OOP and the Janus Principle

Joel C. Adams
Department of Computer Science
Calvin College
Grand Rapids, MI 49546
1 (616) 526-8562
adams@calvin.edu

ABSTRACT

It is easy for computer science students and educators to write software applications in Java or C++ that are not object-oriented. In this paper, we present the *Janus Principle* – a simple software engineering principle (related to the MVC design pattern) whose use produces highly object-oriented code. We demonstrate its effect by developing a simple Java networking application, first without using the Janus Principle, and then using it. Students and educators who follow this principle will write programs containing highly reusable code.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education – *Computer Science Education.*

General Terms

Design, Standardization, Languages, Theory

Keywords

Design Patterns, MVC, Object Oriented Programming, Reusable Code, Software Engineering, User Interfaces.

1. INTRODUCTION

With its emphases on object hierarchies, inheritance, and polymorphism, object-oriented programming (OOP) is a major departure from procedural and modular programming. Computer science (CS) students and educators who learned to program under one of the older approaches face a significant challenge in making the paradigm shift required by the OOP approach (e.g., see [4], [5]).

In this paper, we present a simple software design principle that we have personally found to be quite useful in helping us internalize the practices of OOP. For reasons that will become apparent, we call this the **Janus Principle**. This principle differs from previous, related work (e.g., [2], [4]) by emphasizing support for multiple user-interfaces as a design principle.

2. THE JANUS PRINCIPLE

What we call the **Janus Principle** is conceptually quite simple:

Design and write object-oriented applications
so that they support multiple, reuseable, user interfaces
with minimal redundant coding.

Our principle's namesake was the Roman god of transitions and entryways. (The month of January – the transition from the old year to the new year – is named after Janus.) As shown in Figure 1, he is usually depicted with two (but sometimes more) faces, one looking back to the old, and one looking forward to the new:

Figure 1. Depiction of the Roman God Janus [1]

Janus is a suitable namesake for this principle because (i) the user interface is the "entryway" for a user to a software application; and (ii) just as Janus had two or more *faces*, software written using this principle will support two or more *(inter)faces*.

Following this principle produces software that supports:

- different graphical user interfaces (GUIs) for different platforms (e.g., MacOS, MS-Windows, Linux, …); and
- GUI and command-line interfaces (CLI) on the same platform.

However its real power stems from its "*with minimal redundant coding*" clause. In isolation, this clause leads to the OO mainstays of refactoring, inheritance, and polymorphism. Combined with the rest of the principle, the clause produces an abstraction containing the application's core functionality that is distinct from its user-interface. This abstraction is necessarily reusable by other applications. As we shall see, this principle is strongly related to the well-known Model-View-Controller (MVC) design pattern [7].

In our experience, few CS educators know of this principle. As a result, they neither follow it in the example software they present to students, nor require it of their students on programming projects. Upon hearing it, many CS educators have expressed great appreciation for a single principle that helps them and their students write object-oriented code.

3. NON-JANUS SOFTWARE

Programming in a language that supports OOP (e.g., Java) does not ensure that one is writing OO software. Non-OO programming is especially common in the textbook examples for courses like *Algorithms*, *Operating Systems*, *Network Programming*, and *Parallel Programming*. Such examples frequently use *object-based programming* [3] instead of OOP.

As a simple illustration, consider the implementation of a client for the TCP/IP *daytime service* in Java. A *server* for the daytime service listens to port 13 for client connections and, upon receiving one, sends that client the current date and time. A daytime *client's* behavior is also fairly simple:

1. Get the name of a daytime server *s* from the user
2. Open a TCP connection *c* to port 13 on *s*
3. Read the daytime server's time *t* via *c*
4. Display *t* for the user
5. Close *c*

Figure 2 presents a typical Java daytime client implementation (i.e., without the Janus Principle). This particular client gets the name of the daytime server from the command-line:

```
// TypicalDaytimeClient.java
import java.net.*;    // Socket, InetAddress, ...
import java.io.*;     // BufferedReader, ...

class TypicalDaytimeClient {
 public static void main(String [] args) {
   if (args.length != 1) {
      System.err.println("Usage: java " +
            "TypicalDaytimeClient <server>");
      System.exit(1);
   }

   try {
      // get name of time-server from user
      String host = args[0];
      // open connection to port 13 on it
      final int PORT = 13;
      InetAddress addr =
          InetAddress.getByName(host);
      Socket sock = new Socket(addr, PORT);
      BufferedReader sockIn =
         new BufferedReader(
          new InputStreamReader(
               sock.getInputStream() ));

      // get time from time-server
      String response = "",
             temp = null;
      while (true) {
         temp = sockIn.readLine();;
         if (temp == null) break;
         response += temp;
      }
      // display time
      System.out.println(response);
      // close our end of the connection
      sock.close();
   } catch (IOException ioe) {
         e.printStackTrace();
   }
 }
}
```

Figure 2. An Object-Based CLI Daytime Client

Figure 3 shows an example execution of the program using the U.S. government server *time.nist.gov* in Boulder Colorado:

```
$ java TypicalDaytimeClient time.nist.gov
53601 05-08-19 14:03:10 50 0 0 853.2 UTC(NIST) *
```

Figure 3. Running the Daytime Client

While the code in Figure 2 solves the problem, it represents an object-based solution, not an object-oriented solution. It is in essence the same solution one would write in Fortran or C if those languages provided String, Socket, and so on as built-in types.

Put differently, none of the code in this solution can be reused by another application. If a different application needs to use the daytime service, that application must replicate the code to build a connection to the server, read from it, and close the connection, wasting time, effort, and space. More than 50% of the code in Figure 2 would be replicated in the new application.

If we present solely these kinds of examples to our students in our advanced courses, our students will imitate our examples.

4. THE MVC PATTERN

In Section 2, we mentioned that the Janus Principle is strongly related to the MVC design pattern. Before we apply this principle to the problem of building a daytime client, we first review the MVC pattern, and see how it relates to our principle.

4.1 MVC with One User Interface

The Model-View-Controller (MVC) design pattern originated as a foundation of the Smalltalk-80 system [7], which served as the inspiration for operating systems like Apple's MacOS and Microsoft Windows. More recently, MVC has been used as the design pattern for the widgets in Java's Swing package [8].

While these references give MVC a strong historical pedigree, its importance lies in its separation of an application's state information or *Model* from its user interface or *View*. The third piece of the pattern – the *Controller* – manages communication between the Model and View, specifying what happens in each in response to the user's interaction/behavior.

Because an MVC Model is independent of its user interface, it can be used by different Views (e.g., one View for MacOS, another for Windows, and another for Unix systems). The MVC pattern thus encourages us to write reusable Model code.

In a "pure" MVC design, each piece of the pattern is implemented as a separate class. These can be related to one another as shown in the classic UML diagram of Figure 4:

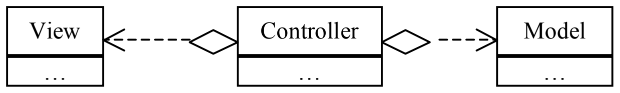


Figure 4. UML Diagram: The MVC Pattern

As depicted in Figure 4, the Model has no linkage to the View, and the View has no linkage to the Model. However the Controller has an aggregation dependency linkage to each.

4.2 MVC with Multiple User Interfaces

Decoupling of the View from the Model allows us to build multiple interfaces for an application, as shown in Figure 5:

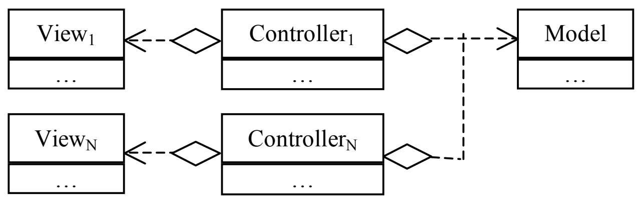


Figure 5. MVC with Multiple Views

Because there is a 1-to-1 correspondence between Views and Controllers, it is common (at least in Java) to combine these into a single class, yielding the diagram shown in Figure 6:

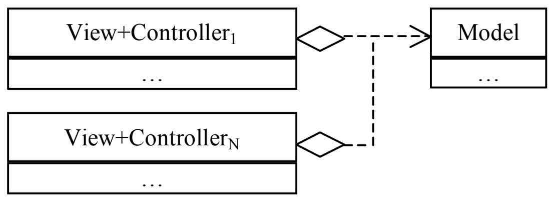


Figure 6. A Simplified Multiple-View MVC

In our experience, most students will acknowledge that such separation is beneficial in theory, but they will not make it their practice unless they personally experience its benefits. To do so, we give them programming assignments that require them to build multiple, reuseable user interfaces for their applications, without redundant coding (i.e., require them to apply the Janus Principle in software projects). For many assignments, students can use a UML diagram based on the one in Figure 6.

The diagram in Figure 6 can be generalized for applications in which the View and Controller need to be separated, and different Controllers inherit common code from a superclass. This more-general diagram is available for download via the world wide web from *http://www.calvin.edu/~adams/software/janus/*.

5. JANUS-PRINCIPLE SOFTWARE

To demonstrate the use of the Janus Principle, we now show how it might be used in a network programming course to implement a multi-interface daytime client. In the code examples that follow, we omit most of the documentation, to save space.

5.1 The Model

Earlier, we saw that the basic steps a daytime client must take are:

1. Get the name of a daytime server *s* from the user
2. Open a TCP connection *c* to port 13 on *s*
3. Read the daytime server's time *t* via *c*
4. Display *t* for the user
5. Close *c*

We begin by separating the responsibilities of the Model from those of the View. Steps 1 and 4 are interactions with the user, so they are the responsibility of the View+Controller. This leaves steps 2, 3, and 5 – the core functionalities of the application – as the responsibility of the Model. We therefore build a *DaytimeClient* class that provides these latter functionalities, as shown in Figure 7:

```
// DaytimeClient.java
import java.net.*;     // Socket, InetAddress
import java.io.*;      // BufferedReader, ...

class DaytimeClient {
  private final int PORT = 13;
  private Socket mySocket;
  private BufferedReader myReader;

  // build connection to time-server
  public DaytimeClient(String host) {
    try {
      InetAddress addr =
                      InetAddress.getByName(host);
      mySocket = new Socket(addr, PORT);
      myReader = new BufferedReader(
                   new InputStreamReader(
                     mySocket.getInputStream() ));
    }
    catch(Exception e) { e.printStackTrace(); }
  }

  // get response from time-server
  public String receive() {
   String result = "";
   try {
      String temp = null;
      while (true) {
         temp = myReader.readLine();
         if (temp == null) break;
         result += temp;
      }
   }
   catch (Exception e) { e.printStackTrace(); }
   return result;
  }

  // close our end of the connection
  public void finalize() {
   try { mySocket.close(); }
   catch (Exception e) { e.printStackTrace(); }
  }
}
```

Figure 7. The Model for a Daytime Client

Our class constructor provides the functionality of step 2; our `receive()` method provides the functionality of step 3; and our `finalize()` method provides the functionality of step 5.

It is worth mentioning that our `finalize()` method overrides the definition of `finalize()` from Java's `Object` class. Java's garbage collector automatically invokes an object's `finalize()` method just before it reclaims the storage of a `DaytimeClient` object. Thanks to this auto-invocation, our View+Controller classes will not need to explicitly invoke `finalize()`. They will (of course) need to construct an instance of `DaytimeClient` and send that instance the `receive()` message (see below).

5.2 A Command-Line View+Controller

As described earlier, steps 1 and 4 are the responsibility of a daytime client's View. In Figure 8, we present a View class containing methods for performing these steps via the command-line, plus a `main()` method that acts as a Controller:

```
class DaytimeCLI{
 private String [] args = null;

 public DaytimeCLI(String [] args) {
   if (args.length != 1) {
     System.err.println("java DaytimeCLI <host>");
     System.exit(1);
   }
   this.args = args;
 }

 public String getServer() { return args[0]; }

 public void display(String daytime) {
   System.out.println(daytime);
 }

 public static void main(String [] args) {
  DaytimeCLI view = new DaytimeCLI(args);
  String server = view.getServer();
  DaytimeClient model = new DaytimeClient(server);
  view.display( model.receive() );
 }
}
```

Figure 8. A Command-Line Daytime View+Controller

This client is functionally equivalent to that shown in Figure 2, but its core functionality now resides in a separate `DaytimeClient` Model, its View lies in a reusable `DaytimeCLI` class, and its controller lies in that class's `main()` method.

This View+Controller is invoked in a manner like that of Figure 3, and its output is similarly equivalent, as shown in Figure 9:

```
$ java DaytimeCLI time.nist.gov
53601 05-08-19 15:54:13 50 0 0 799.7 UTC(NIST) *
```

Figure 9. Running DaytimeCLI

This approach exhibits greater modularity than the approach shown in Figure 2: (i) the application's core functionality now resides in one class (`DaytimeClient`), (ii) its View now resides in another class (`DaytimeCLI`), and (iii) both of these are reusable by other applications. For example, any application that needs to read the day and time from a remote system can create an instance of the `DaytimeClient` class to do so.

5.3 A Menu-Driven View+Controller

Being able to enter the name of the daytime-server on the command-line is convenient for testing our `DaytimeClient` Model, but it isn't a very user-friendly approach. In particular, it requires the user to know the name of a daytime server. While such names are fairly easy to find on the Internet (e.g., see [6]), a more user-friendly View+Controller might use the console to present a menu of choices from which the user can choose a time-server. Because our client's core functionality is encapsulated in our `DaytimeClient` class, building such a View+Controller is straightforward, as shown in Figure 10:

```
import java.io.*;    // BufferedReader, ...

class DaytimeConsole {
 private String MENU = null;
 private BufferedReader kbd = null;

 public DaytimeConsole() {
   MENU = "Choose a daytime server...\n\n" +
          "Enter:\n" +
           "\t1 for time.nist.gov (CO)\n" +
           "\t2 for time-a.nist.gov (MD)\n" +
           "\t3 for time-nw.nist.gov (WA)\n" +
           "\t4 to enter a different server\n" +
           "--> ";
   kbd = new BufferedReader(
            new InputStreamReader( System.in ) );
 }

 public String getServer() {
   String server = null;
   System.out.print(MENU);
   try {
    int item = Integer.parseInt( kbd.readLine() );
    switch (item) {
      case 1:
       server = "time.nist.gov";
       break;
      case 2:
       server = "time-a.nist.gov";
       break;
      case 3:
       server = "time-nw.nist.gov";
       break;
      case 4:
       System.out.print("Enter server name: ");
       server = kbd.readLine();
       break;
      default:
       System.err.println(item + " is invalid");
       System.exit(1);
    }
   } catch(IOException e) { e.printStackTrace(); }

   return server;
  }

 public void display(String daytime) {
   System.out.println(daytime);
 }

 public static void main(String [] args) {
  DaytimeConsole view = new DaytimeConsole();
  String server = view.getServer();
  DaytimeClient model = new DaytimeClient(server);
  view.display( model.receive() );
 }
}
```

Figure 10. A Menu-Driven Daytime View+Controller

Unlike Figure 8, the `getServer()` method in Figure 10 performs step 1 by displaying a menu from which the user chooses a server.

As in Figure 8, the `main`() method in Figure 10 serves as this View's Controller, coordinating the interaction between this View and the Model. The `main()` and `display()` methods of Figures 8 and 10 happen to be identical, though this need not be the case (e.g., in a GUI View). While space limits prevent us from doing so here, this common code can be refactored into (and then inherited from) a superclass, to minimize redundant coding.

When executed, the View+Controller in Figure 10 generates a very different user-interface from that shown in Figure 9, though its output is equivalent. Figure 11 shows a sample execution:

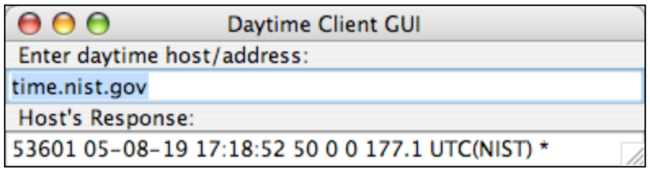
```
$ java DaytimeConsole

Choose a daytime server...

Enter:
        1 for time.nist.gov (CO)
        2 for time-a.nist.gov (MD)
        3 for time-nw.nist.gov (WA)
        4 to enter a different server
--> 3
53601 05-08-19 16:29:31 50 0 0 615.7 UTC(NIST) *
```

Figure 11. Running DaytimeConsole

The Janus Principle thus makes it relatively easy to write software that offers different user-interfaces for different kinds of users.

5.4 A GUI View+Controller

Using the same Model, we can also build a graphical user-interface for the daytime service. Figure 12 shows one in action:



53601 05-08-19 17:18:52 50 0 0 177.1 UTC(NIST) *

Figure 12. Running DaytimeGUI

Space limits prevent us from presenting its source code, but it and the other programs in this paper are available via the world-wide-web from *http://www.calvin.edu/~adams/software/janus/*.

6. DISCUSSION

We have presented a modest application to illustrate the use of the Janus Principle. In a more complex application, the principle's *"with minimal redundant coding"* clause will motivate the use of refactoring, inheritance and polymorphism. We deliberately chose our application for (we hope) clarity, and to show how the Janus Principle is beneficial even for a simple application.

We have successfully used this principle in courses ranging from *Operating Systems* to *Network Programming* to *High Performance Computing* – courses in which OO software engineering practices are frequently neglected. When presenting a new topic, (e.g., client-server networking), we usually begin with an object-based example like that shown in Figure 2. By doing so, the students' first exposure to the topic is in one self-contained file, helping them get an overview of that topic.

Once the students have seen an object-based version and any questions have been addressed, we immediately present an alternative Janus-Principle version, to illustrate and reinforce the importance of OO software-engineering.

We then typically assign a 1-week project in which the students gain hands-on experience with the topic. The students are not required to use the Janus Principle on this project. In our experience, most will build object-based applications because the examples in their textbooks are usually object-based.

Following this 1-week project, we assign a short (3-day) mini-project in which the students must add a second user-interface to their project, using the Janus Principle. Since the students have already completed one user-interface, this mini-project amounts to refactoring the Model from their first project, revising its user-interface slightly to form one View+Controller, and then building a second View+Controller for their second user-interface.

If this "project plus mini-project" approach is used consistently in a course, the students soon learn to "short-circuit" the process and design their projects according to the Janus Principle at the outset, so that they have less work to do on the mini-project.

If this approach is used consistently throughout a CS curriculum, most students will eventually internalize it and begin to design reusable software on their own.

7. CONCLUSIONS

If students see only non-OO examples in our courses, we should not be surprised if they write non-OO programs for their projects. The Janus Principle is a simple guideline that can help both instructors and students write better programs.

Applications written using this principle will support multiple, reusable user interfaces with minimal redundant coding. Designing applications in accordance with the principle requires a level of abstraction that produces highly object-oriented code, *even if we do not actually build more than one user interface*.

By following this principle as instructors and "encouraging" our students to do so, they will directly experience the benefits of OOP, helping them internalize its practice and methodology.

8. REFERENCES

[1] Alciato's Book of Emblems, Used by permission. Online: http://www.mun.ca/alciato/.

[2] E. Arif, A Methodology for Teaching Object-Oriented Programming Concepts in an Advanced Programming Course, *ACM SIGCSE Bulletin* (32) 2, June 2000, 30-34.

[3] O. Astrachan, Using Classes Early, An Object-Based Approach to Using C++ in Introductory Courses, *Proceedings of the 29th SIGCSE*, Feb 1998, 383-387.

[4] J. Bergin and R. Winder, Understanding Object-Oriented Programming, *ACM SIGPLAN Notices* (37) 6, June 2002, 18-25.

[5] R. Decker and S. Hirshfield, The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught In CS-1, *Papers of the 25th SIGCSE*, March 1994, 51-55.

[6] T. O'Brian, *NIST Internet Time Servers*. Online: http://tf.nist.gov/service/time-servers.html, August 20, 2005.

[7] L. Pinson & R. Wiener, *An Introduction to Object-Oriented Programming and Smalltalk*, Addison-Wesley, 1988, 336-358.

[8] T. Sunsted, MVC meets Swing, *JavaWorld*, April 1998, Online: http:// www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html