Can We Manage Programming Knowledge?

James W. Cooper

Toc: Refactoring is probably the key to describing how programmers work.

Deck: Whenever you take a program apart and rearrange its objects, you are engaged in refactoring.

Years ago, a friend of mine was playing clarinet in a dance band. You know, the kind of band that uses real printed music. On the charts for the number he was reading with the band was the notation "Tacet if soloist." In other words, if there was a soloist, he was not supposed to play. Well there was, and he missed it and played along anyway. The trumpet player behind him nudged him and said, "Go ahead man, it says 'take it.'"

Everyone talks about knowledge management. Even those who know better. It was the buzzword of the year a couple of years back and hasn't died off yet. It is rife with slogans like "If only we knew what we know…" and other such monuments to utter obviousness. And yet, it does recognize a problem for large organizations as well as for smaller ones, such as volunteer or ad hoc organizations with rapid turnover. How can we capture the stuff that "everyone knows" and is implicit in the fact that the organization exists at all. How do you get a new member enculturated to all that you have already learned?

Knowledge management gurus like to talk about "tacit knowledge" to describe this problem. Tacit knowledge is that quiet knowledge that isn't played out loud but is central to how the organization got to where it is. A lot of customs and procedures are basically tacit knowledge.

I was thinking the other day about knowledge management in programming. Too many times, our programming works like this. We outline a program design and then we write it. It will probably be a program consisting of a number of objects and if we spend a little time on the design, there may well be some design patterns in it as well. We certainly will worry about how the objects are organized and how they interact.

When we finish the first version of the program, it is usually pretty well designed, and does whatever the version 1.0 version of the job we set out to do. But, over time, we begin making changes. We add features, incorporate code written by others, and connect the objects in unexpected and unplanned ways. We derive classes, add methods, and sometimes have to put the same method in more than one class, because it is too hard to rewrite to avoid this.

After a while, say 2, or 3 or 8 versions, we still have a working program, but it has gotten much harder to change and maintain. In fact, over time, we find that we can't change the program further without excessive effort, and we finally retire it and rewrite the code completely. This is the life cycle of programming. Start with a design, build it, change it, and keep changing it until we can sin no more. Then we either rewrite it or seek a new job assignment.

The question that occurs to me is: what about the knowledge that we accrue during this development and revision process? Is there any way to capture it? If we had that knowledge to draw on, would we write more flexible programs in the first place, and

would we plan for change and growth more effectively? Is there or can there be this sort of programming knowledge management? I don't know. I assume there may be some research projects in this area, but I haven't discovered any that have come to sufficient fruition yet.

Refactoring

Whenever you take a program apart and rearrange its objects, methods or logic, you are engaged in a process commonly called *refactoring*. Refactoring is the $5 name for the 5 cent process we undertake almost every day in programming. There is one major book in this area, by Martin Fowler, and it outlines most of the common techniques programmers use to rearrange code to make it more effective. While some feel that this material is kind of obvious, I found that reading through the book made a lot of things clearer. Fowler organizes this material well and gives a lot of useful examples in Java. He arranges the refactoring procedures into patterns, much like the popular Design Patterns. While these are somewhat easier to grasp than Design Patterns, they are more crucial, because you really do use them all the time. And finally, refactoring techniques are the "knowledge" we've been trying to encapsulate. Here in one book is indeed much of what we actually do every day.

In order to refactor a program, it is important to have a test case you can run both before and after you have rearranged the code. You don't know whether your refactoring is effective if you can't prove the code still gets the right answer to a test case. In addition, Fowler notes, refactoring can help you find bugs in your program. Since you have to really understand how a class works to refactor it, you take a harder look at the code and understand it more deeply. Frequently he notes (and I have also found) that when I refactor a piece of code, I not only make it easier to understand, I make it more correct. In fact, refactoring can make the program easier to understand and in the process you will often improve the design.

When do you refactor a program? Fowler suggests several red flags that should cause your refactoring alarm to go off. Here are a few of them.

1. **Duplicated code.** Refactor when you find yourself duplicating code in several classes. Recently I was adding some test code to a set of classes representing different database tables, so I could instrument the database construction process and find out where time was being consumed. I needed a way to print out elapsed time.

2. **Large Class –** If you have written a very large class, usually by accretion or a very long method in a class, you have an obvious candidate for refactoring.

3. **Long Parameter List** – If some methods in your class have a long parameter list, you definitely need to simplify your code by refactoring. You can pass objects that contain a set of values, keep some persistent data in the class and use get and set methods to reduce this dependency. Long parameter lists are a holdover from on-OO languages like FORTRAN, and really aren't needed in OO languages like Java.

4. **Switch Statements** – If you find any method in your class using a switch statement, be very suspicious. You should probably be able to refactor your code

using polymorphism so that separate methods or classes handle each data type. As Fowler notes so succinctly, polymorphism is another $5 word defining a common OO technique that allows you to avoid writing conditionals when you have objects whose behavior depends on their types.

5. **Shotgun Surgery** – If whenever you make a change in once place, you have to make it in several others, you are engaging in what Fowler calls "shotgun surgery." This is a surefire indication that you need to do some refactoring. Changes in one place in an OO program should not force changes in many other places. Usually, you need to move data or move methods between classes to clear this up.

Lets take a couple of these refactoring ideas and illustrate them in the sections that follow.

Eliminating Duplicated Code

In Java 2, computing time is a little elaborate, because most of the time methods of the Date class have been deprecated. Instead, you create instances of the GregorianCalendar class.

```
GregorianCalendar cal = new GregorianCalendar();
```
You can get a Date object containing that time instant from the calendar :

```
Date dt = cal.getTime();
```
And you can get the elapsed time in milliseconds since 1970 from the Date:

```
long time = dt.getTime();
```

So, to print out an elapsed time around some operation, you do the following:

```
//get the start time
 GregorianCalendar cal = new GregorianCalendar();
 long start = cal.getTime().getTime ();
 doSomething();
//get the finish time
cal = new GregorianCalendar();
 long end = cal.getTime().getTime();
 float interval = (end - start)/1000.0f;
 System.out.println("time=" + interval);
```

That's fine, if you only do it once, but if you have several classes where you'd like to time some method, as I did, this is enough code that duplicating it seems wasteful. So, instead, you should refactor the code to create a little timing class:

```
//class to time events
public class Timeit {
    private GregorianCalendar cal;
    private long start;

    public Timeit() {
        start = getNow();    //save the starting time
    }
    //get the current time
    private long getNow() {
        cal = new GregorianCalendar();
        return cal.getTime().getTime ();
```

```
    }
    //compute the elapsed time
    public float getElapsed(){
        long end = getNow();
        return (end - start)/1000.0f;
    }
}
```

Then our timing code amounts to

```
        Timeit tf = new Timeit();
        doSomething();
        System.out.println("this time=" + tf.getElapsed ());
```

This is about the minimum change necessary to get timing of a method call.

Eliminating Long Argument Lists

Suppose we have a list of data representing names and addresses. We could read in the
name and address fields as tokens using a tokenizer and just print them out. Here are a
few such names"

```
Sam | Spade | 123 Glenview | Stamford | Ct | 06840
Sally | Spunk | 456 W N Broadway | Columbus | OH | 43202
Fred | Fink | 374 Walhalla Rd | Columbus | OH |43202
Chris | Cellier | 14 Spotswood Lane | Redding | CT | 06890
```

The simple-minded code for this would be to read in each line, put it into a string
tokenizer and then print it out.

```
String fname = tok.nextToken();
String lname = tok.nextToken();
// etc.
printAddress(fname, lname, address, city, state, zip);
```

This is a classic example of long argument strings, although it could clearly be worse.
Instead, lets create some objects. We'll make a Person, an Address, a City and a
PersonLoc. One reason we might make this PersonLoc class separate is to allow for the
case where a single person has two or more addresses, such as a regular home and
vacation home.

Our Person class is just

```
public class Person {
    private String frname, lname;
    public Person(String fr_name, String l_name) {
        frname = fr_name;
        lname= l_name;
    }
    public String getFrname() {
        return frname;
    }
    public String getLname() {
        return lname;
    }
}
```

and our Address class merely

```
public class Address {
    private String street;
    private City city;
    public Address(String _street) {
        street = _street;
        }
    public void setCity(City cty) {
        city = cty;
    }
    public City getCity() {
        return city;
    }
    public String getStreet() {
        return street;
    }
}
```

We make the City object separate, as

```
public class City {
    private String cityName, stateName, zipCode;
    public City(String cname, String state, String zip) {
        cityName = cname;
        stateName = state;
        zipCode = zip;
    }
    public String getName() {
        return cityName;
    }
    public String getState() {
        return stateName;
    }
    public String getZip() {
        return zipCode;
    }
}
```

The reason for this is that there may be many people in the same city and each does not need a separate City object. Instead we keep only the city objects that differ. So we create a CityFactory class that returns a city object with that zip code if it exists or creates a new one if it does not:

```
public class CityFactory {
    private HashMap cities;
    public CityFactory() {
        cities = new HashMap();
    }
    public City addCity(String cname, String state, String zip) {
        City city = null;
        if(cities.containsKey (zip)) {
            city =(City)cities.get (zip);
        }
        else {
         city = new City(cname, state, zip);
         cities.put(zip, city);    //replace any other
        }
        return city;
    }
}
```

This CityFactory class is really also an example of the Flyweight pattern, where we create a minimum number of distinct objects and reuse them rather than creating many duplicate instances.

Now, the code for reading in these data becomes a matter of creating these objects:

```
cfact = new CityFactory();
try {
    InputFile fl = new InputFile(filename);
    String s = fl.readLine ();
    while(s != null) {
        StringTokenizer tok = new StringTokenizer(s, "|");
        Person p = new Person(tok.nextToken (),tok.nextToken ());
        Address a = new Address(tok.nextToken ());
        City city = cfact.addCity (tok.nextToken (),
            tok.nextToken (), tok.nextToken ());
        a.setCity (city);
        PersonLoc pl = new PersonLoc(p, a);
        people.add (pl);
        s = fl.readLine ();
    }
```

and finally, we find that the code to print out the address list just takes successive PersonLoc objects:

```
Iterator iter =  people.iterator();
        while(iter.hasNext () ) {
            PersonLoc pl = (PersonLoc)iter.next ();
            printAddress(pl);
        }
```

The actual printing code gets these objects and prints them:

```
private void printAddress(PersonLoc pl) {
        Person p = pl.getPerson ();
        System.out.print (p.getFrname ()+" ");
        System.out.print (p.getLname ()+" ");
        Address add = pl.getAddress ();
        System.out.print (add.getStreet ()+" ");
        City city = add.getCity ();
        System.out.print (city.getName ()+" ");
        System.out.print (city.getState ()+" ");
        System.out.println (city.getZip ()+" ");
    }
```

Summary

We listed some of the more important criteria Fowler suggests for refactoring, and illustrated a couple of them. I recommend that you read through his book. No matter how accomplished a programmer you are, it is helpful and stimulating to see all of this material laid out so clearly.

We haven't solved the problem of knowledge management of programming life cycles, but you can well imagine that refactoring represents the lingua franca we will have to use in describing this knowledge.

Reference

Martin Fowler, *Refactoring*, Addison-Wesley, Reading, MA, 1999.