TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

ACTA TECHNICA NAPOCENSIS

Series: Applied Mathematics, Mechanics, and Engineering
Vol. 65, Issue I, March, 2022

THE CAVEAT OF OBJECT ORIENTED PROGRAMMING IN JAVA

Tiberiu Alexandru ANTAL

Abstract: *The paper presents the wrong way of rewriting a code that uses procedural paradigm into a code that uses object-oriented paradigm. Because the paradigm can be data-driven or code-driven, the problem is common in code-driven paradigms, when the inexperienced programmer tries to rewrite the code under a data-driven paradigm. As today, most programming languages are multi-paradigm, the paper starts from a solved scientific problem using a structured/modular paradigm and rewrites it wrong, and then correctly using the object-oriented paradigm.*

Key words: *code-drive, data-driven, imperative, paradigm, object-oriented, structured.*

1. INTRODUCTION

1.1 The Harvard and the von Neumann (Princeton) architectures.

The term of caveat should be interpreted as a warning of practicing object oriented programming without having any knowledge of object oriented design. Although the problem to be presented is about software, a variant of approaching the concept of software classification in paradigms ca be derived from hardware. The term of computer architecture is used to describe organization or structure of the components that make up the computer based on their role and interconnection. One of the first computer hardware architecture used was called the Harvard architecture.

As shown in Figure 1 the Harvard architecture has two separate buses one for code and one for data. Hence, the CPU (Central Processing Unit) can access code and read/write data at the same time. However, the existence of two distinct busses will increase the number of electrical lines needed for connection and complicate the control unit of the CPU. In, 1945, the great mathematician John von Neumann, while working at Princeton, designed the architecture from Figure 2 where data and code was stored in the same memory.

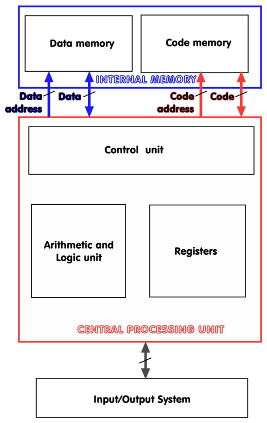

Fig. 1. – The Harvard architecture.

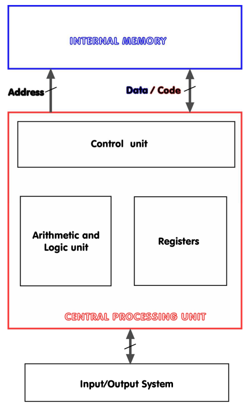

Fig. 2. – The von Neumann (Princeton) architecture.

The CPU is separated from the memory so the statements must be moved from the memory to

Received: 08.01.22; **Similarities**: 09.02.22: **Reviewed**: 11.02./11.02.22: **Accepted**:17.03.22.

the CPU and results must be moved from the CPU to the memory. A common bus was used to data and code transfer between the CPU and the internal memory of the computer. Both architectures have survived to this day, the von Neumann is used for personal computers, being cheaper, while the Harvard architecture is used in micro controllers, being faster but more costly.

1.2 The hardware influence on the software.

The paradigm, in programming, defines a methodology, a style that puts its mark on the way of modeling, and therefore on the solution, of the given problem. The von Neumann architecture gave rise to a category of high-level programming languages that form an isomorphism (Table 1) with the hardware for which they were written. In this sense the following equivalences can be made:

Table 1

The hardware-software isomorphism.

von Neumann architecture (hardware)	**Programming language (software)**
memory locations	variable
machine language	statements
data manipulation	assignment
arithmetic with addressing modes	expressions

One of the first paradigms use in programming was the *imperative* paradigm. In this paradigm it is mandatory to know the solution of the problem to be solved. The programming was consisting in translating the known (mathematical) solution to the high level programming language in order to get the results. In the *procedural* subcategory of the imperative paradigm a known and finite number of transformations are applied to the data stored in variables to obtain the results. Specific to the procedural paradigm is the approach of the resolving is starting from the code that leads to the solution (code-driven). In the *object-oriented* paradigm, also a subcategory of the imperative paradigm, the solution starts from the representation of the data that describe the problem. These data are attached to code sequences, called methods, which describe operations that can be performed with the data. The coupling of data and operations in a single language construction (called class in Java) leads to the formation of data structures that result in objects that interact to obtain results.

1.3 The numerical solution of the problem.

All kinematical problems in robotics lead to equations (see [5] - [8]). The numerical simulation of the manipulator [1] from Figure 3, where O and A are revolute joints (pivots), point E is the end effector and l_{OA} and l_{AE} are the lengths of the OA and AE links can be described by the following equations (origin is considered in O):

$$\begin{cases} x_E = l_{OA}\cos(\varphi_1) + l_{AE}\cos(\varphi_2) \\ y_E = l_{OA}\sin(\varphi_1) + l_{AE}\sin(\varphi_2) \end{cases} \tag{1}$$

For a given set of $\{\varphi_1, \varphi_2\}$ the coordinates (x_E, y_E) of the E end-effector are computed directly from (1).

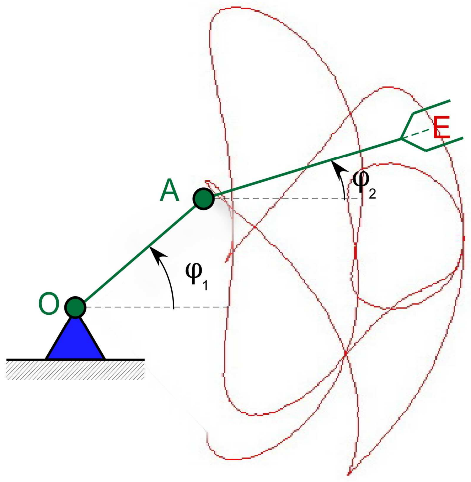

Fig. 3. – Elements of the 2D, 2R manipulator.

2. PROGRAMMING PARARADIGMS USED TO SOLVE THE PROBLEM

2.1 The structured paradigm and the corresponding Java implementation.

The structured paradigm is based on the Böhm–Jacopini theorem [2]. It states that only three rules of grammar are needed to combine any set of basic statements into more complex ones:

1. **Sequence**:
   Do this; then do that
2. **Decision** (or selection or branching):
   IF test is true,
   THEN do this
   ELSE do that
3. **Repetition** (or looping or iteration):
   WHILE test is true
   DO this

(4. optional; depends on the programming language)
STOP/HALT
(5. even more optional, but very useful)
Procedure definition: Define new complex actions by name

The following code if presenting the structured implementation of the solution in Java. It is written inside a class named *rob2Dv1* with a single static method *main( )* as this is the only way in Java to create a runnable code while avoiding the object creation process.

```
public class rob2Dv1 {
 public static void main(String[] args) {
  double fi1, fi2, xe, ye;
  final double l1 = 10., l2 = 7.;

  String s = "";
  for (fi1 = 0.; fi1 <= 6.29; fi1 += 0.1)
   for (fi2 = 0.; fi2 <= 6.29; fi2 += 0.1) {
    xe   =   l1   *   Math.cos(fi1)   +   l2   *
Math.cos(fi2);
    ye   =   l1   *   Math.sin(fi1)   +   l2   *
Math.sin(fi2);
    s += String.format("%.7f,%.7f\n", xe, ye);
   }
   s = "line\n" + s + "\n";
   System.out.print(s);
 }
}
```

All data in the implementation is static, so no *new* operator is needed to create objects to access it.

2.2 The procedural paradigm and the corresponding Java implementation

The procedural paradigm is based on the concept of subroutine and subroutine call. A subroutine (or procedure) is a name given to a group of actions (statements) that can be called from any point of the code (including itself). The modular paradigm is defined as the method of building programs from smaller pieces called usually subroutines. Not any procedural code is modular; modularity is achieved only if a coherent connection of autonomous subroutines can be achieved. The following code if presenting the procedural/modular implementation of the solution in Java:

```
public class rob2Dv2 {
 public  static  double[]  rotate(double  x0,
double y0, double l, double fi) {
  double r[] = new double [2];
  r[0] = x0 + l * Math.cos(fi);
  r[1] = y0 + l * Math.sin(fi);
  return r;
 }

public static void main(String[] args) {
  double fi1, fi2, xa, ya, xe, ye;
  final double l1 = 17., l2 = 5.;
  double  a[]  =  new  double[2],  e[]  =   new
double[2];
  String sr = "", se = "";
  for (fi1 = 0.; fi1 <= 6.29; fi1 += 0.1)
   for (fi2 = 0.; fi2 <= 6.29; fi2 += 0.1) {
    a=rotate(0.,0.,l1, fi1);
    e=rotate(a[0], a[1], l2, fi2);
    sr                                        +=
String.format("pline\n0,0\n%.7f,%.7f\n%.7f,%.7f\
n\n", a[0], a[1], e[0], e[1]);
    se  +=  String.format("%.7f,%.7f\n",  e[0],
e[1]);
   }
   System.out.printf("-layer\ns\nrob\n\n");
   System.out.printf(sr);
   System.out.printf("-layer\ns\ntra\n\n");
   System.out.printf("pline\n"+se+"\n");
  }
 }
```

Modularity is obtained if the inputs are specified syntactically in the form of arguments and the outputs delivered as return values in order to achieve the coupling and the generality of the subroutine. The previous code from *rob2Dv2* class is using the static method *rotate()* that returns an array and inputs the coordinates of the rotation point, the length and the rotation angle. The *main()* method is reusing the subroutine code in two calls in order to perform the two rotations.

2.3 The object oriented paradigm and the corresponding Java implementation

Object oriented software construction is a development method which organizes the architecture of the system to be designed on types of objects that are manipulated to solve the problem. As a class is a user definer data type in Java and the following code is creating the *r1* object based on the class *rob2Dv3* we can state that the implementation is object oriented.

Received: 08.01.22; **Similarities**: 09.02.22: **Reviewed**: 11.02./11.02.22: **Accepted**:17.03.22.

```
public class rob2Dv3 {
 double fi1, fi2, xa, ya, xe, ye, l1 , l2 ;

 public rob2Dv3_1(double l1, double l2) {
  this.l1=l1;
  this.l2=l2;
  compute();
 }

 public  double[] rotate(double x0, double y0,
double l, double fi) {
  double r[] = new double [2];
  r[0] = x0 + l * Math.cos(fi);
  r[1] = y0 + l * Math.sin(fi);
  return r;
 }

 public void compute() {
  double  a[]  =  new  double[2],  e[]  =   new
double[2];
  String sr = "", se = "";
  for (fi1 = 0.; fi1 <= 6.29; fi1 += 0.1)
   for (fi2 = 0.; fi2 <= 6.29; fi2 += 0.1) {
    a=rotate(0.,0.,l1, fi1);
    e=rotate(a[0], a[1], l2, fi2);
    sr                                        +=
String.format("pline\n0,0\n%.7f,%.7f\n%.7f,%.7f\
n\n", a[0], a[1], e[0], e[1]);
    se  +=  String.format("%.7f,%.7f\n",  e[0],
e[1]);
   }
  System.out.printf("-layer\ns\nrob\n\n");
  System.out.printf(sr);
  System.out.printf("-layer\ns\ntra\n\n");
  System.out.printf("pline\n"+se+"\n");
 }

 public static void main(String[] args) {
  rob2Dv3_1 r1 = new rob2Dv3_1(17.,5.);
 }
}
```

The type or the class (*r1*) on which the object (*rob2Dv3*) is based has one constructor and two methods related to the subject: *compute()* and *rotate()*. The *main()* method is mandatory in Java to run the code so it not consider as part of the design.

3. TOWARDS A BETTER OBJECT ORIENTED IMPLEMENTATION USING OBJECT ORIENTED DESIGN CONCEPTS

The elements considered in the above solution were purely geometric, which is why the proposed solution is in fact a procedural one that has been forcibly implemented in an object-oriented form. This is easy to see because there are no specific classes (or user-defined data types) identified on the subject that are used to define the interacting objects to get us to the solution. There is only one class and one object that produce the final results. In the general case, the solution should start from the identification of some general characteristics that can be used in the description of new data types that can describe the individual elements that appear in the context of the problem to be solved. For example, some of the categories of types in the context of the problem could be the:

- structure type: provides identity and linkage description;
- topology type: provides geometrical description based on positions and angles;
- kinematical type: provides displacement, speed and acceleration description;
- dynamic type: provides description of forces, momentum and frictions.

If we consider the concepts of link, joint and robot the following user defined data types can be described: `Link`, `JointR` and `Rob2D`. A link is defined as a moving rigid body (or it can be fixed with respect of a reference when is called frame). As shown in Figure 3 we are in the case of planar mechanisms where all of the relative motions of the rigid bodies are in one plane (or in parallel planes). This will influence the topological description of the types. As no forces and masses are specified the implementation will only refer to a kinematical solution considering only the fundamental concepts of space and time (and maybe quantities like velocity and acceleration derived from there). The new user type called Link will topologically describe the state of a link using four quantities: (x0, y0) - the initial position, l - the length of the body and fi - the angle of the body with respect of the frame. The end point of the link ca be computed with the help of the *getEnd()* method that returns an array of a 2D point.

```
public class Link {
 double x0, y0, l, fi;

 public Link(double x0, double y0, double l,
double fi) {
  this.x0=x0;
  this.y0=y0;
  this.l=l;
  this.fi=fi;
 }

 public double [] getEnd() {
  double e[] = new double[2];
  e[0] = x0+l*Math.cos(fi);
  e[1] = y0+l*Math.sin(fi);
  return e;
```

```
    }

    public String toString() {
     double e[] = getEnd();
     return
String.format("%5.3f,%5.3f\n%5.3f,%5.3f\n",x0,
y0, e[0], e[1] );
    }
   }
```

The following new user type is called `JointR` and will be topologically described by the links l1 and l2 that it connects. This is a revolute joint placed at the intersection of the endpoint of l1 and the initial point of l2. Rotations and translations of the `JointR` types are applied at the initial point of l2.

```
   public class JointR {
    Link l1, l2;

    public JointR(Link l1, Link l2) {
     this.l1 = l1;
     this.l2 = l2;
    }
    public void rotate(double fi) {
     l2.fi = fi;
    }
    public void translate(double t[]) {
     l2.x0 = t[0];
     l2.y0 = t[1];
    }

    public double [] getEnd() {
     return l2.getEnd();
    }
   }
```

The combination of links and joints with a fixed link (a base) are describing the 2D manipulator from Figure 3 in the new user defined data type called `Rob2D`.

```
   public class Rob2D {
    Link l0, l1, l2;
    JointR JO, JA, JE;

    public Rob2D() {
     l0 = new Link(0., 0., 0., 0.);//fixed
     l1 = new Link(0., 0., 17., 0.);//oa
     l2 = new Link(0., 17., 5., 0.);//ae
     JO = new JointR(l0, l1); //O
     JA = new JointR(l1, l2); //A
     JE = new JointR(l2, l2); //E
    }

    public void compute() {
     for (double fi1 = 0.; fi1 <= 6.3; fi1 +=
0.01) {
     JO.rotate(fi1);
     JA.translate(JO.getEnd());
     for (double fi2 = 0.; fi2 <= 6.3; fi2 +=
0.1) {
      JA.rotate(fi2);
      System.out.print("pline\n");
      System.out.print(l1);
      System.out.println(l2);
     }
    }
   }

    public static void main(String[] args) {
     Rob2D r = new Rob2D();
     r.compute();
    }
   }
```

Object oriented design involves finding new data types that are used to describe the data to be processed as well as the use of techniques specific to object-oriented design called composition and inheritance [3], [4]. Composition is applied inside the `JointR` type as the data of this category is composed of instance variables based on the `Link` user defined data type. Composition is also applied in the `Rob2D` class where all the instance variables are based on the `Link` and `JointR` user defined data types. One of the major advantages of a properly object oriented designed code is it's the adaptation to new requirements and problems that are close to the already solved problem. Consider a new problem similar to the one already presented, in which a new joint and a new link are added to the structure. The code to solve the new problem is further presented.

```
   public class Rob2D3R {
    Link l0, l1, l2;
    Link l3;
    JointR JO, JA, JE;
    JointR JE1;

    public Rob2D3R() {
     l0 = new Link(0., 0., 0., 0.);
     l1 = new Link(0., 0., 17., 0.);
     l2 = new Link(0., 17., 5., 0.);
     l3 = new Link(0., 22., 9., 0.);
     JO = new JointR(l0, l1);
     JA = new JointR(l1, l2);
     JE = new JointR(l2, l3);
     JE1 = new JointR(l3, l3);
    }

    public void computeEE(double fi1, double fi2,
double fi3) {
     JO.rotate(fi1);
     JA.translate(JO.getEnd());
     JA.rotate(fi2);
     JE.translate(JA.getEnd());
     JE.rotate(fi3);
     System.out.println("pline");
     System.out.print(l1);
     System.out.print(l2);
     System.out.println(l3);
    }

    public void computeWorkspace() {
     for (double fi1 = 0.; fi1 <= 6.29; fi1 +=
0.3) {
      JO.rotate(fi1);
      JA.translate(JO.getEnd());
      for (double fi2 = 0.; fi2 <= 6.29; fi2 +=
0.3) {
       JA.rotate(fi2);
```

Received: 08.01.22; **Similarities**: 09.02.22: **Reviewed**: 11.02./11.02.22: **Accepted**:17.03.22.

```
       JE.translate(JA.getEnd());
       for (double fi3 = 0.; fi3 <= 6.29; fi3 +=
0.3) {
        JE.rotate(fi3);
        System.out.println("pline");
        System.out.print(l1);
        System.out.print(l2);
        System.out.println(l3);
       }
      }
     }
    }

    public void computeTrajectory(){
     for (double fi1 = 0.; fi1 <= 6.29; fi1 +=
0.01) {
      double fi2=Math.sin(fi1/2.);
      double fi3=Math.sin(fi1);
      computeEE(fi1,fi2,fi3);
     }
    }

    public static void main(String[] args) {
     Rob2D3R r = new Rob2D3R();
     //r.computeWorkspace();
     r.computeTrajectory();
    }
   }
```

4. CONCLUSION

As can be seen, the adaptation of the old code to the new problem is clearly visible at the data level and at the code level. The extension is clear, natural and directly reflects the changes needed to be followed to get the new solution (see the bold elements).

5. REFERENCES

[1] ANTAL, Tiberiu Alexandru. *Principles of motion simulation of a 2 dof rr planar manipulator using java swing.* Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, And Engineering, v. 63, n. 1, 2020. ISSN 2393–2988.

[2] Boehm, Corrado, & Jacopini, Giuseppe *Flow Diagrams, Turing Machines, and Languages with only Two Formation Ru*les, Communications of the ACM, Volume 9, issue 5, pp 366-371, May 1966, https://doi.org/10.1145/355592.365646.

[3] ANTAL, T. A., *Elemente de Java cu JDeveloper - îndrumător de laborator*, Editura UTPRES, 2013, p.150, ISBN: 978-973-662-827-6.

[4] ANTAL, T. A., *Java - Iniţiere - îndrumător de laborator*, Editura UTPRES, 2013, p. 246, ISBN: 978-973-662-832-0.

[5] DETESAN, Ovidiu-Aurelian. *The numerical simulation of TRR small-sized robot.* Acta Technica Napocensis - Series: Applied Mathematics, Mechanics, And Engineering, [S.l.], v. 58, n. 4, nov. 2015. ISSN 1221-5872.

[6] Husty, M., Birlescu, I., Tucan, P., Vaida, C., Pisla, D.: *An algebraic parameterization approach for parallel robots analysis*, Mechanism and Machine Theory, vol. 140, pp. 245-257, 2019

[7] B. Gherman, C. Vaida, D. Pisla, N. Plitea, B. Gyurka, D. Lese, M. Glogoveanu, *Singularities and workspace analysis for a parallel robot for minimally invasive surgery*, Automation Quality and Testing Robotics (AQTR), 2010 IEEE International Conference on, DOI: 10.1109/AQTR. 2010.5520866

[8] Vaida, C; Plitea, N; Gherman, B; Szilaghyi, A; Galdau, B; Cocorean, D; Covaciu, F; Pisla, D; Structural analysis and synthesis of parallel robots for brachytherapy, New Trends in Medical and Service Robots, pp 191-204, 2014, DOI: 10.1007/978-3-319-01592-7_14

Un avertisment cu privire la programarea orientată pe obiect aplicată greşit în Java

Lucrarea prezintă modul de eronat de transcriere a unui cod ce foloseşte paradigma procedurală într-un cod care este implementat utilizând paradigma orientată pe obiect. Deoarece paradigma poate fi condusă atât de date cât şi de cod, problema prezentată apare frecvent în cazul paradigmelor conduse de cod, când programatorul neexperimentat, încearcă sa-l rescrie într-o paradigmă condusă de date. Deoarece astăzi, majoritatea limbajelor de programare sunt multi-paradigma, lucrarea pleacă de la o problemă științifică rezolvată folosind paradigma structurată/modulară şi o reface greşit, iar apoi corect, utilizând paradigma orientată pe obiect.

ANTAL Tiberiu Alexandru, Professor, dr. eng., Technical University of Cluj-Napoca, Department of Mechanical System Engineering, antaljr@bavaria.utcluj.ro, 0264-401667, B-dul Muncii, Nr. 103-105, Cluj-Napoca, ROMANIA.