Object Inheritance: Definition, Uses, and Motivations

Mike Abney
mike@streamlinedmodeling.com

Introduction

One of the core concepts of object-oriented programming is that of inheritance. In most languages that means class-based inheritance: the inheritance of structure and implementation. Conceptually, however, inheritance can also be applied to objects themselves. Such an object-based inheritance would allow for the inheritance of state: a child object asked for its name provides the same answer as its parent.

As a concept, object inheritance has been around for quite some time. The Self language (see http://research.sun.com/self/ and http://www.cetus-links.org/oo_self.html) was – and is – based in part upon the concept of object inheritance. In Self, object inheritance was used to make prototype-based programming easier. New objects could be built quickly by inheriting the state and abilities of existing objects. Any object could be made the parent of another. A message sent to an object was answered by looking for a "slot" in the child that corresponded to the particular message sent. Inheritance was implemented through special "parent slots." If none of the regular slots matched the message, the message would be passed on to the parents until a match was found.

While object inheritance as a programming language concept is interesting, the concept has also shown itself to be useful in business modeling.

Uses of Object Inheritance

Object inheritance can be applied to business problems when a single real-world person, thing, or event needs to be represented by more than one object. This occurs in three of the twelve collaboration patterns presented in *Streamlined Object Modeling: Patterns, Rules, and Implementation*. (Nicola, Mayfield, and Abney[1]).

In the actor – role pattern, a single real-world person is represented by an actor and some number of roles that the person can play in the system. In this case it is desirable for the roles to be able to answer certain key questions that would typically be answered by their actor. For example employees and customers (roles) should both be able to answer the question, "What is your name?" The name they would provide is actually a property of their associated person (actor).

In the item – specific item pattern, a single real-world thing is represented by a detailed specific item and an associated item description shared with other specific items. Each specific item that is associated with a given item description should share the values of that description. For example, if a movie (item) is available in both VHS and DVD formats (specific items), the VHS version should answer the same values for the title, director, actors, etc. as the DVD, and the values they provide should come from the associated movie.

In the composite transaction – line item pattern, many entities are taking part in a single event. The composite transaction describes the properties of the overall event while the line items provide detail about the connection between the event and the involved entities. In a sales transaction with more than one specific item involved, the composite transaction – line item pattern is used. The line items record any extra information, at a minimum the quantity purchased, about the association of the specific item to the transaction. If a specific item asks its line item when the event happened, the line item should answer with the date and time provided by the composite transaction.

The Most Useful Aspects of Object Inheritance

Since we know that most languages do not directly support object inheritance, we limit ourselves to those ideas that are the most useful. It turns out that there are two concepts related to object inheritance that are really useful. First, a child object should be able to act as a partial stand-in for its parent. Second, a child object should be able to answer messages in the same way and with the same values as its parent.

While the second concept seems more obviously useful, the motivation behind making the child object a stand-in for its parent is a little more obscure. The objects that are related through object inheritance represent a single real-world entity. The actor and its roles represent a single person; the specific item and its item description represent a single thing; the composite transaction and its line items represent a single real-world event. When the other objects in the system are working with one of the child objects what they are really doing is working with the combined entity in a particular context. Therefore, the other objects will expect the children to be able to provide many of the same capabilities as their parents.

In addition to useful features there are useful restrictions when applying object inheritance to a business model. Since a child object represents its parent only in a specific context, it does not make sense for the child to be able to change its parent. Any changes made could be invalid in other contexts. Additionally, the parent's collaborations with other objects should only be visible to its children if they make sense for all of the children of a given type. For example, a line item in a sales transaction would not inherit the relationship between the sale and a subsequent shipment transaction because it is possible that not all line items are included in a given shipment.

Implementing Object Inheritance

Now that we know what we want and what we wish to avoid, the question remains: How do we implement object inheritance in a language that provides no direct facility for it? A good language to use for an example implementation is Java. Java has no explicit facility for object inheritance. It is class-based, and it has many of the same features and limitations as several other object-oriented languages.

Creating a Profile

One feature (or limitation depending upon your point of view) common in many languages is compile-time type checking. This feature allows the compiler to catch many common programming mistakes. In the case of object inheritance, however, it makes the second of our desirable aspects a little more difficult to implement. The child cannot stand-in for its parent unless they can be viewed as the same type.

Regular class inheritance would accomplish this but is not what we really want. We do not want to inherit the implementation. In fact, inheriting the implementation would make it likely that some method we intended to override would be forgotten leading to unexpected, and somewhat difficult to find, errors. Also, class inheritance is not selective; it is all or nothing. We need to be able to choose which properties and services will be shared between the parent and its children.

Fortunately, Java also has a facility for purely abstract inheritance of interfaces. Java interfaces define only the method signatures and leave implementation up to concrete classes. If we specify that the parent and its children implement the same interface, then other objects can refer to both parent and child through this interface without worrying about which specific type they are messaging.

We call the inheritable interface the “profile” of the parent. The profile interface includes all the services and data accessing methods that can be inherited. From the actor – role example, a person’s profile would include things like read-only accessors for the name, address, and any other globally useful contact and demographic information. It would not include services that change information or involve the actor’s roles.

```
public interface PersonProfile {
  String getName();
  Address getAddress();
  PhoneNumber getPhoneNumber();
}
```

Code Listing 1. The PersonProfile Interface

Making the Parent Responsible

Our other necessity is to have the child answer messages in the same way and with the same values as its parent. This can be accomplished in two basic ways: copying and forwarding.

Copying, in this situation, would mean that the values of the parent’s properties (and inheritable collaborations) would be copied into the child. This causes a normalization problem. The data is now kept in more than one place. Changing the values in the parent will mean that the child is out of sync.

Forwarding means that the child’s implementation of the profile interface will directly pass the request on to the parent. This implementation makes the parent solely responsible for access to its information. This is not only better normalization it is better encapsulation.

```
public class Employee implements PersonProfile {
  protected Person parent;
```

```
    // The rest of the Employee code
    // goes here.

    public String getName() {
      return this.parent.getName();
    }

    public Address getAddress() {
      return this.parent.getAddress();
    }

    public PhoneNumber getPhoneNumber() {
      return this.parent.getPhoneNumber();
    }
  }
```

Code Listing 2. Implementation of Message Forwarding

There is one problem with message forwarding. As you can see in Code Listing 2, it makes for very tedious code writing. Each method implementation is basically the same: “Hey parent, do this for me, okay?” As with many of the more tedious facets of programming, this cannot be helped until either more languages have a facility for it built in or tools are created to automatically generate the code.

Overriding Object Inheritance

So far, we have talked only about the need for the child to duplicate the responses of its parent. What about cases where the child does need to change its appearance?

For example, in a retail environment a general descriptive item object and a more detailed specific item object combine to represent a shirt. The item would provide information such as brand, style, possibly a textual description, and a price. The specific item would provide details such as the SKU number, the size, and color. Suppose, however, that certain SKUs are not selling well – no one wants the lime green, XXX-small shirts this year. The merchant would like to change the price for just those particular shirts.

One solution is to simply not object inherit the price. The price property would move from the item to the specific item. The problem with this is that now any time the merchant wants to change the price on all the shirts – end of season clearance – each SKU must be changed separately.

A better solution is to provide the child object with its own price attribute. While this price is empty (null), the child forwards the message on to its parent. If this value is set, however, the child’s value overrides the value of the parent. This same scheme works for collaborations as well.

```
public class SpecificItem implements ItemProfile {
  protected Item parent;
  protected Currency price;
  // The rest of the SpecificItem code
  // goes here.

  public Currency getPrice() {
    if (this.price == null) {
      return this.parent.getPrice();
    }
    else {
      return this.price;
    }
  }
}
```

Code Listing 3. Overriding a Property Value

Summary

Many of the benefits and pitfalls of class-based inheritance are widely known. Object-based inheritance has remained in relative obscurity. Although it has been long been a tool in prototype-based programming, it has less often been applied to business modeling. By examining the concept in the context of business modeling, we find that object inheritance is immensely useful in three collaboration patterns: actor – role, item – specific item, and composite transaction – line item. By identifying the particular aspects of object inheritance that are useful and by being careful to properly limit its scope, object inheritance can be implemented in a class-based language such as Java.

References

1. Nicola, Jill, Mark Mayfield, and Mike Abney. Streamlined Object Modeling. Upper Saddle River, N.J.: Prentice Hall PTR, 2002.