible with future version of the JDK. JL allows the implementation of mixins by supporting C++ style templates. The main difference with C++ templates is that JL provides type safety by enforcing the superclass to adhere to specified interfaces.

As an example we use a simple scenario with one interface called TransportIfc, a class TCP that implements the TransportIfc and a session layer that is placed over the TCP class again implementing the TransportIfc.

```
public interface TransportIfc
{
     int send(byte[] outBuf);
     int recv(byte[] inBuf);
     void disconnect();
}
```

The interface for our example:

The TCP class implementing the TransportIfc interface:

```
class TCP implements TransportIfc
{
     public int send(byte[] outBuf) { … };
     public int recv(byte[] inBuf) { … };
     public void disconnect() { … };
}
```

The session layer enforcing a superclass that implements at least the TransportIfc interface:

```
class Session<T implements TransportIfc> extends T
 implements TransportIfc
{
      public int send(byte[] outBuf) { … ; super.send(outBuf); };
      public int recv(byte[] inBuf) { super.recv(inBuf); … };
      public void disconnect() { … ; super.disconnect(); };
}
```

Instantiation of a TCP session:

```
Session<TCP> tcpSession;
```

On compilation the JL compiler first creates an intermediate Session class:

```
class Session_A77B2FE extends TCP implements
TransportIfc
{
      public int send(byte[] outBuf) { … };
      public int recv(byte[] inBuf) { … };
      public void disconnect() { … };
}
```

The instantiation is rewritten as follows:

```
Session_A77B2FE tcpSession;
```

After the intermediate Session class is created the standard Java compiler can take over and generate the appropriate class files.

Conclusion

Collaboration-based design and layered programming can provide the unit of modularity that is useful for building large applications and software product lines. Applications can be built as layers stacked on top of each other, every layer providing a single feature or fulfilling a single responsibility. Maintenance and modification of the applications now boils down to simply swapping or adding layers. Layered design has proven to be very successful in the field of data communications and network interconnects, there is no reason why it cannot be applied equally well to software engineering.

The mixin layer concept is quite general and is not tied to any particular language idiom. For languages that do not provide native support for the mixin construct, often very elegant solutions can be applied to extend these languages. By introducing a pre-processor that is executed before the actual compilation phase the Java Layers source-to-source compiler adds the concept of parameterized inheritance to Java. Because the mechanism for extension exists of only a pre-processor it is likely that Java Layers can be used for future Java versions as well, as no major changes to the syntax of Java are to be expected.

References

[1] Matthew Flatt, Shriram Krishnamurthi, Matthias Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. TR 97-293: Corrected Version, June 8, 1999.

[2] Yannis Smaragdakis and Don Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs.

[3] Davide Ancona, Giovanni Lagorio and Elena Zucca. Jam - Theory and Practice of a Java Extension with Mixins.

[4] Yannis Smaragdakis. Implementing Layered Designs with Mixin Layers.

[5] Gilad Bracha and William Cook. Mixin-based Inheritance.

[6] Don Batory and Yannis Smaragdakis. Building Product-Lines with Mixin-Layers.

[7] Richard Cardone and Calvin Lin. Comparing Frameworks and Layered Refinement. ICSE 2001.

[8] Yannis Smaragdakis and Don Batory. Mixin-Based Programming in C++.

[9] Nguyen Truong Thang and Takuya Katayama. Collaboration-based Evolvable Software Implementations: Java and HyperlJ vs. C++-templates composition.

[10] Richard Cardone, Don Batory and Calvin Lin. Java Layers: Extending Java to Support Component-Based Programming. June 28, 2000.

[11] Rich Cardone. Java Layers: Language Support for Layered Refinement. January 16, 2001. PowerPoint presentation.

[12] Anton Eliëns, Principles of Object-Oriented Software Development. Second Edition, 2000.