Limitations of Data Encapsulation and Abstract Data Types

Paul L. Bergstein
University of Massachusetts Dartmouth
pbergstein@umassd.edu

Abstract

One of the key benefits provided by object-oriented programming languages is support for strong data encapsulation and user defined abstract data types. The use of these features is intended to improve the resiliency of programs to changes in data representation by localizing the effects. Recently, the software industry has experienced the problems caused at least in part by a change in the representation of dates – the Y2K problem.

In this paper, we consider how the use of data encapsulation and abstract data types might have hypothetically prevented these Y2K problems. We consider examples that demonstrate that data encapsulation and abstract data types as they are commonly defined and used are not enough. Rather, it is necessary to provide an interface that not only encapsulates data, but also presents only an abstract view of the data. Finally, these rules are formulated as an extension and generalization of the Law of Demeter.

1. Introduction

One of the key benefits provided by object-oriented programming languages is support for strong data encapsulation and user defined abstract data types [1-5, 6-9]. Strong data encapsulation refers to the prevention of access to data defined in a class from outside of the class (e.g. by declaring a class member *private*). Abstract data types are types that are defined operationally as opposed to structurally. That is, the type is defined in terms of the operations (or methods) it supports rather than on its internal structure. A class definition in a typical class based object-oriented programming language where all data members have been declared private would generally be considered an implementation of an abstract data type. An interface in Java where only method signatures are defined is an even better example of an abstract data type.

The use of these features is intended to improve the resiliency of programs to changes in data representation by localizing the effects of the changes. Until recently the classic example of software breaking due to change in data representation was the United States Post Office adoption of zip+4 postal codes. When the representation of zip codes was extended to include an additional 4 digits, the cost of modifying software to handle the seemingly trivial change was reported to be in the hundreds of millions of dollars [9, p.18]. More recently, we have experienced the Y2K problem, which is at least partially due to changing from a 2-digit to a 4-digit representation of years. In this case, the cost estimates reported in the press range in the hundreds of billions of dollars. In this paper, we imagine that we are able to go back in time and introduce object-oriented software engineering techniques before any software using 2-digit dates is constructed. We imagine that these techniques are used in the ways that they are typically applied today, and we consider how the use of these techniques would have impacted the Y2K crisis, and why.

2. Object-oriented techniques applied to dates

If we were to implement an abstract data type for dates, the public interface would probably look very similar to the Java interface shown in Figure 1. The interface includes methods for comparing dates, for parsing and formatting dates, and for manipulating the month, day, and year of dates. In fact, the methods included in our interface are a subset of the methods from the Java 1.0 java.util.Date class[1].

The Date definition in Figure 1 clearly meets the definition of an abstract data type since it defines dates in terms of operations only. A class implementing this interface would include implementations of each of the public methods as well as whatever private data members and methods are necessary to support the public methods. Such an implementation meets the criteria of strong data encapsulation. Notice that the methods for manipulating month, day, and year do not necessarily imply anything about the underlying internal representation of Date objects. That is, the *get* and *set* methods are not necessarily accessor functions for month, day, and year data members. Also notice that the getYear and setYear methods return an int and take an int as argument, respectively, so the interface should be stable for more than 2 billion years. There are many possibilities for the data structures that could be used in implementations of

[1] In java 1.1 the parsing and formatting methods were moved to java.text.DateFormat, and the methods for manipulating month, day, and year were moved to java.util.Calendar.

```
interface Date {

    // Methods for comparing dates
    public boolean before(Date d);
    public boolean after(Date d);
    public boolean equals(Date d);

    // Formatting & parsing methods
    public String toString();
    public void parse(String s);

    // Methods for manipulating
    // month, day, and year
    public int getMonth();
    public void setMonth(int
month);
    public int getDay();
    public void setDay(int day);
    public int getYear();
    public int setYear(int year);
}
```

Figure 1

the Date interface. For example, the year might be stored as a character array of length two, but since we have used strong data encapsulation, changing to a character array of length four should only require local changes in classes implementing the Date interface. It seems that the use of current software engineering techniques would have prevented the Y2K crisis! Unfortunately, there is a serious flaw in the approach we have taken, and it is doubtful that the use of abstract data types and strong data encapsulation by themselves would have had much impact on the Y2K problem.

3. Example Usage

Although the interface definition fits the criteria of abstract data type definitions, the inclusion of the *get* and *set* methods invites clients (users of Date implementations) to view dates as compositions of month, day, and year, rather than as abstractions of the date concept, e.g. a point in time. It also provides clients with the means to circumvent the comparison, formatting, and parsing methods, by performing these functions outside of the class implementing the interface. Consider, for example, the code in Figure 2. Here the programmer has written some code to display textboxes for the user to enter the month and year of an expiration date, and then check to see if the expiration date has passed. The code works only if the user enters for the year the last two digits of a year in the range 1900-1999. The code breaks when expiration dates reach the year 2000, despite the use of a Date abstract data type with a stable interface and programming language enforced strong data encapsulation. The trouble is that the getYear and getMonth methods encourage clients to perform parsing and comparisons of months and years outside the class implementing the Date interface.

```
// Prompt user for month and year of expiration, and check to see
// if the expiration date has passed

Date today;
Textfield month, year;
Boolean expired;

// Code to initialize variables, display user interface, etc.
// ...

int todayMonth = today.getMonth();
int todayYear = today.getYear();
int expMonth = Integer.parse(month.getText()).intValue();
int expYear = Integer.parse(year.getText()).intValue();

if (expYear + 1900 == today.getYear())
    expired = (todayMonth > expMonth);
else
    expired = (today.getYear() > expYear + 1900);
```

Figure 2

```
interface Date {

    // Precision constants
    public static final int YEAR = 0;
    public static final int MONTH = 1;
    public static final int DAY = 2;

    // Methods for comparing dates
    public boolean before(Date d);
    public boolean after(Date d);
    public boolean equals(Date d);
    public boolean before(Date d, int precision);
    public boolean after(Date d, int precision);
    public boolean equals(Date d, int precision);

    // Formatting and parsing of dates
    public String toString();
    public void parse(String s);
}
```

Figure 3

```
// Prompt user for month and year of expiration, and check to see
// if the expiration date has passed

Date today, expiration;
Textfield month, year;
Boolean expired;

// Code to initialize variables, display user interface, etc.
// ...

String expDate = month.getText() + "/" + year.getText();
expiration.parse(expDate);
expired = today.after(expiration, Date.MONTH);
```

Figure 4

The problem could be avoided by performing the parsing and comparison within the class implementing the Date interface. This requires extending the parse method to recognize string formats containing only month and year information, e.g. mm/yy, and overloading the comparison functions with versions that take a precision with which to perform the comparison, e.g. to the nearest year, month, day, hour, minute, second, or millisecond. Figure 3 gives the improved Date interface, and Figure 4 illustrates the correct implementation of the client code. Now the effects of date format changes and Y2K are localized in the class implementing the Date interface. As a fringe benefit, we get client code that is simpler and more readable. We have also improved the reusability of some parsing and comparison code by including it in the Date interface.

The problems caused by the *get* and *set* methods could be even worse if the interface to those methods was not robust with respect to changes in data representation. Consider, for example, the damage that might have resulted from a getYear method that returned a value in the

range 0-99 as a byte (8-bit integer), or as a character array containing only the last two digits of the year.

While this example looks mainly at the harm caused by the getYear method, the setYear method is also likely to cause problems. Consider client code that parses a two-digit year, interprets it as a year in the range 1900-1999, and then sends a setYear message. In this case, we would have been better off if the setYear method took an argument in the range 0-99, so that the interpretation could be modified in the class implementing the Date interface.

4. Discussion

As the preceding example illustrates, methods that allow clients to view objects as compositions rather than as abstractions may cause serious repercussions. Defining abstract data types is not enough. Implementers of both the abstract data types and the client code that uses them must learn to take an abstract view of the data – but it is not always easy or natural. People don't generally think about dates in an abstract way, e.g. as points in time. They think of a date as a month, a day, and a year, so it is natural for the implementers of date classes to supply methods for manipulating month, day, and year, and it is natural for implementers of client code to expect or request such methods.

The lesson here is that "accessors", whether or not they correspond to actual data members, may break the abstraction of a class, with all of the ensuing bad consequences, even if they do not break the encapsulation. Whenever we provide an accessor method we should first ask how the access will be used, and whether or not it is truly necessary. In most cases, it will be better to move the functionality for which access is required out of client code and into the class where the data is encapsulated. When writing client code, we should avoid using accessors, even if they are available, whenever possible.

The elimination of accessor use might be viewed as an extension or generalization of the Law of Demeter [6], which restricts the set of objects to which clients can send messages. Suppose an object of class A holds a reference to an object of class B which in turn holds a reference to an object of class C. It would be a violation of the Law of Demeter for the A object to request from the B object the reference to the C object (via a *get* method) and then use that reference to send messages directly to the C object. Instead, the A object should send messages only to the B object, delegating responsibility for communicating with C. The effect is a decrease in the coupling and brittleness of the code.

The suggestion here is that in addition to restricting the set of objects that can be sent messages, there should be a restriction on the kind of messages they can be sent. The messages that are sent to an object should be restricted to those that treat the object as an abstraction. This is a generalization of the principle that we should delegate rather than use *get* methods to retrieve object references for message sending. It says that we should delegate rather than use *get* methods to retrieve any data for any type of manipulation. It is also an extension, in that it says we should not use *set* methods if they are used to manipulate a component of the object, rather than the abstraction as a whole.

5. Conclusions

Data encapsulation and user defined abstract data types are powerful mechanisms for creating flexible and robust data representations, but they are not enough to prevent a future Y2K type crisis. Even strict adherence to rules such as "declare all data members private" will not insulate software from changes in data representations, unless programmers learn to think about abstract data types abstractly. The "No Accessors" rule goes a long way toward forcing programmers to think more abstractly and can result in code which is significantly more robust. Fringe benefits may include more readable code and additional opportunities for reuse.

On the other hand, the benefits of working with objects at a higher level of abstraction do not come without a cost. In the short run, it may be quicker and easier to write code using accessors, but the maintenance risk may be very high. There are surely situations where it is best to break the Law of Demeter, and there are situations where it is best to break the No Accessors rule. The point is to understand the reasons why the rules exist, and to make sure that if a rule is broken that the benefits outweigh the risks.

6. References

[1] Appelbe, W. and Ravn, A. "Encapsulation Constructs in Systems Programming Languages", *ACM Transactions on Programming Languages and Systems*, 6(2), 1984.

[2] Booch, Grady, Object-Oriented Analysis and Design with Applications, 2nd Edition, Addison-Wesley, 1994.

[3] Claybrook, B. and Wyckof, M., "Module: an Encapsulation Mechanism for Specifying and Implementing Abstract Data Types", *Communications of the ACM*, 1980.

[4] Cohen, A. “Data Abstraction, Data Encapsulation, and Object-Oriented Programming”, *SIGPLAN Notices*, (19)1, 1984.

[5] Guttag, John V., “Abstract Data Types and the Development of Data Structures”, Communications of the ACM, 20(6), June 1977, pp. 396-404.

[6] Lieberherr, Karl and Holland, Ian, “Assuring Good Style for Object-Oriented Programs”, *IEEE Software*, September 1989, pp. 38-48.

[7] Liskov, Barbara H. , and Zilles, Stephen N., “Programming with Abstract Data Types”, *SIGPLAN Notice*s, 9(4), April 1974, pp 50-59.

[8] Meyer, Bertrand, “Principles of Package Design”, *Communications of the ACM*, 25(7), July 1982, pp. 419-428.

[9] Meyer, Bertrand, *Object-Oriented Software Construction*, 2nd Edition, Prentice Hall, 1997.

[10] Sun Microsystems, The Java Platform 1.1 API Specification, URL: http://java.sun.com/products/jdk/1.1/docs/api/packages.html