

Composition and Multiple-Inheritance in OO Design (Where in the Madness is the Method?)

William Harrison (harrison@watson.ibm.com)
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Category: A/B

Abstract

One of the difficult issues in mapping UML to Java is the question of how to deal with multiple-inheritance used in a high-level design. In the absence of composition, Tengger¹ (a UML-to-Java generator) makes an admittedly oversimplified design decision in answer. Work on the use of Tengger in design-for-composition situations has suggested a much better approach that gives insight into the deeper question of the relationship of design to reuse specifications in the implementation.

Multiple Inheritance

Inheritance² is a technique for specifying the reuse of implementations. It has two entirely different reasons being used: differential development and composition. Differential development is the concept behind “this one is just like that one except ...”. It is reflected in the programming language construct usually called *single inheritance*. Composition is the concept behind “this one is both a one of these and a one of those”. It is reflected in the programming language construct usually called *multiple inheritance*.

Programming Language Problems Caused by Multiple Inheritance

The earliest OO Languages, Simula-67 and Smalltalk, had only single inheritance. Multiple-inheritance appeared³ in Flavors and several related LISP-based OO languages, and is described in the following terms: “When flavors are mixed together, Flavors organizes and manages the interactions between them. This *multiple inheritance* is a key aspect ...”. The usefulness of the mixing of flavors was so great that multiple-inheritance was independently added to Smalltalk-80

¹ William Harrison, Charles Barton, Mukund Raghavachari, Mapping UML Designs to Java, Proceedings of 2000 Conference on Object-Oriented Programming Systems, Languages, and Applications, Minneapolis 2000

² The term “inheritance” is often applied both to implementations (classes) and to declarations (interfaces), although the effective meanings are so different that some programming languages even permit “multiple inheritance” for interfaces while prohibiting it for implementations. Preferring to use other terms, e.g. “inclusion”, for the declarations, I use “inheritance” to refer specifically to implementations.

³ The earliest reference I can find to “multiple-inheritance” is the paper by Alan H. Borning and Daniel H.H. Ingalls, “Multiple Inheritance in Smalltalk-80,” Proceedings at the National Conference on AI, Pittsburgh, PA, 1982, but, although Flavors was later described in those terms in David A. Moon, “Object-Oriented Programming with Flavors”, OOPSLA’86 Conference Proceedings, it was developed around 1978.

and to C++ as an extension after their first appearances. But when added to a programming language, multiple-inheritance was treated as a straightforward programming construct about which simple and universal rules for the composition could be asserted. In the end, the Smalltalk-80 extensions were little used, and C++ was left with virtual inheritance, non-virtual inheritance, complex rules about constructor sequencing, and a non-obvious dominance rule for selecting implementations.

The developers of Java avoided this quagmire by excluding multiple inheritance. In doing so, Java has focused on differential development and effectively pointed to the need for extra-linguistic support for composition⁴. So it is fortunate for the software development community that Java composition tools like Hyper/J⁵ and AspectJ⁶ are available.

Generalization and Multiple Inheritance in High-Level Design

While programming languages have taken the term “multiple-inheritance” to be one that directs code-reuse, design languages like UML use it to stand for a different concept: generalization. In a sense, generalization is the “up-side-down” concept from inheritance. Rather than saying: “car and truck inherit from vehicle”, one says: “vehicle is a concept that includes both car and truck”. This might appear to be much the same as interface inclusion, except that the inclusion is specified in connection with the super-interface, rather than by the sub-interface.⁷

Tengger⁸ is a tool for generating stylized JavaTM code from the merge-by-name composition of collections of high-level UML designs. In presenting the rationale for Tengger’s mapping choices⁹, we identified severe problems that arise from mapping UML’s multiple-generalizations onto Java’s single-inheritance. As we explored Tengger’s use in larger-scale (composition-based) software development, we realized that a synergy existed that would allow us to define a feature-development style in which Tengger’s UML-Java mapping was unnecessary and in which Hyper/J’s composition took over that role in a way that much more cleanly separated the design from the implementation.

⁴ Unfortunately, the JVM is too weak to permit code reuse except as reflected in the language’s single-inheritance mechanism. This makes composition support less efficient and more difficult and obscure than it should be and may hasten movement from the Java engine to Microsoft’s C# engine.

⁵ <http://www.alphaworks.ibm.com/tech/hyperj>

⁶ <http://aspectj.org>

⁷ Linguistically, it is actually specified in a neutral corner – by the creation of an arrow between them. But the sense is often that a generalization arises to cover several previously unrelated entities. For example if an association might lead to either of two types of entity, a generalization must be created because the association can really target only one type. The definer of the generalization then “lifts” into the generalization the state or operations from the individual types that are needed by software that traverses the association.

⁸ <http://www.alphaworks.ibm.com/tech/tengger>

⁹ William Harrison, Charles Barton, Mukund Raghavachari, op. cit.

The Design Tension – Where to Find the Method in the Madness

The Surface Problem – Arborization is Faulty

To illustrate the tension, consider a small example that arises in the design and implementation of SAGE¹⁰, a message-mapping tool. Figures 1 and 2 are culled from the UML design of two of SAGE’s features. They both involve an entity called “AbstractType” in their designs. Figure 1 comes from the “Dictionary” feature. In this feature an entity also called “Dictionary” (not shown) has an association connecting it to either AbstractType or NamedValue. To characterize the target of that association, the entity IdentifiableEntity was defined as a generalization of AbstractType.

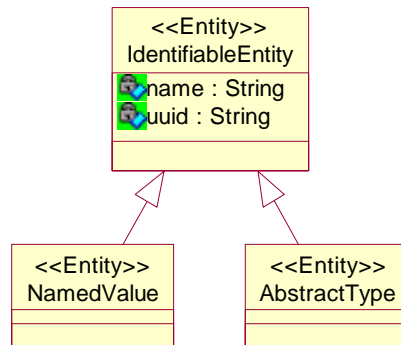


Figure 1 – Design from Dictionary Feature

Figure 2 is from the “UI” feature. In it, the AbstractElement and AbstractType entities must be associated with UITree Nodes, leading to the need for the generalization called “DisplayPart”.

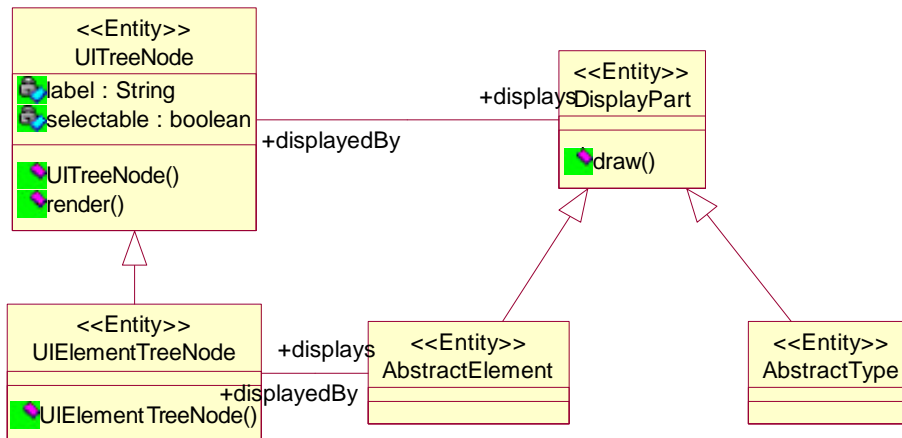


Figure 2 – Design from UI Feature

These two features are independently designed, but when composed, the effective design includes the multiple-generalization structure shown in Figure 3. Used without composition, Tengger

¹⁰ <http://www.research.ibm.com/messagecentral/>

produced the class arborization¹¹ shown in Figure 4. As we pointed out, this arborization suffers from the potential of false inheritances: an “equals()” method implemented in IdentifiableEntity would be inherited into AbstractElement. More severe cases happen with more complex models, as the number of false inheritances rises.

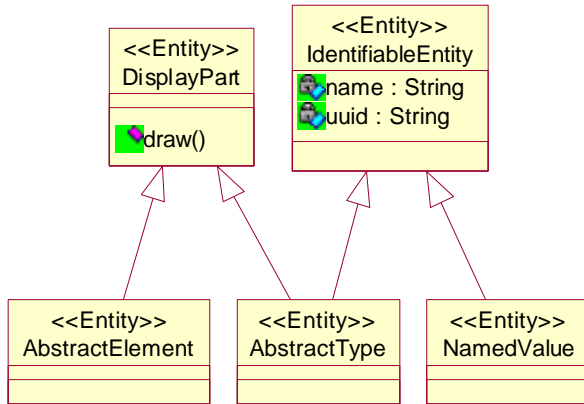


Figure 3 – Multiple-Generalization

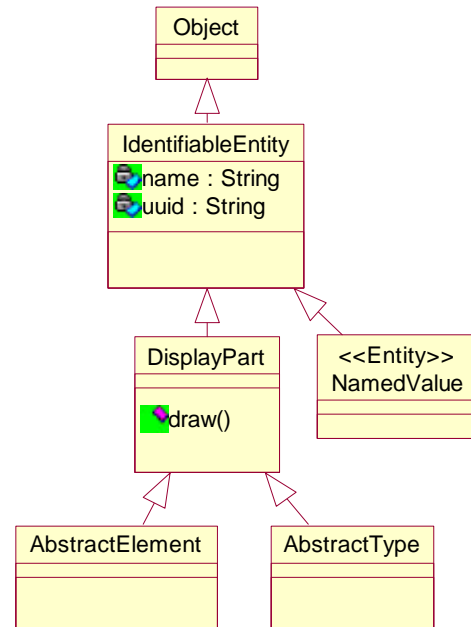


Figure 4 – Single-Inheritance Arborization

As we also observed, the false inheritances could be fixed by method renaming. If Tengger knows that IdentifiableEntity implements equals(), then it could change the calls used in AbstractElement to calls more directly targeted to bypass IdentifiableEntity. But the information about which classes actually *implement* a method is present only in the implementations, and is not even appropriate information to put in a high-level design.

The Deeper Problem – Design/Implementation Tension

In fact, working with the implementations, Hyper/J performs the much more sophisticated method renaming and superclass redirection needed to fix the false inheritances. But recognizing that the implementation notes did not belong in the design and that they *could* not appear in the naïve Java code led us to observe that there is a somewhat deeper problem here. In the design, multiple generalization is quite appropriate and meaningful and, in mappings like those used by Tengger that create an interface corresponding to each entity, they imply a multiple-inclusion structure among the interfaces. Most languages with interfaces allow multiple-inclusion for interfaces, making for a natural design/implementation map. But a mapping that uses the design to determine *single-inheritance* relationships among implementation classes stands on very shaky ground. Inheritance is an implementation reuse mechanism, not a high-level design structure mechanism.

¹¹ The boxes in this Figure 4 represent Java classes, not high-level entities.

What is taking place is an attempt to infer the implementation from a design structure that contains insufficient information, and that the information does not lie in the base-level code either. Recognizing this crystallizes the realization that we need to use some other development model artifact for the specification. With Tengger's feature-oriented development model, or more generally with Hyper/J's hyperslices, another artifact is at hand that could be employed.

The Resolution – Inheritance of Implementations is Specified in the Implementations in a Way that Depends on Where the Implementation Sits in the Design (whew!)

Tengger's Feature-Oriented Development Model

Tengger supports two development models, one for stand-alone use and one for compositional use with Hyper/J. The cardinal artifact in the development model is called a "feature". Features contain a public design, feature code, and a protected design, and may contain other features. A feature not contained in another feature is called a "base".

Designs are created in UML and stored as collections of XMI files. Feature code is created in Java, using the Tengger-specified conventions for interacting with the Java representation of the design. For an entity E in feature F , these conventions say:

1. the class ($EInst$) containing the feature code is to be **abstract**
2. the abstract class is in package F
3. the abstract class is declared to implement the interface E .

The feature code gets access to E and other interfaces by importing the Java representation of the feature's composite design. The composite design of a feature is the merge-by-name composition of its public design, its protected design, the public designs of its sub-features, and the composite design of the feature that contains it.

The Java representation of a feature's composite design is generated using Tengger, and then compiled using an available Java compiler. The feature code is then compiled using the result. When the code for all features has been compiled successfully, Hyper/J is used to compose all an entity's class definitions from all the features, using a merge-by-name rule.

Where in the Madness Is the Method?

With this development model, we can more clearly revisit the UI and Dictionary features used in Figures 1 and 2. They are sub-features of a base called SAGE. This means that we need only consider the composite design for SAGE, which includes the multiple-generalization shown in Figure 3. But unlike the arborized Java class diagram in Figure 4, we have other alternatives open. The Tengger-specified conventions for feature code place no constraint on the choice of the superclass named in the "extends" clause in a feature's Java classes. According to the generation, compilation, and composition process described above, the method(s) actually present in a class are the merge-by-name of the methods present in all the features. Figure 5 shows class diagrams for the feature code in the Dictionary and UI features. The result of the merge is that `AbstractElementInst` and `AbstractTypeInst`. Classes contain get/set method implementations that they inherited from `IdentifiableEntityInst` in the Dictionary feature and the draw method implementations they inherited from `DisplayPart` in the UI feature.

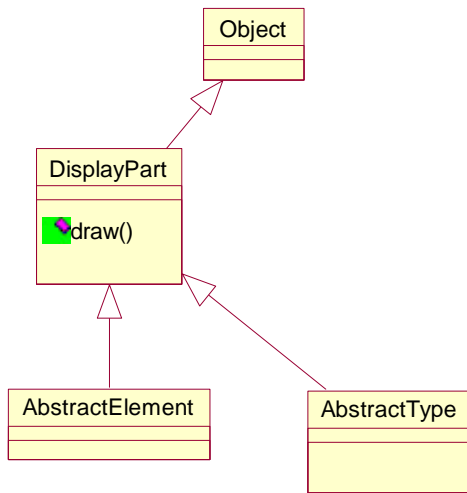


Figure 5a – Dictionary Feature Class Hierarchy

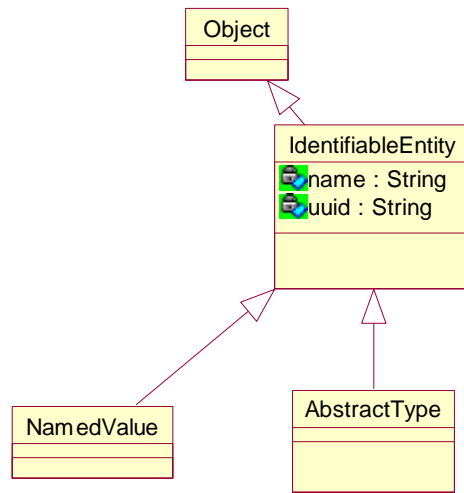


Figure 5b – UI Feature Class Hierarchy

Now reconsider the false-inheritance problem discussed above. With respect to Figure 4, we noted an example whereby the implementation of the “equals()” method in IdentifiableEntityInst would be falsely inherited into AbstractElement. We further noted that there was no place in either the design or in the implementation to note the fact that AbstractElement should get its equals() from Object and not from IdentifiableEntity. Note that in this composition-based method, AbstractElementInst is nowhere defined as a subclass of IdentifiableEntityInst. And since they are not composed with it by name, no “false inheritance” into AbstractElementInst takes place. The note that AbstractElement gets its implementation of equals() from Object lies in the fact that it does not extend IdentifiableEntity, even though AbstractType *does* extend IdentifiableEntity. In addition, using composition semantics rather than inheritance, the selection of which implementation of equals() to use in AbstractType, whether both by merge, one by override, or other explicit control, has been brought back to the use of composition rules. This takes multiple-inheritance back through the whole circle of its trials to its true nature and to the way it was first introduced by Flavors – as a composition function governed by combination rules.

Inheritance vs. Composition

One of the observations engendered by this discussion of multiple inheritance and composition is that inheritance is fundamentally a code-reuse facility. As such, it has much to do with how a component is implemented, but little to do with governing the results of composition. If class A is a subclass of class B in one feature, but class A is independent of class B in another, only the methods inherited from A in the first feature are present in the result. Methods that A may contain in the second feature do not suddenly become inherited into B just because the first feature used inheritance from A to B as part of its implementation strategy. In fact, trying to include inheritance in the composition easily results in false inheritance and even cycles, especially when implemented against a single-inheritance language like Java. But with the exactly opposite effect, we believe that generalization *is* appropriate for inclusion in composition. The difference has to do with the difference in level of exposure between generalizations (interfaces) and inheritances (implementations). Some Java programmers are surprised by the fact that inheritance is an implementation characteristic. This surprise may result from failure to clearly articulate principles when we teach OO technology, and we hope that the inclusion of compositional programming concepts in software education will help remedy this situation.