Unchained Malady

James W. Cooper

toc: Here's a simple way to expand the number of tests you make on a set of values without writing spaghetti code.

deck: The Chain of Responsibility pattern can help you write code that is easier to modify.

Some years ago, I became interested in the technology of data handling for competitive swimming, and as a hobby developed a set of programs for running swim meets and managing swim team records. [A hobby is a business (labsoftware.com) that doesn't make any money.]

Recently I was presented with a problem in these programs that was difficult to solve, and it got me to thinking about how Design Patterns might have helped me out of this tangle. The current software is the sixth generation of a simple program written in Basic, and it is written in Visual Basic. VB has some OO features but as a language system, it doesn't particularly encourage or support OO approaches.

This is another way of saying "if I only knew then what I know now," which is a common programmer's complaint in all languages. You learn a lot in writing a complete system and you could always do this better if you had the time to rewrite the system based on what you've learned. In this case, we'll assume that my learning includes both design patterns and the fact that Java might be a superior way to tackle this system.

A Tangle of Times

In order to solve this problem, we have to define it carefully. End of season championship swim meets frequently have qualifying times. In other words, you have to have gone this fast in this distance and stroke to enter the meet. For youngsters, these cut off times are further defined by age group as well as by sex. So the problem of selecting swimmers from your team's roster who qualify for an event would seem to be quite simple: for each event, look for swimmers who have times faster than the established standard and enter them in the meet. But, of course it is more complicated than that. In the next three paragraphs, I'm going to give you a simplified explanation of the problem. The actual problem is worse, but the simplified version is enough to get us back to OO programming sooner.

In the US there are "short course" and "long course" pools. The shorter pools are 25 *yards*, and the longer "Olympic-size" pools are 50 *meters*. It is not possible to convert times achieved in these two courses because there are a different number of turns in a race and there simply is no analytic algorithm for such a conversion. Instead, qualifying times are usually posted for both courses. So the entry algorithm is: If the swimmer has a time in the course the meet is to be swum in, use that. Otherwise, check to see if they have a qualifying time in the other (nonconforming) course, and use that.

Now consider the fact that swimmers have to qualify somewhere. So qualifying meets are held for swimmers who have not achieved this standard yet. If a swimmer has a time *slower* than the standard, enter them in the qualifying meet. Since this could cut too wide a swath, a more common case is to enter the kid in the qualifying meet if his time is slower than the championship meet standard, but faster than some arbitrary slower time that is used to keep out the rank beginners. So now, we have an algorithm that says: if the swimmer has a time slower than the standard but faster than some slower standard, enter them in the qualifying meet.

Now, this final fillip is where it gets tricky. Supposing a swimmer has a qualifying time in the non-conforming course but not in the conforming course. That means he can legitimately enter the championship meet. But suppose he wants to compete to obtain a time in the conforming course. Should this be allowed? The organization decided that this should not be allowed, making the decision of who is legitimately entered that much more difficult. It was this change in policy that made me throw up my hands and say it was too hard to change the code quickly, and it would lead to a spaghetti code of if-statements that I couldn't guarantee would work soon enough for the now just-ended season.

Trying A New Pattern

Now if I had used better OO principles and an appropriate design pattern, would I have been better off? Emphatically yes. Let me briefly outline the Chain of Responsibility pattern. You use this pattern when you want several classes to attempt to handle some kind of request without any of them having knowledge of the other classes or their capabilities.

For example, in a Help system you might have specific help on one visual control, more general help for a group of related controls and even more general help for the entire window. You start with help at the most local level and search upward until you find a help message that has been implemented. The way to do this is to have a series of classes linked together in a chain, with each one forwarding to the next if it can't satisfy the request.

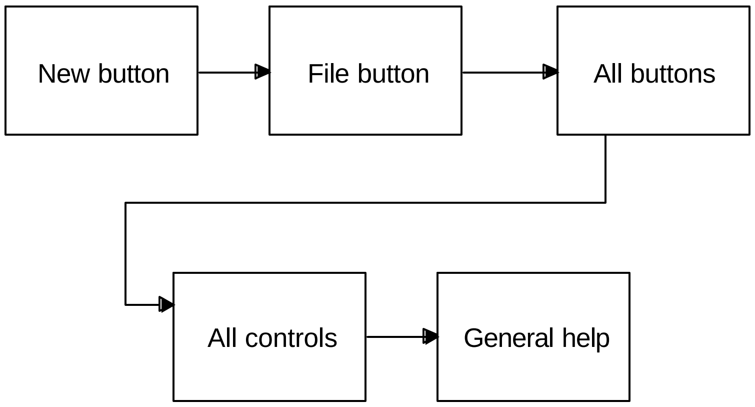

Figure 1- A model of a Help system using the Chain of Responsibility pattern,

Figure 1 shows a Chain of Responsibility for a Help system for a simple user interface. The interface has a File button and a New button. If you ask for help, it the control receiving the help request sends it to the chain until one item in the chain can satisfy the request and display the appropriate level of help. Thus, if you have not written help for every element, the next more general help is shown instead.

Additionally, the Chain of Responsibility pattern helps keep separate the knowledge of what each object in a program can do. Objects don't need to know about other related objects, and the each can act independently.

Chains of Responsibility also are frequently used in compilers and interpreters, where you recognize language tokens in a group on a stack and then send the stack frame pointer to a chain until one can act on it. They probably have lots of other uses I haven't yet come across, but it occurred to me that this swimming cutoff time problem is one that is amenable to this pattern.
In both the Help system example and the compiler-interpreter example, the request is passed along the chain of objects, each one examines the request and acts on it if it can. Otherwise, it sends it on along the chain. Another way to build such a system is to have each object act on the request and *pass it along as well.* This would mean that each object could further modify a decision made by a previous object without knowing whether it was the first, last, or only object in the chain.

Swimmers in Chains

Now let's actually write some Java. We'll start by defining an interface for a Chain:

```
//The basic Chain interface
interface Chain {
    //add an element to the chain
    public void addChain(Chain c);
    //get the next object in the chain
    public Chain getChain();
    //send data requests along the chain
    public void sendToChain(Swimmer swmr, Event evnt);
}
```

For simplicity, we'll define a Swimmer object as one who has a single time in each of two courses, and ignore the complication that there are multiple strokes and distances. Our Swimmer will just have a name, two times (conforming course and nonconforming course) and a flag that indicates whether he is eligible to swim in this meet.

```
//a simple Swimmer object
public class Swimmer {
    //name
    private String firstName, lastName;
    //whether eligible to enter event
    private boolean eligible;
    //times in conforming nad nonconforming course
    private float stime, nctime;

    public Swimmer(String frname, String lname) {
      firstName = frname;
      lastName = lname;
    }
    //set the times
    public void setTimes(float time, float ncTime) {
        stime = time;
        nctime =ncTime;
    }
    //get the times
    public float getTime() {
        return stime;
    }
    public float getNcTime() {
        return nctime;
    }
    //set whetehr eligible
    public void setEligible(boolean b) {
        eligible = b;
    }
    public boolean getEligible() {
        return eligible;
    }
    //get the name
    public String getName() {
        return firstName+" "+lastName;
    }
}
```

Likewise, we’ll define an event as having a stroke and distance and fast and slow cutoff times for both the conforming and nonconforming course:

```
//an Event object
public class Event {
    private int eventNumber;
    private String strokeName;
    private int distance;
    private float slowCut, fastCut;
    private float ncSlowCut, ncFastCut;
    //save event number, distance and strokt
    public Event(int number, int dist, String stroke) {
        eventNumber = number;
        distance = dist;
        strokeName = stroke;
    }
    //save the slow cuts
    public void setSlowCuts(float slow, float ncSlow) {
        slowCut = slow;
        ncSlowCut = ncSlow;
    }
    //save the fast cuts
    public void setFastCuts(float fast, float ncFast) {
        fastCut = fast;
        ncFastCut = ncFast;
    }
    //return the cut you ask for
    public float getSlowCut() {
        return slowCut;
    }
    public float getFastCut() {
        return fastCut;
    }
    public float getNcSlowCut() {
        return ncSlowCut;
    }
    public float getNcFastCut() {
        return ncFastCut;
    }
}
```

Building Your Own Timing Chain

Now what about this chain? The crucial technical breakthrough is realizing that rather than setting an eligible/ineligible Boolean in a set of if-statements, it is much better to keep that flag inside the Swimmer object. Then each chain element can set the flag to eligible/no eligible depending on the result of its test. And each element thus needs to make only one test. Rather than writing a bunch of complicated tests, you string together a set of simple tests in a chain. You just have to make sure that the chain goes from least to most restrictive. Here's a basic chain element that tests for a time faster than the slow cut:

```
//Check a swimmer's time in a chain
public class TimeChain implements Chain {
    protected Chain chain;
    public TimeChain() {
        chain = null;
    }
    public void addChain(Chain c) {
        chain = c;
    }
    public Chain getChain() {
        return chain;
    }
    //check to see if swimmer's time is faster than slow cut
    public void sendToChain(Swimmer swmr, Event evnt) {
        if (swmr.getTime () <= evnt.getSlowCut ())
            swmr.setEligible (true);
        else
            swmr.setEligible (false);
        System.out.println("TimeChain:"+swmr.getEligible ());
        sendChain(swmr, evnt);
    }
    //test for null and send to next chain element
    protected void sendChain(Swimmer swmr, Event evnt) {
        if( chain != null)
            chain.sendToChain (swmr, evnt);
    }
}
```

We can derive the rest of the chain elements from this one, so the amount we have to rewrite is very small indeed. Here is the one that tests for a time slower than the fast cut:

```
public class sTimeChain extends TimeChain {
    //check to see if swimmer's time is slower than fast cut
    public void sendToChain(Swimmer swmr, Event evnt) {
     if(swmr.getTime () > evnt.getFastCut () )
         swmr.setEligible (true);
     else
         swmr.setEligible (false);
     System.out.println("sTimeChain:"+swmr.getEligible ());

    sendChain (swmr,evnt);  //send along chain
    }
}
```

Overall here, we create 4 little classes for the four tests we've described and connect them together in a chain. Here is the code that sets this all up. The times were taken from standards for 11-12 girls 100-yard breaststroke at a qualifying meet.

```
public class ChainSwim {
    public ChainSwim() {
        //set up the event
        Event evnt = new Event(2,100,"Breast");
        evnt.setFastCuts (121.0f, 136.5f);
        evnt.setSlowCuts (132.99f,149.49f);
        //set up a swimmer
        Swimmer Evelyn = new Swimmer("Evelyn", "Earnest");
        Evelyn.setTimes (123.5f, 131.0f);
        //set up the timing chain
        TimeChain timeChn = new TimeChain();
        sTimeChain stChn = new sTimeChain();
        timeChn.addChain (stChn);
        ncTime nctChn = new ncTime();
        stChn.addChain (nctChn);
        sNcTime snChn = new sNcTime();
        nctChn.addChain (snChn);

        //begin the tests
        timeChn.sendToChain (Evelyn, evnt);

        //and print out the result
        System.out.print(Evelyn.getName ()+" is ");
        if(! Evelyn.getEligible ())
            System.out.print("not ");
        System.out.println("eligible");
    }
    static public void main(String argv[]) {
        new ChainSwim();
    }
}
```

We defined this swimmer so she qualified for the meet based on her conforming times but not based on her non-conforming times. We designed each of the timing chain classes so it prints out its decision, so we can monitor the decisions it makes, in case we got one wrong. Here are the results:

```
TimeChain:true
sTimeChain:true
ncTime:true
sNcTime:false
Evelyn Earnest is not eligible
```

As you can see, it found her eligible until the very last test.

Advantages of Chains

One great advantage to taking these tests apart and putting them in a chain of objects is that you can add more at any time. I alluded to further complexity earlier. Some of that lies in the fact that even in the US and Canada there are really 3 courses to consider, the other being 25- *meter* pools. In the UK they recognize a wide variety of other pool lengths based on the fact that there are a wider variety of pools in the UK. This system is adaptable to more courses and more distances very easily.

Diagramming Our Chains

It is sometimes instructive to see that this is really quite a simple and extensible system by looking at its UML diagram, as we show in Figure 2.

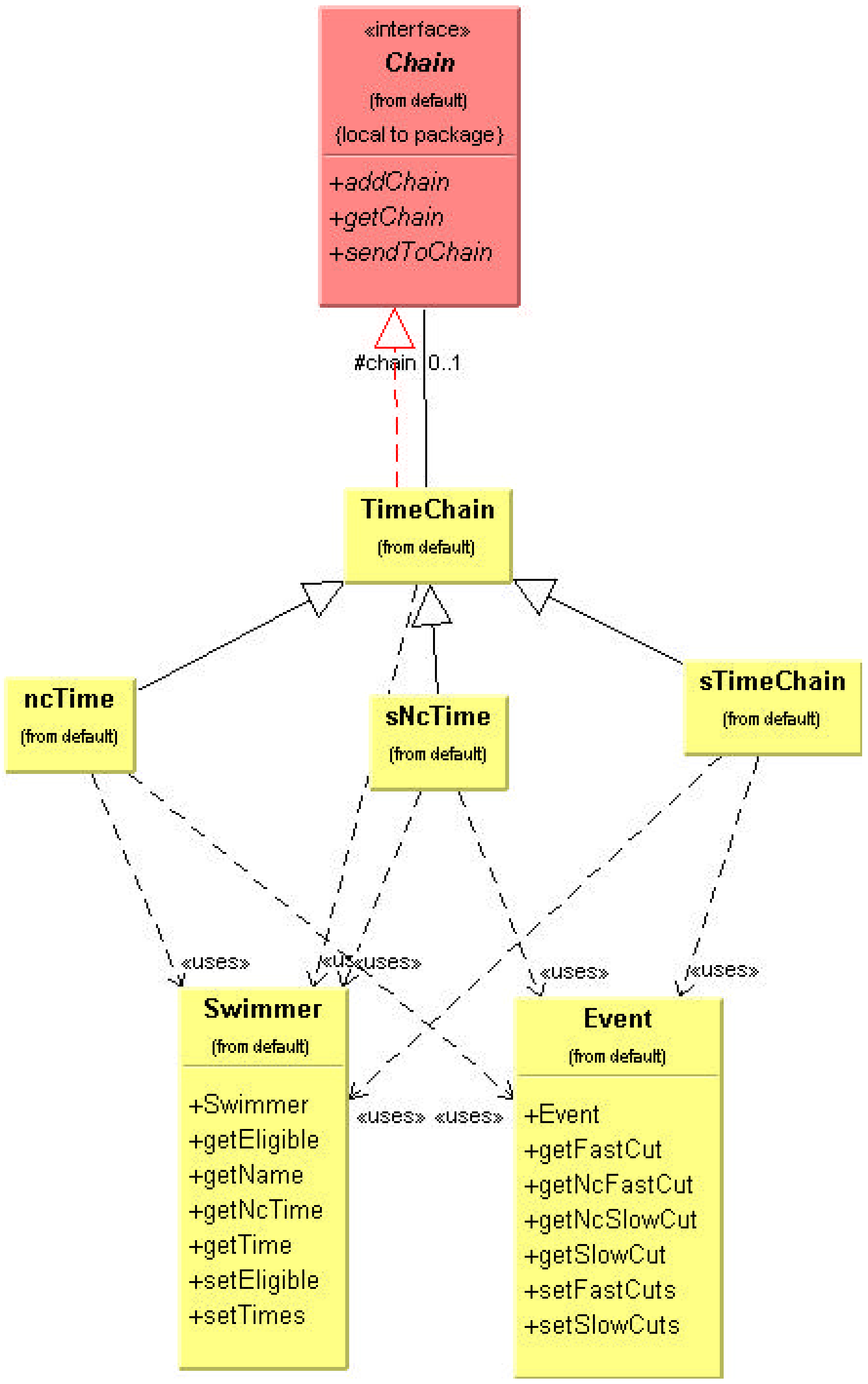

Figure 2- The UML diagram of our timing chain of responsibility.

To review, we start with a Swimmer and an Event object, and a Chain interface. We implement the Chain interface in TimingChain, and then derive the remaining classes from it.

Concluding the Chain

We've seen here that the Chain of Responsibility takes a set of decisions and puts on in each object and passes the data long so each object can work on it. The objects can then decide to pass it on or not based on what they are to accomplish. In a real-life situation you also need to decide what to do when the data does not match any of your preconceived notions. You can fail silently, pass the data to an error handler chain object or take some default action. Any of these are possible, based on the last item in the chain. In any case, it's a far simpler and more flexible system than a page of if-else tests and flags.

James W. Cooper is the author 13 books, most recent of Java Design Patterns: A Tutorial, published by Addison-Wesley in 200. He is also the president of Lab Software Associates, which loses money supplying swimming software.