The Type Object Pattern

Bobby Woolf

Knowledge Systems Corp.
4001 Weston Pkwy, Cary, NC 27513-2303
919-677-1119 x541, bwoolf@ksccary.com

TYPE OBJECT — Object Structural

Intent

Decouple instances from their classes so that those classes can be implemented as instances of a class. Type Object allows new "classes" to be created dynamically at runtime because they are instances, yet it also enables the system to create instances of those class-like instances.

Also Known As

Power Type, Reified Type, Metaobject, Instance of an Instance

Motivation

Sometimes a class not only requires an unknown number of instances, but an unknown number of subclasses as well. Although an object system can create new instances on demand, it usually cannot create new classes without recompilation. Thus the subclasses must actually be implemented as instances of the class and the original instances are now instances of the class' instances.

Consider for example a system for tracking the videotapes in a video rental store's inventory. The system will obviously require a class called "Videotape." Each instance of Videotape will represent one of the videotapes in the store's inventory. However, since many of the videotapes are very similar, the Videotape instances will contain a lot of redundant information. For example, all copies of *Star Wars* are pretty similar: they have the same title, rental price, MPAA[1] rating, etc. This information is different for *The Terminator*, but all copies of *The Terminator* have the same data. Thus repeating this information through all copies of *Star Wars* or all copies of *The Terminator* is redundant.

One way to solve this problem is to create a subclass of Videotape for each movie. Thus two of the subclasses would be StarWarsTape and TerminatorTape. The class itself would keep the information for that movie. So the information common to all copies of *Star Wars* would be stored only once. It might be hardcoded on the instance side of StarWarsTape or stored in variables on the class side or in an object assigned to the class for this purpose. Now Videotape would be an abstract class; the system would not create instances of it. Instead, when the store bought a new videotape and started renting it, the system would create an instance of the class for that movie. So if the new videotape were for *The Terminator*, the system would create an instance of TerminatorTape.

This solution works, but not very well. One problem is that if the store stocks lots of different movies, Videotape could require a huge number of subclasses. Another problem is what would happen when the system is deployed and the store starts stocking a new movie—perhaps *Independence Day*. There is no IndependenceDayTape class in the system. If the developer did not predict this situation, he would have to modify the code to add a new IndependenceDayTape class, recompile the system, and redeploy it. If the developer did predict this situation, he could provide a special subclass of Videotape—such as UnknownTape—and the store would create an instance of it for all videotapes of the new movie. The problem with UnknownTape is that it has the same lack of flexibility that

[1] The Motion Picture Association of America, the industry group that rates movies in the United States as G, PG, R, etc.

Videotape had. Just as Videotape required subclasses, so will UnknownTape, so UnknownTape is not a very good solution.

Instead, since the number of types of videotapes is unknown, each type of videotape needs to be an instance of a class. However, each videotape needs to be an instance of a type of videotape. Class-based object languages give support for instances of classes, but they do not give support for instances of instances of classes. So to implement this solution in a typical class-based language, you need to implement two classes: one to represent a type of videotape (Movie) and one to represent a videotape (Videotape). Each instance of Videotape would have a pointer to its corresponding instance of Movie.

```
                         Videotape
     Movie    <------     movie
     title               isRented
  rentalPrice             renter
                                           ___Classes___
                                             Instances

     Movie    <------ Videotape
   Star Wars     |  John's Star Wars
                 |
     Movie      |--- Videotape
 The Terminator     Sue's Star Wars
```

This diagram illustrates how each instance of Videotape has a corresponding instance of Movie. It shows how properties defined by the type of videotape are separated from those which differ for each particular videotape. In this case, the movie's title and how much is costs to rent are separated from whether the tape is rented and who is currently renting it. It also shows how there is an instance of Movie to represent each type of videotape and an instance of Videotape to represent each video the store stocks. *Star Wars* and *The Terminator* are movies; videotapes are the copy of *Star Wars* that John is renting verses the one that Sue is renting. Finally, it shows how each Videotape knows what type it is because of its relationship to a particular instance of Movie.

If a new movie, such as *Independence Day*, were to be rented to Jack, the system would create a new Movie and a new Videotape that points to the Movie. The movie is Independence Day and the tape is the copy of *Independence Day* that Jack ends up renting.

Videotape, Movie, and the is-instance-of relationship between them (a Videotape is an instance of a Movie) is an example of the Type Object pattern. It is used to create instances of a set of classes when the number of classes is unknown. It allows an application to create new "classes" at runtime because the classes are really instances of a class. The application must then maintain the relationship between the real instances and their class-like instances.

The key to the Type Object pattern is two concrete classes, one whose instances represent the application's instances and another whose instances represent what would normally be the application's classes. Each application instance has a pointer to its corresponding class-instance.

Notice that the Type Object pattern also lets an object dynamically change its "class," the type object. It is much simpler for an object to change its pointer to a different type object than for the object to mutate to a new class. For example, let's say that a shipment to the video store is supposed to contain three copies of *The Terminator* and two copies of *Star Wars*, so those objects are entered into the system. When the shipment arrives, it really contains **two** copies of *The Terminator* and **three** copies of *Star Wars*. So one of the three new copies of *The Terminator* in the system needs to be changed to a copy of *Star Wars*. This can easily be done by changing the videotape's Movie pointer from The Terminator to Star Wars.

Dividing videotapes into Movie and Videotape makes it easy to support videodisks as well. A new Videodisk instance would point to its Movie instance just like a Videotape would. If the store carried three tapes and two disks of the same movie, three Videotapes and two Videodisks would all share the same Movie.

Also note that the Type Object pattern can be nested recursively. For example, many video stores have several categories of movies, such as New Release (high price), General Release (standard price), Classic (low price), and Children's (very low price). If the store wanted to raise the price on all New Release rentals from $3.00 to $3.50, it would have to iterate through all of the New Release movies and raise their rental price. It would be easier to store the rental price for a New Release in one place and have all of the New Release movies reference that one place. Thus the system needs a MovieCategory class that has four instances. The MovieCategory would store its rental price and each Movie would delegate to its corresponding MovieCategory to determine its price. Thus a MovieCategory is the Type Object for a Movie, and a Movie is the Type Object for a Videotape.

Applicability

Use the Type Object pattern when

- instances of a class need to be grouped together according to their common attributes and/or behavior.
- the class needs a subclass for each group to implement that group's common attributes/behavior.
- the class requires a large number of subclasses and/or the total variety of subclasses that may be required is unknown.
- you want to be able to create new groupings at runtime that may not have been predicted during design.
- you want to be able to change an object's subclass after its been instantiated without having to mutate it to a new class.
- you want to be able to nest groupings recursively so that a group is itself an item in another group.

Structure

```
                         Class
    TypeClass <------     type
type-attribute          attribute
                                       ___Classes___
                                         Instances
  TypeObject1<------   Object1A
               |
  TypeObject2  |---    Object1B
```

The Type Object pattern has two concrete classes, one that represents objects and another that represents their types. Each object has a pointer to its corresponding type.

Participants

- TypeClass (Movie)
  - is the class of TypeObject.
  - has a separate instance for each type of Object.
- TypeObject (Star Wars, The Terminator, Independence Day)
  - is an instance of TypeClass.
  - represents a type of Object. Establishes all properties of an Object that are the same for all Objects of the same type.
- Class (Videotape)
  - is the class of Object.

  - represents instances of TypeClass.
- Object (John's Star Wars, Sue's Star Wars)
  - is an instance of Class.
  - represents a unique item that has a unique context. Establishes all properties of that item that can differ between items of the same type.
  - has an associated TypeObject that describes its type. Delegates properties defined by its type to its TypeObject.

TypeClass and Class are classes. TypeObject and Object are instances of their respective classes. As with any instance, a TypeObject or Object knows what its class is. In addition, an Object has a pointer to its TypeObject so that it knows what its TypeObject is. The Object uses its TypeObject to define its type behavior. When the Object receives requests that are type specific but not instance specific, it delegates those requests to its TypeObject. A TypeObject can also have pointers to all of its Objects.

Thus Movie is a TypeClass and Videotape is a Class. Instances of Movie like Star Wars, The Terminator, and Independence Day are TypeObjects. Instances of Videotape like John's Star Wars and Sue's Star Wars are Objects. Since an Object has a pointer to its TypeObject, John's videotape and Sue's videotape have pointers to their corresponding Movie, which in this case is Star Wars for both videotapes. That is how the videotapes know that they contain *Star Wars* and not some other movie.

Collaborations

- An Object gets two categories of requests: those defined by its instance and those defined by its type. It handles the instance requests itself and delegates the type requests to its TypeObject.
- Some clients may want to interact with the TypeObjects directly. For example, rather than iterate through all of the Videotapes the store has in stock, a renter might want to browse all of the Movies that the store offers.
- If necessary, the TypeObject can have a set of pointers to its Objects. This way the system can easily retrieve an Object that fits a TypeObject's description. This would be similar to the allInstances message that Smalltalk classes implement. For example, once a renter finds an appealing Movie, he would then want to know which videotapes the store has that fit the description.

Consequences

The advantages of the Type Object pattern are that it

- allows new "classes" to be created at runtime. These new classes are not actually classes, they are instances called TypeObjects that are created by the TypeClass just like any instance is created by its class.
- avoids an explosion of subclasses needed to represent different types of Objects. Instead of numerous subclasses, the system can use one TypeClass and numerous TypeObjects.
- hides the separation between Object and TypeObject from the Object's clients. The client makes requests of the Object, and the Object in turn decides which requests to forward to the TypeObject. Clients that are aware of the TypeObjects may collaborate with them directly without going through the Objects.
- allows the Object to dynamically change its TypeObject, which has the effect of changing its class. This is simpler than mutating an object to a new class. [DeKezel96]
- allows TypeClass and Class to be subclassed independently.

- can be nested recursively such that a TypeObject in turn acts like the Object for another TypeObject.
- allows an Object to have multiple TypeObjects where each defines some part of the Object's type. The Object must then decide which type behavior to delegate to which TypeObject.

The disadvantages of the Type Object pattern are that it

- factors one logical object into two classes. This is confusing for modelers and programmers alike. It is difficult to recognize or explain the relationship between a TypeObject and an Object. This confusion hurts simplicity and maintainability. In a nutshell: "Use inheritance; it's easier."
- moves implementation differences out of the subclasses and into the state of the TypeObject instances. Whereas each subclass could implement a method differently, now the TypeClass can only implement the method one way and each TypeObject's state must make the instance behave differently.
- requires that each Object keep a reference to its TypeObject. Just as an object knows what its class is, an Object knows what its TypeObject is. But whereas the object system or language automatically establishes and maintains the class-instance relationship, the application must itself establish and maintain the TypeObject-Object relationship.

Implementation

There are several issues to consider when implementing the Type Object pattern:

1. *Object references TypeObject.* The relationship between an Object and its TypeObject is very similar to the relationship between an instance and its class. However, an object system or language establishes and maintains the class-instance relationship automatically, but you must be responsible for the type object-object relationship yourself. When you create an instance of Person called "John," you don't explicitly have to tell the John object that its class is Person; the object already knows that because of the way the language works. However, the language does not know the relationship between an Object and its TypeObject, so you must implement the application to establish that relationship and recognize its meaning.
2. *Object behavior vs. TypeObject behavior.* Because the language does not know the relationship between an Object and its TypeObject, it cannot decide how to delegate behavior between the two. The application must make this decision. The application's decides which responsibilities will be implemented in the Class and which in the TypeClass. The Class' implementation then decides which requests to forward to the TypeObject and which to handle in the Object. The Object may decide to perform additional operations before and after forwarding the request, similar to the way a Decorator can enhance the requests it forwards to its Component [GHJV95, page 175].
3. *Object creation via TypeObject.* A new Object is often created by sending a request to the appropriate TypeObject. This is unusual because the TypeObject is an instance and instance creation requests are usually sent to a class, not an instance. But the TypeObject is like a class to the Object, so it often has the responsibility of creating new Objects.

   Whatever technique the application uses to create the new Object, its TypeObject must be specified when the Object is created.
4. *TypeObject is not multiple inheritance.* The Class—not the TypeObject—is the template for the new Object. The messages that Object understands are defined by its Class, not by its TypeObject. The Class' implementation decides which messages to forward to the TypeObject; the Object does not inherit the TypeObject's messages. Whenever you add behavior to TypeClass, you must also add it to Class for it to become available to the Objects.
5. *Multiple TypeObjects.* An Object can have more than one TypeObject, but this is unusual. In this case, the Class would have to decide which TypeObject to delegate each request to.

Sample Code

For an example implementation of the Type Object pattern, let's look at the implementation of the video store example discussed in the Motivation.

You start with two classes, Movie and Videotape.

```
Object ()
  Movie (title rentalPrice rating ...)
  Videotape (movie isRented renter ...)
```

Notice how the attributes are factored between the two classes. If there are several videotapes of the same movie, some can be rented while others are not. Various copies can certainly be rented to different people. Thus the attributes isRented and renter are assigned at the Videotape level. On the other hand, if all of the videotapes in the group contain the same movie, they will all have the same name, will rent for the same price, and will have the same rating. Thus the attributes title, rentalPrice, and rating are assigned at the Movie level. This is the general technique for factoring the TypeObject out of the Object: Divide the attributes that vary for each instance from those that are the same for a given type.

You create a new Movie by specifying its title. In turn, a Movie knows how to create a new Videotape.

```
Movie class>>newMovieTitle: aString
      ^self new initTitle: aString

Videotape>>initTitle: aString
      title := aString

Movie>>newVideotape
      ^Videotape movie: self

Videotape class>>movie: aMovie
      ^self new initMovie: aMovie

Videotape>>initMovie: aMovie
      movie := aMovie
```

Since Movie is Videotape's TypeClass, Videotape has a movie attribute that contains a pointer to its corresponding Movie instance. This is how a Videotape knows what its Movie is. The movie attribute is set when the Videotape instance is created by Videotape class>>movie:.

A Videotape knows how to be rented. It knows whether it is already being rented. Although it does not know its price directly, it knows how to determine its price.

```
Videotape>>rentTo: aCustomer
      self isRented ifTrue: [^self error].
      aCustomer addRental: self.
      aCustomer chargeForRental: self rentalPrice.
      self makeRented.
      renter := aCustomer

Videotape>>isRented
      ^isRented

Videotape>>rentalPrice
      ^movie rentalPrice

Videotape>>makeRented
      isRented := true
```

Thus it chooses to implement its isRented behavior itself but delegates its rentalPrice behavior to its Type Object.

When *Independence Day* is released on home video, the system creates a Movie for it. It gathers the appropriate information about the new movie via a GUI and executes the necessary code. The system then creates Videotapes using the new Movie.

The system could also support videodisks. The commonalties of videotapes and videodisks are captured in the abstract superclass RentableItem.

```
Object ()
  Movie (title rentalPrice rating ...)
  RentableItem (movie isRented rented ...)
    Videotape (isRewound ...)
    Videodisk (numberOfDisks ...)
```

Most of Videotape's behavior and implementation is moved to RentableItem. Now Videodisk inherits this code for free.

The system could also support movie categories. This requires a new MovieCategory class and refactoring Movie's behavior.

```
Object ()
  MovieCategory (name rentalPrice ...)
  Movie (category title rating ...)
  RentableItem (movie isRented rented ...)
    Videotape (isRewound ...)
    Videodisk (numberOfDisks ...)
```

Before, rentalPrice was a attribute of Movie because all videotapes of the same movie had the same price. Now all movies in the same category will have the same price, so rentalPrice becomes an attribute of MovieCategory. Since Movie now has a type object, it has an attribute—category—to point to its type object.

Now behavior like rentalPrice gets delegated in two stages.

```
Videotape>>rentalPrice
      ^movie rentalPrice

Movie>>rentalPrice
      ^category rentalPrice
```

Once *Independence Day* is no longer a New Release, its category can easily be changed to a General Release because its category is a Type Object and not a class.

```
Movie>>changeCategoryTo: aMovieCategory
  category removeMovie: self.
  category := aMovieCategory.
  category addMovie: self.
```

Thus the system still works primarily with Videotapes, but they delegate their type behavior to Movies, which in turn delegate their type behavior to MovieCategorys. Videotape hides from the rest of the system where each set of behavior is implemented. Each piece of information about a tape is stored in just one place, not duplicated by various tapes. The system can easily add new MovieCategorys, Movies, and Videotapes when necessary. It can even easily change a Movie's MovieCategory when desired.

Known Uses

MovieCategory/Movie/Videotape

A video store application mentioned in the Motivation is an example of the Type Object pattern. It nests the pattern recursively where each MovieCategory has Movie instances and each Movie has Videotape instances. It also shows how the Videotape class may be a specific example of a more general RentableItem class, where Videotape and Videodisk are subclasses of RentableItem. Both concrete classes delegate their type behavior to Movie, so Movie does not need to be subclassed.

Movie may turn out to be a specific example of a more general Title class. Title might have subclasses like Movie, Documentary, and HowTo. Movies have ratings whereas Documentary and HowTo videos often do not. HowTo videos often come in a series or collection that is rented all at once whereas Movies and Documentarys do not. Thus Title might also need a Composite subclass such as HowToSeries. Movie itself might also have subclasses like RatedMovie for those that have MPAA ratings and UnratedMovie for those that don't. However Movie and Title are subclassed, this can be done without affecting the way RentableItem and Videotape are subclassed.

This ability to independently subclass Title and RentableItem would be impossible to achieve if the videotape object had not first been divided into Movie and Videotape components.

ProductType/Product

The TypeObject pattern is often used to separate a ProductType from its Product when the total variety of ProductTypes changes often or is unknown. ProductType instances represent the different types of products a company offers and Product instances represent the specific items the company has built and sold.

For example, there are lots of different kinds of cars; two of them are Mustang and Corvette. If a dealer stocks twenty different Mustangs, lots of the information about each of those cars is redundant. Thus in addition to a Car class to represent the cars, an inventory system needs a MakeAndModel class to represent the various kinds of cars. Each Car has a corresponding MakeAndModel. If a new kind of car comes out, such as Viper, then the system will create a new instance of MakeAndModel for it.

As another example, a hotel might promote a special offer called Weekend Getaway. The Jones make a reservation at the Weekend Getaway rate. The Weekend Getaway offer is a ProductType; the Jones' reservation under that offer is a Product. The reservation object determines its rate by asking its offer object.

SampleType/Sample

The Type Object pattern has been used in the medical field to model medical samples. A sample has four independent properties:

- the system it is taken from (e.g., John Doe)
- the subsystem (e.g., blood, urine, sputum)
- the collection procedure (aspiration, drainage, scraping)
- the preservation additive (heparin, EDTA)

This is easily modeled as a Sample object with four attributes: system, subsystem, collection procedure, and additive. Although the system (the person who gave the sample) is different for almost all samples, the triplet (subsystem, collection procedure, and additive) is shared by a lot of samples. For example, medical technicians refer to a "blood" sample, meaning a blood/aspiration/EDTA sample. Thus the triplet attributes can be gathered into a single SampleType object.

A SampleType is responsible for creating new Sample objects. There are about 5,000 different triplet combinations possible, but most of them don't make any sense, so the system just provides the most common SampleTypes. If another SampleType is needed, the users can create a new one by specifying its subsystem, collection procedure, and additive. While the system tracks tens of thousands of Samples, it only needs to track about one-hundred SampleTypes. So the SampleTypes are TypeObjects and the Samples are their Objects. [DeKezel96]

Power Type: TreeSpecies/Tree

James Odell uses the example of TreeSpecies/Tree to illustrate power types, his name for Type Objects. An instance of TreeSpecies describes a type of tree such as American elm, sugar maple, apricot, or saguaro. An instance of Tree represents a tree, such as the one in my front yard or the one in your back yard. Each Tree has a corresponding TreeSpecies that describes what type of tree it is. A Tree is just a tree but its TreeSpecies tells you that the tree is a sugar maple tree. [MO95]

Signal/Exception

The Type Object pattern is much more common in domain frameworks than vendor frameworks, but one vendor example is the Signal/Exception framework in VisualWorks Smalltalk. When Smalltalk code encounters an error, it can raise an Exception. The Exception records the context of where the error occurred for debugging purposes. Yet the Exception itself doesn't know *what* went wrong, just *where*. It delegates the *what* information to a Signal. Each Signal describes a potential type of problem such as user-interrupt, message-not-understood, and subscript-out-of-bounds. Thus two message-not-understood errors create two separate Exception instances that point to the same Signal instance. Signal is the TypeClass and Exception is the Class. [VW95]

Related Patterns

A TypeObject implementation can become complex enough that there are Class and TypeClass hierarchies. These hierarchies look a lot like the Abstraction and Implementor hierarchies in the Bridge pattern [GHJV95, page 151], where Class is the abstraction and TypeClass is the implementation. However, clients can collaborate directly with the TypeObjects, an interaction that usually doesn't occur with ConcreteImplementors.

An Object can seem to be a Decorator [GHJV95, page 175] for its TypeObject. An Object and its TypeObject have similar interfaces and the Object chooses which messages to forward to its TypeObject and which ones to enhance. However, a Decorator does not behave like an instance of its Component.

The TypeObjects can seem like Flyweights [GHJV95, page 195] to their Objects. Two Objects using the same TypeObject might think that they each have their own copy, but instead are sharing the same one. Thus it is important that neither Object change the intrinsic state of the TypeObject.

Ralph Johnson has also documented the Type Object pattern, which he also refers to as "Reified Type." [Johnson96]

References

[DeKezel96] De Kezel, Raoul. E-mail correspondence.

[GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995; http://www.aw.com/cp/Gamma.html.

[Johnson96] Johnson, Ralph. "Type Object." Submitted to EuroPLoP '96.

[MO95] Martin, James and James Odell. *Object Oriented Methods: A Foundation*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[VW95] VisualWorks Release 2.5, ParcPlace-Digitalk, Inc., Sunnyvale, CA, 1995; http://www.parcplace.com.