Implementing Relationship Constraints in OO Programming Languages

Andrew Eales **Rhys Owen**

Wellington Institute of Technology
Petone, Wellington, NZ
andrew.eales@weltec.ac.nz

Rigorously defined object relationships are crucial to the successful expression of a conceptual design idea in a programming language. Conceptual relationships expressed by various forms of aggregation and association form the cornerstones of object-oriented systems. These relationships must be unambiguously articulated by the design notation and clearly implemented in a programming language. Object-oriented programming languages have neglected to provide support for implementing aggregate and associative relationships. Additionally, UML contains ambiguities that can only be resolved using non-diagrammatic extensions such as the Object Constraint Language (OCL). This paper examines the expressive deficiencies in the diagrammatic representations within UML when confronted by the semantic implications of many different types of aggregate and associative relationships. The authors propose extensions to object-oriented programming languages to accurately reflect the semantics of design relationships expressed in UML. A novel Java implementation of aggregation and association relationships that is used within a Java visual development environment is discussed. These language extensions ensure that programming language implementations of object relationships are checked at compile-time and enforced at run-time.

Keywords

object-oriented, programming languages, Java, software design, teaching programming, BlueJ

1. INTRODUCTION

Rigorously defined object relationships are essential for the creation of robust software and formal verification of object-oriented systems. Conceptual relationships expressed by various kinds of aggregation and association form the cornerstones of object-oriented systems. The semantics of these object relationships must be unambiguously articulated by the design notation and clearly implemented in a programming language. Unfortunately, object-oriented programming languages have neglected to provide support for the implementation of aggregate and associative relationships. This situation is aggravated by UML, which does not clearly and unambiguously define the semantics of different types of object relationships. This paper examines different types of object relationships and proposes a programming language construct to correctly implement whole-part relationships. The authors discuss the implementation of constraints to enforce object relationship semantics within the BlueJ Java development environment (Barnes & Kolling, 2003) by making use of the BlueJ extension mechanism. Programming students working within BlueJ are given the illusion that the compiler and Java virtual machine are enforcing the meaning of object relationships.

2. OBJECT-ORIENTED RELATIONSHIP SEMANTICS

Whole-part and associative relationships in UML do not support the many different types of relationships covered by relationship theory (Winston, Chaffin & Hermann, 1987). Relationships that are supported by UML are not clearly defined in the UML specification (UML v1.5, 2003). A model can be viewed as a formal specification for the software implementation. Development of this approach within UML has lead to the OCL (Object Constraint Language), which is a formal language "that allows additional semantics to be added to UML models which, with the remaining UML elements, can either not be expressed at all or only insufficiently" (Oestereich, 2002). Using OCL, we can express precisely what we mean by a particular relationship, and also indicate which classes are responsible for maintaining the constraints.

```
context Car inv Wheels4: self.wheels->size() = 4
context Car inv SafeWheels: self.wheels->forAll(oclIsKindOf (Wheel))
context Wheel inv Car1: self.theCar->notEmpty()
```

Figure 1. A composite relationship in UML with OCL annotations.

2.1 Whole-part relationships

A wide variety of whole-part relationships exist. Odell (Odell, 1994) discusses the subtle semantic differences of these relationships by applying the work of Winston (Winston, 1987) to object orientation. Aggregation, denoted by a white diamond and composition, denoted by a black diamond are often confused in the literature. Barbier (Barbier & Henderson-Sellers, 1999) and Saksena (Saksena, France & Larrondo-Petrie, 1998) discussed the interpretation of aggregate and composite relationships in UML. The most common whole-part relationship consists of objects that are components with well-defined relationships to each other and the whole. Odell refers to this aggregate relationship as component-integral object composition, which corresponds to composite aggregation in UML (denoted by a black diamond). Programming languages only support implementations using references or instances that are topologically included within a class definition. Unfortunately, topological inclusion and attachment do not guarantee composition. Something being "inside" something does not necessarily imply a composite relationship; a person inside a car is not a part of the car. Neither does attachment of one object to another guarantee composition. For example, toes are attached to feet and are also parts of feet; however, while rings are attached to fingers, they are logically not parts of fingers. The UML 1.5 specification provides the following definition of composition:

> *"Composite aggregation is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. The multiplicity of the aggregate end may not exceed one (it is unshared)." (UML v1.5, 1998)*

Composition (composite aggregation) is thus equivalent to strong aggregation, implying a coincident lifetime between the whole and its parts. This definition does not prohibit a part existing without a corresponding whole (or vice-versa), nor does it prohibit the parts being used outside of the whole. For example, an aircraft has the sole responsibility for the disposition of its wings. The wings belong to the aircraft and should not be manipulated from outside of the aircraft. However, a wing can be created outside of an aircraft and attached to the aircraft after creating the aircraft, or, a wing can be detached from an aircraft and destroyed after the aircraft is destroyed. The UML specification is ambiguous and does not provide answers to the many questions that arise when attempting to rigorously define a relationship. A more rigorous semantics of composition suggested by Henderson-Sellers (Henderson-Sellers & Barbier, 1999) requires a minimum multiplicity of parts as well as unshareability amongst parts. A UML design can clarify a relationship using OCL statements such as shown in Figure 1.

The cardinality of the part must be exactly four, with Car objects responsible for managing the cardinality of Wheel instances via the cardinality of the set of wheels stored within class Car. Car objects are also responsible for ensuring that the set of wheels only contains objects of type Wheel. The set implies that the programming language implementation stores Wheel instances as a collection, and not as four separate references. Additionally, each Wheel instance must be attached to a Car by means of the reference in class Wheel.

2.2 Associative Relationships

Associative relationships are relationships that cannot be defined by a composite relationship if we accept the constraints of coincident lifetime and unshareability. Association clearly differs from composition. Unfortunately, the term "composition" is used indiscriminately within the object-oriented community to refer to any object relationship that is not inheritance. This confusion of different relationship semantics arises from programming language implementations of associative and composite relationships that utilize topological inclusion as discussed in section 2.1. Aggregate relationships encourage

interaction with the whole while associative relationships place objects within a co-operative relationship. The relationships differ in that associations need not have a minimum or fixed cardinality, need not implement unshareability unless required to do so, and do not require coincident lifetimes. Consider the relationships where a person may own an arbitrary (possibly zero) number of items as a shared owner, while a theatre patron has a fixed one-to-one relationship where the relationship defines the meaning of "theatre patron".

3. VERIFYING OBJECT-ORIENTED RELATIONSHIPS

Direct programming language support for object relationships would enable fundamental relationship constraints to be enforced by the compiler and run-time system. If whole-part relationships are defined within the programming language syntax, the common manipulation of parts by the whole becomes possible using a programming language construct. As an example, consider the animation of a car implemented using programming language extensions that support whole-part relationships:

```
class Wheel partOf Car
{    void Rotate (float factor)
      {   rotation *= factor;
      }
      float rotation;
}
class Car hasPart Wheel (4)
{   Car ()
    {  for (int i=0; i<4; i++)
            wheels.add (new Wheel
(this) );
    }
    void Accelerate (float factor)
    {    forAllParts
         {    Rotate (factor);
         }
    }
     Container carWheels;
}
```

Listing 1. Proposed programming language extensions.

The compiler can check that both ends of the relationship are correctly declared and that the cardinality of the part is observed when creating the whole. The parts can also be correctly manipulated via the whole using the forAllParts statement in listing one. These extensions cannot be directly added to the Java language but can be indirectly implemented using facilities provided by the BlueJ extension API.

3.1 Implementing Relationships in BlueJ

BlueJ (BlueJ, 2004) is a visual development environment for Java that supports rudimentary class diagrams that indicate class relationships. This environment is widely used to teach novice Java programmers. All relationships that are not inheritance relationships are simply indicated as referential relationships by a connecting dotted line between classes. The authors contend that BlueJ does not go far enough in emphasizing high-level object relationships within an object-oriented design and should at a minimum strive to differentiate between composite aggregate and associative relationships.

The BlueJ extension mechanism allows functionality to be added to the BlueJ IDE using Java code and the provided extension API. Unfortunately, the extension mechanism does not provide access to the class diagrams, preventing our extension from adding UML relationship diagrams to the existing class diagrams. As an extension can access the source code of a BlueJ project, our extension can parse the source code at compile-time, perform static checks, and then generate Java code that implements run-time checks on object instances.

3.1.1 Implementing Static Constraints

Relationship semantics can be added to Java code as comments following class declarations in the source code:

```
class Car    // hasPart Wheel(4)
             …
class Wheel   // partOf Car
             …
class Driver  // associates with Car
```

Static analysis of source code to ensure consistent composite relationships and associations is trivial to implement. At compile-time our extension parses the source code, examining the specified relationships following class declarations. Each part must have a matching whole and vice-versa, and the part cardinality must be specified. A similar procedure is followed for associations where the relationship can be unidirectional or bi-directional. When an infringement occurs, a message box displays the error giving the student the illusion that the compiler has flagged the error. Ensuring that semantic declara-

tions are not violated at run-time is much more difficult to achieve.

3.1.2 Implementing Dynamic Constraints

Ensuring that constraints are not violated during program execution requires three different validations. Minimum and maximum relationship cardinalities must be ensured, while composite relationships require enforcement of coincident lifetimes and unshareability. Our current model makes use of existing object wrappers provided by the BlueJ runtime system. This approach is extremely difficult to implement, as we must attempt to build a graph that tracks all object instances and references between objects. A better approach is to attempt to integrate executing Java code that expresses constraints with object instances created on the BlueJ object workbench. An appropriate methodology to express constraints within an object-oriented programming language is the Design by Contract (DBC) methodology proposed by Meyer (Meyer, 1988) for the Eiffel programming language. Design by Contract views the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations as pre-conditions, post-conditions and class invariants expressed as assertions. Java introduced support for assertions in Java 1.4 (Sun Microsystems, 2002). Another way of expressing constraints in Java is to use a third-party DBC tool specifically designed for Java. There are a number of candidates, (iContract, Jass, jContractor and others) but a significant one for the future, in our opinion is JML (Java Modeling Language) described by Leavens (Leavens & Cheon, 2004). JML is tightly coupled to the Java compiler, providing the best integration with the Java code used in a BlueJ extension.

4. CONCLUSIONS

Clarification of the semantics of relationships expressed in UML would benefit designers and allow extensions expressing UML constructs to be added to programming languages. Attempts should be made to clarify UML specifications and not rely on OCL annotations to provide semantic clarity. By using the BlueJ extension mechanisms in conjunction with runtime Java tools that support design by contract, programming students are made aware of the semantic differences between composite and associative relationships. An extension to the BlueJ environment simulates programming language extensions using embedded comments. Classes within BlueJ are required to declare their relationships to other classes at compile time, and to enforce these relationships during program execution. Deficiencies in the current UML specification have led us to restrict our focus to composite aggregate and simple associative relationships. The authors believe that if the BlueJ approach to programming, which emphasizes high-level object relationships is adopted, a clear distinction between different types of relationships is desirable. We continue to examine different strategies for the efficient checking of constraints during program execution.

REFERENCES

Barnes, D.J. and Kölling, M. (2003) ***Objects First with Java - A Practical Introduction using BlueJ.*** Prentice Hall - Pearson Education.

Barbier, F. and Henderson-Sellers, B. (1999) *Object Metamodelling of the Whole-Part Relationship*. In (Mingins,C., Meyer,B. Eds.) Proceedings TOOLS 32, IEEE Computer Society Press, Los Alamitos, California.

BlueJ (2004) <http://www.bluej.org>

Henderson-Sellers, B. and F. Barbier, F. (1999) *Black and White Diamonds* Proceedings of the 2nd IEEE conference on UML, pp.550-565.

Leavens, G. and Cheon, Y. (2004) *Design by Contract with JML*. <http://www.cs.iastate.edu/~leavens/JML/index.shtml>

Meyer, B. (1987) *Object-Oriented Software Construction,* 2nd edn. Prentice-Hall.

Odell, J.J. (1994) *Six Different Kinds of Composition.* Journal Of Object-Oriented Programming, 5(8).

Oestereich, E. (2003) *Developing Software with UML*, 2nd edn. Addison-Wesley.

Saksena, M., France, R.B. and Larrondo-Petrie, M.M. (1998) *A Characterization of Aggregation*, p.363. Proceedings of the 5th International Conference on OO Information Systems.

Sun Microsystems (2002) *Programming with Assertions*. <http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>

UML Specification v1.5. (2003) 3-48, p.481. <http://www.omg.org>

Winston, M.E., Chaffin, R. and Hermann, D. (1987) *A Taxonomy of Whole-Part Relations*. Cognitive Science 11, pp.417-444.