The Type Object Pattern

Ralph Johnson
johnson@cs.uiuc.edu

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, lets a system provide its own type–checking rules, and can lead to simpler, smaller systems.

Also Known As

Power Type [MO95], Item Descriptor [Coad93], Metaobject [KRB91]

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. A design in which a class has an unknown number of subclasses can be converted to one in which the class has an unknown number of instances.

Consider 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* 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.

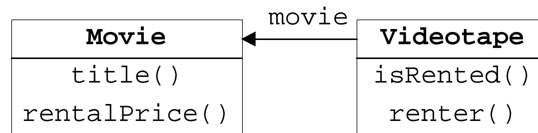
One way to solve this problem is to create a subclass of `Videotape` for each movie. Thus two of the subclasses would be `StarWarsTape` and `TerminatorTape`. The class itself would keep the information for that movie. So the information common to all copies of *Star Wars* would be stored only once. It 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

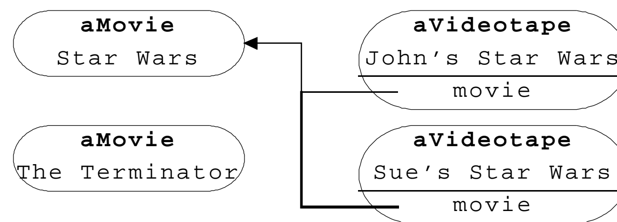
[1] The Motion Picture Association of America, the industry group that rates movies in the United States as G, PG, R, etc.

same lack of flexibility that `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`.



This class 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.



This instance diagram 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. It also 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 types of application instances. Each application instance has a pointer to its corresponding type.

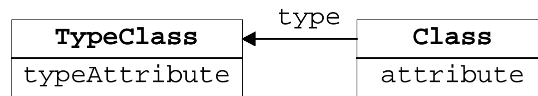
Applicability

Use the Type Object pattern when:

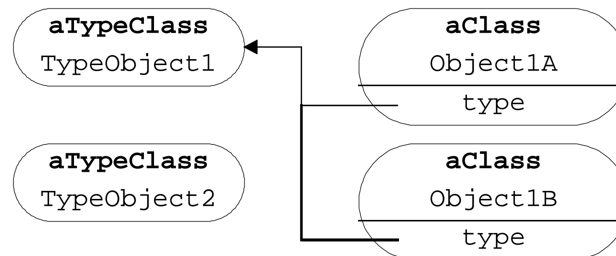
- instances of a class need to be grouped together according to their common attributes and/or behavior.
- the class needs a subclass for each group to implement that group's common attributes/behavior.

- 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 were not 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



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.



For example, the system uses a `TypeObject` to represent each type in the system and an `Object` to represent each of the instances of those `TypeObjects`. Each `Object` has a pointer to its `TypeObject`.

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:

- *Runtime class creation.* The pattern 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 subclass explosion.* The system no longer needs numerous subclasses to represent different types of Objects. Instead of numerous subclasses, the system can use one TypeClass and numerous TypeObjects.
- *Hides separation of instance and type.* An Object's clients does not need to be aware of the separation between Object and TypeObject. 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.
- *Dynamic type change.* The pattern 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]
- *Independent subclassing.* TypeClass and Class can be subclassed independently.
- *Multiple Type Objects.* The pattern 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:

- *Design complexity.* The pattern factors one logical object into two classes. Their relationship, a thing and its type, is difficult to understand. 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."
- *Implementation complexity.* The pattern 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.
- *Reference management.* Each `Object` must 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 that you must always address when implementing the Type Object pattern:

1. *`Object` references `TypeObject`.* Each `Object` has a reference to its `TypeObject`, and delegates some of its responsibility to the `TypeObject`. An `Object`Õs `TypeObject` must be specified when the `Object` is created.
2. *`Object` behavior vs. `TypeObject` behavior.* An `Object`Õs behavior can either be implemented in its class or can be delegated to its `TypeObject`. The `TypeObject` implements behavior common to the type, while the `Object` implements behavior that differs for each instance of a type. When the `Object` delegates behavior to its `TypeObject`, it can pass a reference to itself so that the `TypeObject` can access its data or behavior. 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].
1. *`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 a delegating method to `Class` before the behavior is available to the `Objects`.

There are other issues you may need to consider when implementing the Type Object pattern:

3. *`Object` creation using a `TypeObject`.* Often, a new `Object` is created by sending a request to the appropriate `TypeObject`. This is notable 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`.
4. *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.
5. *Changing `TypeObject`.* The Type Object pattern lets an object dynamically change its "class," the type object. It is simpler for an object to change its pointer to a different type object (a different instance of the same class) than to mutate to a new class.
6. For example, suppose 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.

7. *Subclassing `Class` and `TypeClass`.* It is possible to subclass either `Class` or `TypeClass`. The video store could support videodisks by making another `Class` called `Videodisk`. 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`.

8. The hard part of Type Object occurs after it has been used. There is an almost irresistible urge to make the `TypeObjects` more composable, and to build tools that let non–programmers specify new `TypeObjects`. These tools can get quite complex, and the structure of the `TypeObjects` can get quite complex. Avoid any complexity unless it brings a big payoff.

Sample Code

Video Store

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>>title: aString
      ^self new initTitle: aString

Movie>>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 checkNotRented.
      aCustomer addRental: self.
      self makeRentedTo: aCustomer

Videotape>>checkNotRented
      isRented ifTrue: [^self error]
```

```
Customer>>addRental: aVideotape
      rentals add: aVideotape.
      self chargeForRental: aVideotape rentalPrice

Videotape>>rentalPrice
      ^self movie rentalPrice

Videotape>>movie
      ^movie

Movie>>rentalPrice
      ^rentalPrice

Videotape>>makeRentedTo: aCustomer
      isRented := true.
      renter := aCustomer
```

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 (title, rental price, rating, etc.) via a GUI and executes the necessary code. The system then creates the new `Videotapes` using the new `Movie`.

Video Store–Nested Type Objects

The Type Object pattern can be nested recursively. For example, many video stores have 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`.

A `MovieCategory` class requires refactoring `Movie`'s behavior.

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

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 and implemented by the third.

```
Videotape>>rentalPrice
      ^self movie rentalPrice

Movie>>rentalPrice
      ^self category rentalPrice

MovieCategory>>rentalPrice
      ^rentalPrice
```

This example nests the Type Object pattern recursively where each `MovieCategory` has `Movie` instances and each `Movie` has `Videotape` instances. 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 by creating new instances.

Video Store–Dynamic Type Change

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 its class.

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

With the Type Object pattern, an Object can easily change its Type Object when desired.

Video Store–Independent Subclassing

The system could also support videodisks. The commonalties of videotapes and videodisks are captured in the abstract superclass `RentableItem`, where `Videotape` and `Videodisk` are subclasses. Both concrete classes delegate their type behavior to `Movie`, so `Movie` does not need to be subclassed.

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

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

`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 [GHJV95, page 163] 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.

```
Object ()
  MovieCategory (name rentalPrice ...)
  Title (category title ...)
    Documentary (...)
    HowTo (...)
    Movie (...)
      RatedMovie (rating ...)
      UnratedMovie (...)
    TitleComposite (children ...)
      HowToSeries (...)
  RentableItem (movie isRented rented ...)
    Videotape (isRewound ...)
    Videodisk (numberOfDisks ...)
```

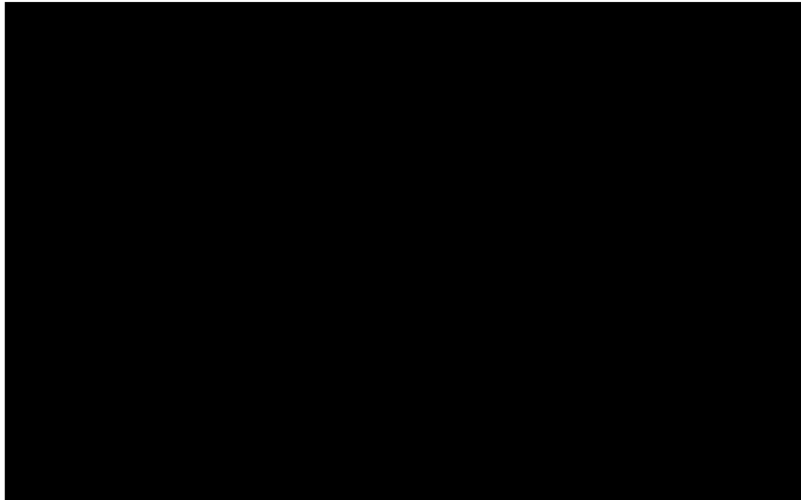
`Movie` and `Title` can be subclassed without affecting the way `RentableItem` and `Videotape` are subclassed. This ability to independently subclass `Title` and `RentableItem` would be impossible to achieve if the videotape object had not first been divided into `Movie` and `Videotape` components. Obviously, all of this nesting and subclassing can get complex, but it shows the flexibility the Type Object pattern can give you–flexibility that would be impossible without the pattern.

Manufacturing

Consider a factory with many different machines manufacturing many different products. Every order has to specify the kinds of products it requires. Each kind of product has a list of parts and a list of the kinds of machines needed to make it. One approach is to make a class hierarchy for the kinds of machines and the kinds of products. But this means that adding a new kind of machine or product requires programming, since you have to define a new class. Moreover, the main difference between different products is how they are made. You can probably specify a new kind of product just by specifying its parts and the sequence of machine tools that is needed to make it.

It is better to make objects that represent "kind of product" and "kind of machine." They are both examples of type objects. Thus, there will be classes `Machine`, `Product`, `MachineType`, and `ProductType`. A `ProductType` has a "manufacturing plan" which knows the `MachineTypes` that make it. But a particular instance of `Product` was made on a particular set of `Machines`. This lets you tell which machine is at fault when a product is defective.

Suppose we want to schedule orders for the factory. When an order comes in, the system will figure out the earliest that it can fill the order. Each order knows what kind of product it is going to produce. For simplicity, weÕll assume each order consists of one kind of product. WeÕll also assume that each kind of product is made on one kind of machine. But that product is probably made up of other products, which will probably require many other machines. Thus, `Product` is an example of the Composite pattern [GHJV95, page 163] (not shown below). For example, a hammer consists of a handle and a head, which are combined at an assembly station. The wooden handle is carved at one machine, and the head is cast at another. `ProductType` and `Order` are also composites, but are not shown.



There are six main classes:

```
Object ()
  MachineType (name machines ...)
  Machine (type location age schedule ...)
  ProductType (manufacturingMachine duration parts ...)
  Product (type creationDate machine parts ...)
  Order (productType dueDate requestor parts ...)
  Factory (machines orders)
```

We will omit all the accessing methods, since they are similar to those in the video store example. Instead, we will focus on how a factory schedules an order.

A factory acts as a Facade [GHJV95, page 185], creating the order and then scheduling it.

```
Factory>>orderProduct: aType by: aDate for: aCustomer
      | order |
      order := Order product: aType by: aDate for: aCustomer.
      order scheduleFor: self.
      ^order
```

```
Order>>scheduleFor: aFactory
      | partDate earliestDate |
      partDate := dueDate minusDays: productType duration.
      parts := productType parts collect: [:eachType |
            aFactory
                  orderProduct: eachType
                  by: partDate
                  for: order]
      productType
            schedule: self
            between: self datePartsAreReady
            and: dueDate
```

```
ProductType>>schedule: anOrder between: startDate and: dueDate
      (startDate plusDays: duration) > dueDate
            ifTrue: [anOrder fixSchedule].
      manufacturingMachine
            schedule: anOrder
            between: startDate
            and: dueDate
```

There are at least two different subclasses of `ProductType`, one for machines that can only be used to make one product at a time, and one for assembly lines and other machines that can be pipelined and so make several products at a time. A non–pipelined machine type is scheduled by finding a machine with a schedule with enough free time open between the `startDate` and the `dueDate`.

```
NonpipelinedMachineType>>schedule: anOrder between: startDate and:
dueDate
      machines do: [:each | | theDate |
            theDate := each schedule
                        slotOfSize: anOrder duration
                        freeBetween: startDate
                        and: dueDate.
            theDate notNil
                  ifTrue: [^each schedule: anOrder at: theDate]].
      anOrder fixSchedule.
```

A pipelined machine type is scheduled by finding a machine with an open slot between the `startDate` and the `dueDate`.

```
PipelinedMachineType>>schedule: anOrder between: startDate and:
dueDate
      machines do: [:each | | theDate |
            theDate := each schedule
                        slotOfSize: 1
                        freeBetween: startDate
                        and: dueDate.
            theDate notNil
                  ifTrue: [^each schedule: anOrder at: theDate]].
      anOrder fixSchedule.
```

This design lets you define new `ProductTypes` without programming. This lets product managers, who usually arenÕt programmers, specify a new product type. It will be possible to design a tool that product managers can use to define a new product type by specifying the manufacturing plan, defining the labor and raw materials needed, the price of the final product, and so on. As long as a new kind of product can be defined without subclassing `Product`, it will be possible for product managers to do their work without depending on programmers.

There are constraints between types. For example, the sequence of actual `MachineTools` that manufactured a `Product` must match the `MachineToolTypes` in the manufacturing plan of its `ProductType`. This is a form of type checking, but it can only be done at runtime. It might not be necessary to check that the types match when the sequence of `MachineTools` is assigned to a `Product`, because this sequence will be built by iterating over a manufacturing plan to find the available `MachineTools`. However, scheduling can be complex, and errors are likely, so it is

probably a good idea to double–check that a `Product`'s sequence of `MachineTools` matches what its `ProductType` says it should be.

Known Uses

Coad

CoadÕs Item Description pattern is the Type Object pattern except that he only emphasized the fact that a Type holds values that all its Instances have in common. He used an "aircraft description" object as an example. [Coad92]

Hay

Hay uses Type Object in many of his data modeling patterns, and discusses it as a modeling principle, but doesn't call it a separate pattern. He uses it to define types for activities, products, assets (a supertype of product), incidents, accounts, tests, documents, and sections of a Material Safety Data Sheet. [Hay96]

Fowler

Fowler talks about the separate Object Type and Object worlds, and calls these the "knowledge level" and the "operational level." He uses Type Object to define types for organizational units, accountability relationships, parties involved in relationships, contracts, the terms for contracts, and measurements, as well as many of the things that Hay discussed. [Fowler97]

Odell

Odell's Power Type pattern is the Type Object pattern. He illustrates it with the example of tree species and tree. A tree species describes a type of tree such as American elm, sugar maple, apricot, or saguaro. A tree represents a particular tree in my front yard or the one in your back yard. Each tree has a corresponding tree species that describes what kind of tree it is. [MO95]

Sample Types and Samples

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]

Signals and Exceptions

The Type Object pattern is 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]

Reflection

Type Object is present in most reflective systems, where a type object is often called a metaobject. The class/instance separation in Smalltalk is an example of the Type Object pattern. Programmers can manipulate classes directly, adding methods, changing the class hierarchy, and creating new classes. By far the most common use of a class is to make instances, but the other uses are part of the culture and often discussed, even if not often used. [KRB91]

Reflection has a well–deserved reputation for being hard to understand. Type Object pattern shows that it does not have to be difficult, and can be an easy entrance into the more complex world of reflective programming.

Related Patterns

The Type Object pattern is similar to the Strategy and State patterns [GHJV95, page 315 and page 305]. All three patterns break an object into pieces and the Òreal objectÓ delegates to the new object–either the Type Object, the Strategy, or the State. Strategy and State are usually pure behavior, while a Type Object often holds a lot of shared state. States change frequently, while Type Objects rarely change. A Strategy usually has one main responsibility, while a Type Object usually has many responsibilities. So, the patterns are not exactly the same, even though their object diagrams are similar.

Any system with a Type Object is well on its way to having a Reflective Architecture [BMRSS96]. Often a Type Object holds Strategies for its instances. This is a good way to define behavior in a type.

A Type Object implementation can become complex enough that there are Class and Type Class 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 Type Class is the implementation. However, clients can collaborate directly with the Type Objects, an interaction that usually doesn't occur with Concrete Implementors.

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

The Type Objects can seem like Flyweights [GHJV95, page 195] to their Objects. Two Objects using the same Type Object 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 Type Object.

Another way to make one object act like the type of another is with the Prototype pattern [GHJV95, page 117], when each object keeps track of its prototype and delegates requests to it that it does not know how to handle.

References

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern–Oriented Software Architecture – A System of Patterns*. Wiley and Sons Ltd., 1996.

[Coad92] Peter Coad. "Object–oriented Patterns," *Communications of the ACM*. 35(9):152–159, September 1992.

[Fowler97] Martin Fowler. *Analysis Patterns: Reusable Object Models,* Addison–Wesley, 1997.

[DeKezel96] Raoul De Kezel. E–mail correspondence.

[GHJV95] Erich Gamma, 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.

[Hay96] David Hay. *Data Modeling Patterns*, Dorsett House Publishing, 1996.

[KRB91] Gregor Kiczales, Jim des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, Massachusetts, 1991.

[MO95] James Martin 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.