Don’t Use Arrays?

Juanita J. Ewing
Instantiations, Inc.

Derived from Smalltalk Report

This column discusses inappropriate use of arrays, and how misuse affects reusability. We will analyze several Smalltalk methods that use arrays and revise them to use classes instead of arrays. We will also show you how to search your image for methods that use arrays.

Motivation
A class in Smalltalk is a specification of behavior and supporting data. Each instance contains a particular set of related data. For example, the data for an instance of Rectangle is two points. The points are related because they are both part of a rectangle: one is the origin point and the other is the corner point.

In Smalltalk, you can also use a data structure such as an array to represent related data. Instead of the class Rectangle, you could use an array with the first element of the array being the origin point and the second element being the corner point. Which is more reusable?

First, let’s examine how clients access data. Clients of the class Rectangle can send the messages origin and corner. Clients of the rectangle-as-array must access the correct element by specifying the index, and the index might not have any correlation to the values stored in the array.

Accessing the data is not the only consideration. Rectangle has specialized behavior, such as height, containsPoint:, intersects:, and expandBy:. The rectangle-as-array has no specialized behavior. For example, each client that needed the height of the rectangle-as-array would have to duplicate the code that subtracted the two y coordinates to obtain the height of the rectangle.

There are three reasons why the class is more reusable than the array:

- Ease of Use: Clients of the rectangle-as-array need to know arbitrary indices to obtain the data. Clients of the rectangle-as-class send messages with meaningful names.

• Encapsulation: The behavior of rectangle is not encapsulated with the data in the rectangle-as-array. Clients of the rectangle-as-array would need to write much more code than the clients of the rectangle-as-class in order to duplicate the behavior of rectangle. Most clients would write the same code over and over.

• Information Hiding: The constituent data for the rectangle is accessible to all clients in the rectangle-as-array. Indeed, it must be in order for clients to the duplicate the behavior of Rectangle. But, it also means the rectangle-as-array cannot change its representation without affecting all its clients.

Inappropriate Use I

Standard Smalltalk even provides us with a bad example of array usage (nobody's perfect). On page 109 of *Smalltalk-80: The Language and its Implementation* is the specification of a class method for Date.

Date class protocol

general inquiries

dateAndTimeNow	Answer an Array whose first element is the current date (an instance of class Date representing today's date) and whose second element is the current time (an instance of class Time representing the time right now).

Here is one possible implementation of the method.

Date class methods

```
dateAndTimeNow
        "Answer an Array of two elements. The first element is
        a Date representing the current date and the second
        element is a Time representing the current time."

        ^ (Array new: 2)
                at: 1 put: self today;
                at: 2 put: Time now;
                yourself
```

Clients of this method must keep track of which elements are where in the array. The code to compare two data-and-time arrays looks like this (the variables now and then contain date-and-time arrays):

```
| now then oldest |
then := self oldDateAndTime.
now := Date dateAndTimeNow.
((now at: 1) >= (then at: 1) and: [(now at: 2) > (then at: 2)])
```

```
        ifTrue: [oldest := then]
```

This kind of code is not easy to read, and is likely to be duplicated in an application that manipulates time stamps.

In the dateAndTimeNow method, the array is merely a shortcut way of implementing a return of two values. The elements in the array have nothing to do with their indices. Clients have to remember which element is which. They also have to remember the algorithm for comparing date/time pairs. This kind of shortcut is not good coding practice because it does not facilitate reuse.

A better solution is to create a new class that represents an associated date and time. We will call this class TimeStamp. It would have messages for accessing its date and time, and for comparing itself with other TimeStamps. Using this new class, the dateAndTimeNow method can be rewritten:

```
Date class methods
    dateAndTimeNow
        "Answer an instance of TimeStamp containing the current date
        and the current time."

        ^TimeStamp date: self today time: Time now
```

Even better would be to eliminate the Date method and create a TimeStamp method that returns the current date and time. A TimeStamp method is better because the instance is created in the class that relates date and time. The Date class is a less desirable location because dates don't have an explicit relationship with time. Time is not referenced in other Date methods.

```
TimeStamp class methods
    now
        "Answer an instance of the receiver containing the current date
        and time."

        | current |
        current := self new.
        current date: Date today.
        current time: Time now.
        ^current
```

The client of this functionality can now write much simpler fragments of code.

```
    | now then oldest |
    then := self oldTimeStamp.
    now := TimeStamp now.
    now > then
```

```
        ifTrue: [oldest := then].
```

Inappropriate Use II

A method from Directory provides us with another inappropriate use of an array. In this method, a collection of arrays provides detailed information about each file in a directory.

```
Directory methods
    formatted
        "Answer a collection of arrays of file information for
        the receiver directory. Each array has four entries:
        file name, size, date/time and attributes."

            | answer fileEntries anArray |
            fileEntries := self contents.
            answer := OrderedCollection new: fileEntries size.
            fileEntries do: [ :each |
                anArray := Array new: 5.
                anArray
                    at: 1 put: (Directory extractFileNameFrom: each);
                    at: 2 put: (Directory extractSizeFrom: each);
                    at: 3 put: (Directory extractDateTimeFrom: each);
                    at: 4 put: (Directory extractResourceSizeFrom: each);
                    at: 5 put: (Directory extractCreatorTypeFrom: each).
                answer add: anArray].
            ^ answer
```

Note that the method comment is wrong. It references an array with four entries, but the code has an array with five entries, indicating that a small change in the implementation has a big impact on clients. Users of this method must know where relevant information is stored in the array. It is impossible to tell, from either the comment or the code, which array element is new.

In this fragment of code, the client of Directory needs the names of files of zero length. This code must reference elements stored at arbitrary locations, and requires heavy commenting to be maintainable.

```
    | zeros |
    zeros := myDirectory formatted
                    select: [:info | (info at: 2) = 0].  "size is stored at 2"
    ^zeros collect: [:info | info at: 1]   "name is stored at 1"
```

Related data stored in arrays is more appropriate as an instance of a class. In this example, the information stored in an array represents detailed status information about a file. An alternate solution is to create a class, called

FileInformation to store this data. FileInformation has a class method to create new instances, and instance methods to access its components. A partial class specification follows:

FileInformation methods

fromFileEntry: aFileEntry
Create and return an instance of the receiver for a file entry

FileInformation methods

fileName
Return the name of the file.
size
Return the size of the file, including both the data and resource fork.
timeStamp
Return the date and time when the receiver was last modified.
resourceSize
Return the size of the resource part of the file.
creatorType
Return the code that indicates the application that created the file.

With the FileInformation class, we can eliminate the usage of Array and incorporate usage of our new class. The formatted method now looks like:

Directory methods

formatted

```
"Answer a collection of file informations, one for each entry in the
    receiver. "

    | answer fileEntries anArray |
    fileEntries := self contents.
    answer := OrderedCollection new: fileEntries size.
    fileEntries do:
        [ :each |
        answer add: (FileInformation fromFileEntry: each)].
    ^ answer
```

Clients of this method can then use meaningful selectors instead of indexing into an array. This code is more maintainable now and doesn't need any extra commenting.

```
| zeros |
zeros := myDirectory formatted select: [:info | info size = 0]
^zeros collect: [:info | info fileName]
```

There are good examples of Array use in your Smalltalk system. These are uses in which the index is a relevant part of the data structure, such as a

numeric id allocated by the operating system. The array contains the relationship between the id and a related Smalltalk object. Literal arrays are convenient for collections of values.

Identifying Inappropriate Use

You can look for inappropriate use of array and other data structures in your image. Use these techniques to find methods that reference Array. You may also want to look for references to other data structures such as OrderedCollection.

In Team/V
    In the Package Browser select Array. Select the menu item Class/Browse Refs.
In Smalltalk/V for OS/2 and Smalltalk/V Windows
    Execute Smalltalk sendersOf: (Smalltalk associationAt: #Array)
In Smalltalk/V Mac
    Execute Smalltalk referencesTo: #Array
In Objectworks\Smalltalk
    In the System Browser select Array. Select the menu item Class Refs from the class pane menu.

When examining a method, inappropriate use will have one or more of the following characteristics:

- indices that are irrelevant to data and functionality
- array elements that are related by some abstraction NOT captured by a class
- awkward client use due to violation of information hiding and encapsulation

If you find a methods that uses arrays inappropriately, you should improve the quality of your code by

1. creating classes to represent related array elements and
2. rewriting offending methods to reference new classes and to eliminate arrays.

Conclusion

Don't use arrays as a shortcut to pass around related items. Instead, create a class to represent the abstraction relating the items. Your code will immediately be more understandable, extensible, maintainable and reusable. Classes are the basic building blocks of Smalltalk programs. Use them.