TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

ACTA TECHNICA NAPOCENSIS

Series: Applied Mathematics, Mechanics, and Engineering
Vol. 68, Issue IV, November, 2025

AUTOCAD PROGRAMMING IN JAVA

Tiberiu Alexandru ANTAL, Radu Mircea MORARIU-GLIGOR, Julieta Daniela CHELARU

Abstract: *AutoCAD is commercial CAD software used across multiple industries for 2D and 3D drafting and modeling. Java is a high-level, object-oriented programming language and computing platform used across numerous domains. The following paper describes how AutoCAD can be programmed from Java using COM Automation via Jacob Library.*

Key words: *AutoCAD, Automation, COM, Java, mechanism, trajectory.*

1. INTRODUCTION

Most CAD products can be programmed [2] using different approaches through programming languages and technologies. Because the work refers to AutoCAD the technologies and languages will be briefly presented related to this product. Not all CAD products implement the same technologies and programming languages, most have proprietary language implementations and a lot of the technologies are platform dependent (for example Windows operating system and x86/x64 processor architecture). AutoCAD's native programming language is AutoLISP, which is a dialect of LISP. In time AutoLISP was replaced by Visual LISP a considerably enhanced version of AutoLISP including an integrated development environment, debugger, compiler, and ActiveX support (see Figure 1). Consider the following VLISP file:

```
(defun c:DrawL ()
  (command "LINE" "0,0" "10,5" "")
  (princ)
)
```

saved under the *draw.lsp* name with the following path "D:/Work/Lucrari stiintifice/2025/AutoCADcuRadu/draw.lsp". Once the LISP file is loaded from the *File > Load File* menu, the `DrawL` command can be launched from the command prompt and will draw a line on the *Drawing1* AutoCAD drawing.

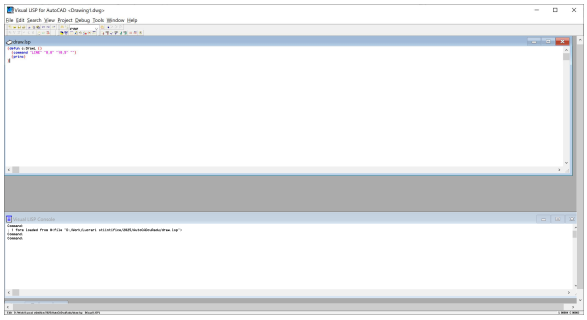

Fig. 1. - The Visual LISP IDE from AutoCAD.

VBA (Microsoft Visual Basic for Applications) software is another language that can be used to program AutoCAD. Compared to VLISP, VBA is no longer installed by default and must be downloaded and installed properly (using the right version and platform) from the link https://www.autodesk.com/vba-download. In order to use the IDE for the Visual Basic Editor a new module must be created from *Insert > Module* and a VBA subroutine must be created (see Figure 2):

```
Sub DrawL()
 Dim acadApp As Object
 Dim acadDoc As Object
 Dim startPoint(0 To 2) As Double
```

Received: 30.09.25; **Similarities**: 21.10.25: **Reviewed**: 22.10./26.10.25: **Accepted**:12.11.25.

```
 Dim endPoint(0 To 2) As Double
 Set      acadApp      =      GetObject(,
"AutoCAD.Application")
 Set acadDoc = acadApp.ActiveDocument
 ' Define points
 startPoint(0)  =  0:  startPoint(1)  =  0:
startPoint(2) = 0
 endPoint(0)   =   10:   endPoint(1)   =   5:
endPoint(2) = 0
 ' Create line
 acadDoc.ModelSpace.AddLine    startPoint,
endPoint
 acadApp.ZoomExtents
End Sub
```

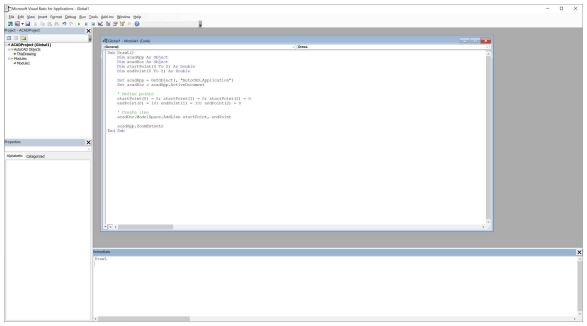

Fig. 2. - The VBA IDE from AutoCAD.

In order to run the `DrawL()` subroutine the *Immediate Windows* from *Insert* menu must be opened and the name of the must be typed in to obtain a line on the current drawing.

So far we have described the two programming languages integrated in AutoCAD, what follows is a suite of technologies through which the Java programming language can be used to interact with AutoCAD using JACOB. JACOB (Java COM Bridge) is a library that allows Java applications to interact with COM (Component Object Model) components, including Windows applications like AutoCAD. JACOB is an open-source Java library that provides a bridge between Java and COM/ActiveX components to automate and control Windows applications that expose COM interfaces. Java [3], [4] and COM are fundamentally different technologies, Java is platform-independent, runs in JVM and uses JNI for native calls while COM is a Windows-specific, binary standard, language-independent and the bridge allows them to interoperate despite these differences. JACOB translates Java method calls into COM calls, which AutoCAD's COM Automation server executes, giving Java programs control over AutoCAD. AutoCAD exposes COM objects as AutoCAD has a registered COM server (e.g., "AutoCAD.Application") that lets external programs control it. So a Java piece of code can manipulate drawings, layers, blocks, and run commands. Java code is using JACOB's Dispatch class to talk to the COM objects exposed by AutoCAD.

2. JACOB and COM AUTOMATION

COM (Component Object Model) is a Microsoft technology from the 1990s. It lets different programs (written in different languages) talk to each other by exposing objects with properties and methods. COM Automation (also called OLE Automation) is a scripting interface for applications like AutoCAD, Excel, Word, etc. An application like AutoCAD exposes itself as a COM Automation server. Other programs (clients, like Java or Python) can connect and automate tasks like: open files, draw lines or circles, run commands, save/export. Java has no built-in COM support (COM is a Windows-only, C/C++ technology). JACOB uses JNI to bridge Java. JNI (Java Native Interface) is a standard way for Java code (running on the JVM) to call native code written in C, C++, or assembly, and vice versa. Without JNI, Java is "trapped" inside the JVM and can't directly call Windows system APIs like COM. When interacting with AutoCAD from Java a call flow can be tracked by the sequence: Java → JNI → JACOB → COM → AutoCAD. When the programmer writes the sequence to draw a line (`AddLine`) between two points (0, 0 , 0) and (100, 100, 0):

```
ActiveXComponent          acad          =
ActiveXComponent("AutoCAD.Application");

Dispatch.call(modelSpace,  "AddLine",  new
Variant(0), new Variant(0), new Variant(0),
new  Variant(100),  new  Variant(100),  new
Variant(0));
```

the `Dispatch.call()` is declared as a native method and the JVM sees it and says: "I need JNI to run this". At the JACOB Java layer the following classes are defined:

- `ActiveXComponent` - connects to COM server;

- `Dispatch` - wrapper for COM objects;
- `Variant` - wraps COM data types;
- Methods like "Dispatch.call(...)" are declared as "native" methods.

The control jumps into jacob.dll through JNI. The jacob.dll (native code = no virtual machine, the code is compiled into raw CPU instructions) used COM calls like `IDispatch::Invoke` to tell AutoCAD "Run the method `AddLine` on the ModelSpace object with these arguments." AutoCAD exposes objects via COM like: Application, Documents, ActiveDocument, ModelSpace and methods like: AddLine, AddCircle, SaveAs, ZoomAll. AutoCAD's COM engine parses the request and executes `AddLine` internally and a line appears in the drawing from (0, 0, 0) → (100, 100, 0). AutoCAD may return a handle to the new line object. COM sends it back to jacob.dll which sends it to JNI which will wrap it in a Java `Dispatch` or `Variant`. Briefly we can summarize that:

- Java calls JACOB (Java side).
- JACOB calls into its jacob.dll (native C++).
- jacob.dll calls the Windows COM API CoCreateInstance() to create an AutoCAD instance.
- Windows COM system looks up "AutoCAD.Application" in the registry.
- It finds AutoCAD's COM server and asks it to start (if not already running).
- AutoCAD creates an Application object in memory;
- It gives back an IDispatch pointer to the caller; that COM pointer travels back → jacob.dll → JACOB Java wrapper → Java `ActiveXComponent`.
- Now the Java object `acad` is just a reference to the real AutoCAD Application object.

3. A JACOB AND JDEVELOPER

JACOB is not a standalone program; it's a library you include in your Java project by downloading it from the official JACOB project page on SourceForge: https://sourceforge.net/projects/jacob-project/

At this moment the latest version is `jacob-1.20.zip`. After extracting the ZIP file we will find a file named *jacob.jar* (the Java library containing Java classes and interfaces). This must be added to the Java project classpath in JDeveloper as shown in Figure 3.

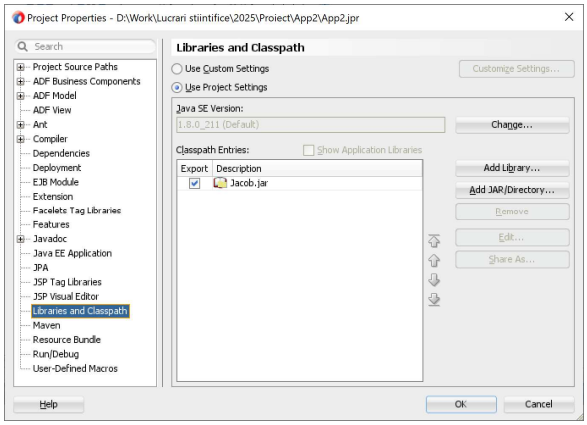

Fig. 3. - jacob.jar classpath location in JDeveloper.

The ZIP also contains a crucial *.dll* (the native Windows DLL that handles the actual COM communication) file (e.g., *jacob-1.20-x64.dll* for 64-bit systems *or jacob-1.20-x86.dll* for 32-bit). This native library must be placed in a location where Java can find it.

4. A JAVA APPLICATION TO SIMULATE MECHANISM MOTION

A mechanism simulation application must be able to graphically represent in AutoCAD the positions of the kinematic elements (linkages) and joints together with the trajectories they follow. As the joints are connections between links that allow for specific relative motions (e.g., pin joints for rotation, prismatic joints for sliding) it is possible not to represent them physically because their type can be deduced from the generated trajectories. In Figure 4, a simulation of the four-bar linkage mechanism ([1], [5]) is provided. Having knowledge about the mechanism and seeing the trajectories, the type of joints can be deduced. Sometimes there are situations when we want to represent the joints too (see Figure 5) and in AutoCAD this is not a problem. As it allows drawing on layers we can organize the code on several layers, a layer for the initial state of the mechanism with all

joints and linkages, another layer for the simulation and a last layer for the trajectory of the coupler point ([1], [8]). It is useful to generate intermediate positions of the mechanism using polylines because when selecting any part of a position, all the other parts corresponding to the position will be automatically selected. This way we can easily identify all the linkages corresponding to a position by selecting a single linkage.

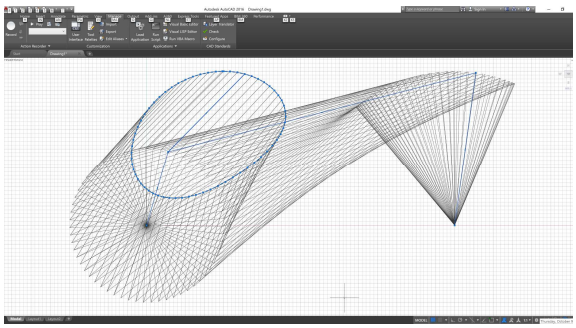

Fig. 4. - A four-bar linkage mechanism simulation in AutoCAD.

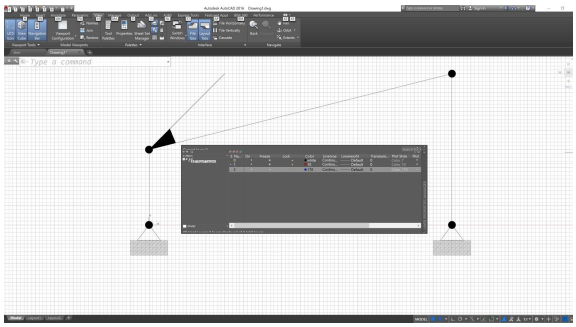

Fig. 5. - The four-bar linkage mechanism initial state in AutoCAD with revolute joints and base represented.

The following code is creating:

1. Two lines from origin to (10,10) and (20,0)
2. One circle at (5,5) with radius 3
3. A four-bar linkage drawn as a polyline
4. Sine wave trajectory - 50-point mathematical curve
5. Final zoom to show all geometry

```
  import
com.jacob.activeX.ActiveXComponent;
  import com.jacob.com.Dispatch;
  import com.jacob.com.Variant;
  import com.jacob.com.SafeArray;

  public class acad1 {
    private ActiveXComponent acadApp;
    private Dispatch activeDocument;
    private Dispatch modelSpace;
    public boolean initialize() {
     try {
      acadApp            =            new
ActiveXComponent("AutoCAD.Application");
      acadApp.setProperty("Visible",    new
Variant(true));
      activeDocument                      =
acadApp.getProperty("ActiveDocument").toDi
spatch();
      modelSpace                          =
Dispatch.get(activeDocument,
"ModelSpace").toDispatch();
      return true;
     } catch (Exception e) {
      System.err.println("Failed        to
initialize AutoCAD: " + e.getMessage());
      return false;
     }
   }

  public void drawLine(double x1, double
y1, double z1, double x2, double y2, double
z2) {
    try {
   // Create the start point as a Variant
array
    Variant          startPoint          =
createPointVariant(x1, y1, z1);
   // Create the end point as a Variant
array
    Variant           endPoint           =
createPointVariant(x2, y2, z2);
   // Call AddLine with the two point arrays
    Dispatch.call(modelSpace,   "AddLine",
startPoint, endPoint);

    System.out.println("Line drawn from ("
+ x1 + "," + y1 + "," + z1 +
                                    ") to (" +
x2 + "," + y2 + "," + z2 + ")");
    } catch (Exception e) {
    System.err.println("Failed   to   draw
line: " + e.getMessage());
              e.printStackTrace();
    }
   }
   // Helper method to create a point as a
Variant containing a SafeArray
   private                          Variant
createPointVariant(double   x,   double   y,
double z) {
   //   Create   a   SafeArray   of   type
Variant.VariantDouble with 3 elements
    SafeArray     pointArray     =     new
SafeArray(Variant.VariantDouble, 3);
    pointArray.setDouble(0, x);
    pointArray.setDouble(1, y);
```

```
    pointArray.setDouble(2, z);
    return new Variant(pointArray);
   }

   public void drawCircle(double centerX,
double centerY, double centerZ, double
radius) {
    try {
   // Create center point as Variant array
    Variant          centerPoint          =
createPointVariant(centerX,       centerY,
centerZ);
    Variant      radiusVar      =      new
Variant(radius);
    Dispatch.call(modelSpace, "AddCircle",
centerPoint, radiusVar);
    System.out.println("Circle drawn at ("
+ centerX + "," + centerY +
                                  ","      +
centerZ + ") with radius " + radius);
    } catch (Exception e) {
    System.err.println("Failed   to   draw
circle: " + e.getMessage());
    e.printStackTrace();
    }
   }

   public void zoomAll() {
    try {
     Dispatch.call(activeDocument,
"SendCommand", "ZOOM A ");
     System.out.println("Zoomed All");
    } catch (Exception e) {
    System.err.println("Failed to zoom: " +
e.getMessage());
    }
   }

   public void testConnection() {
    try {
   // Test if we can get AutoCAD version
    String            version            =
acadApp.getProperty("Version").getString()
;
    System.out.println("Connected       to
AutoCAD version: " + version);
     } catch (Exception e) {
    System.err.println("Connection    test
failed: " + e.getMessage());
    }
   }

   public                               void
drawPolylineWithProperties(double[][]
points, String layer, int color, double
width) {
    try {
   // Create coordinates array
    SafeArray     coordinates     =     new
SafeArray(Variant.VariantDouble,
points.length * 2);
    int index = 0;
    for (double[] point : points) {
     coordinates.setDouble(index++,
point[0]);
     coordinates.setDouble(index++,
point[1]);
   }

   // Create polyline
    Variant     coordsVariant     =     new
Variant(coordinates);
    Dispatch           polyline           =
Dispatch.call(modelSpace,
"AddLightWeightPolyline",
coordsVariant).toDispatch();
    delay(1500);
   // Set properties
    if (layer != null) {
     Dispatch.put(polyline,  "Layer",  new
Variant(layer));
    }
    if (color > 0) {
     Dispatch.put(polyline,  "Color",  new
Variant(color));
     }
    if (width > 0) {
   // Set constant width
     Dispatch.put(polyline,
"ConstantWidth", new Variant(width));
     }
     System.out.println("Polyline  created
with custom properties");
    } catch (Exception e) {
    System.err.println("Error     creating
polyline: " + e.getMessage());
    e.printStackTrace();
    }
   }

   private void delay(int milliseconds) {
    try {
     Thread.sleep(milliseconds);
    } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    }
   }

   public static void main(String[] args){
    acad1 manager = new acad1();
    if (manager.initialize()) {
     try {
      manager.testConnection();

   // Draw some geometry
      manager.drawLine(0, 0, 0, 10, 10, 0);
      manager.drawLine(0, 0, 0, 20, 0, 0);
      manager.drawCircle(5, 5, 0, 3);
```

```
    double[][] fourbarlinkagePL = {
     {0, 0}, {0, 1}, {5, 3}, {6, -1},
     {5, 3}, {0, 1}, {3, 5}
     };

manager.drawPolylineWithProperties(fourbar
linkagePL, "0", 0, 0);
    double [][] trajectoryPL = new double
[50][2];
    for (int i = 0; i < 50; ++i) {
     double x = (double) i/ Math.PI;
     trajectoryPL[i][0] = x;
     trajectoryPL[i][1] = Math.sin(x);
    }

manager.drawPolylineWithProperties(traject
oryPL, null, 1, 0);
    manager.zoomAll();
    System.out.println("Drawing completed
successfully!");
    } catch (Exception e) {
      e.printStackTrace();
     }
    }
   }
  }
```

The results printed by the code are following as well as the drawing created in AutoCAD (see Figure 6). Some explanations on the code are also given the help understating the meaning of the methods, the way they work together with some of their parameters and the returned values.

```
Connected to AutoCAD version: 20.1s (LMS Tech)
Line drawn from (0.0,0.0,0.0) to (10.0,10.0,0.0)
Line drawn from (0.0,0.0,0.0) to (20.0,0.0,0.0)
Circle drawn at (5.0,5.0,0.0) with radius 3.0
Zoomed All
Drawing          completed          successfully!
```

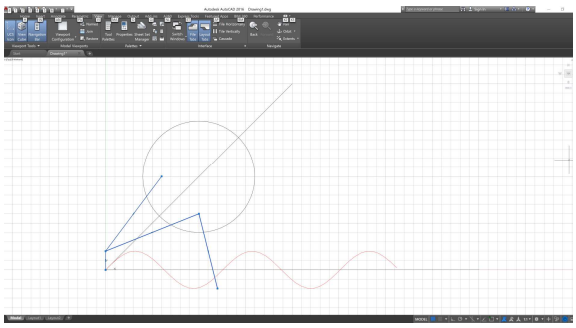

Fig. 6. - Result of the Java code in AutoCAD.

The import statements are:

- `ActiveXComponent`: Main class for connecting to COM objects (AutoCAD)
- `Dispatch`: Used to call methods and properties on COM objects
- `Variant`: Wrapper for data types passed between Java and COM
- `SafeArray`: Used to create arrays that COM can understand

The class fields are:

- `acadApp`: Represents the AutoCAD application itself
- `activeDocument`: Represents the current drawing document
- `modelSpace`: Represents the model space where geometry is drawn

The `initialize()` method contains:

1. `new ActiveXComponent("AutoCAD.Application")` - Connects to running AutoCAD or starts new instance
2. `setProperty("Visible", new Variant(true))` - Makes AutoCAD window visible
3. `getProperty("ActiveDocument")` - Gets the current drawing document
4. `Dispatch.get(activeDocument, "ModelSpace")` - Accesses the model space container

The `createPointVariant()` is a helper method that creates a 3D point as a COM-safe array based on:

1. `SafeArray`: Special array type that COM understands
2. Stores X, Y, Z coordinates in indices 0, 1, 2
3. Returns as Variant for COM interoperability

The `drawLine()` method works like this:

1. Creates start and end points as Variant arrays
2. Calls `AddLine` method on model space with both points
3. AutoCAD creates a line between the specified coordinates

The `drawCircle()` method work like this:

1. Creates center point as Variant array
2. Creates radius as Variant
3. Calls `AddCircle` method with center and radius

The `drawPolylineWithProperties()` method creates a 2D polyline in AutoCAD with customizable properties like layer, color, and line width. The used parameters are :

- `points`: 2D array of coordinates `[[x1,y1], [x2,y2], [x3,y3], ...]`
- `layer`: Layer name (null = use current layer)
- `color`: AutoCAD color index (0 = ByBlock, 1-255 = specific colors, 256 = ByLayer)
- `width`: Line width in drawing units (0 = default width)

The coordinate array creation is based on:

- `SafeArray(Variant.VariantDouble, points.length * 2)`:
  1. Creates a COM-compatible array of double type
  2. Size = number of points × 2 (each point has X and Y)
  3. Example: 5 points → 10 elements array
- Loop through points:
  1. point[0] = X coordinate
  2. point[1] = Y coordinate
  3. Result: Flat array [x1, y1, x2, y2, x3, y3, ...]

The polyline creation is based on:

- `Wrap in Variant`: coordinates → coordsVariant (COM-compatible)
- Call AutoCAD Method: `Dispatch.call()` invokes AutoCAD's COM interface
- Method: `AddLightWeightPolyline` - creates optimized 2D polyline
- Parameters: Only the coordinates array
- Return: Dispatch object representing the created polyline

The `ZoomAll()` method is a utility and sends "ZOOM A" command to AutoCAD command line ("A" means "All" - zooms to show all objects).

The `testConnection()` method verifies the successful connection to AutoCAD.

The `delay()` method prevents COM "application busy" errors by adding pauses.

The error handling in this AutoCAD automation code follows a defensive programming paradigm designed specifically for the unpredictable nature of COM interop and external application control. Rather than allowing the application to crash when AutoCAD encounters issues, the code implements a multi-layered safety net that gracefully handles failures while providing meaningful diagnostic information. The error handling architecture is built on several key principles:

- **Graceful Degradation**: The application is designed to continue operating even when individual AutoCAD commands fail. This is crucial for batch processing scenarios where you might want to process hundreds of drawings - a single failure shouldn't halt the entire operation.
- **Contextual Error Reporting**: Each method provides specific error messages that clearly indicate which operation failed. Instead of generic "COM error" messages, you get precise information like "Failed to draw line" or "Error creating polyline," making troubleshooting significantly easier.
- **Resource Safety**: While not explicitly shown in resource cleanup code, the Jacob library manages COM resource lifecycle automatically. The error handling ensures that even when operations fail, there are no resource leaks that could destabilize either the Java application or the AutoCAD process

5. CONCLUSIONS

The code demonstrates a robust approach to automating AutoCAD from Java, handling both simple geometry creation and complex polylines with proper error handling and performance considerations. The bridge architecture between two fundamentally different technology ecosystems is used with success to demonstrate

the Java-AutoCAD integration creating a powerful synergy that leverages the strengths of both platforms. The implementation demonstrates the cross-technology value maximization by using Java as a "glue language" to orchestrate specialized CAD operations. This approach recognizes that while AutoCAD excels at geometric modeling and visualization, Java provides superior capabilities [6], [7], [9] for enterprise integration, data processing, and web services - creating a whole greater than the sum of its parts.

6. REFERENCES

[1] ANTAL, Tiberiu Alexandru. *Python in the planar four-bar linkage mechanism simulation*. ACTA TECHNICA NAPOCENSIS - Series: APPLIED MATHEMATICS, MECHANICS, and ENGINEERING**,** v. 68, n. 1 & 2, jun. 2025. ISSN 2393–2988.

[2] ANTAL, T .A., *Programming AutoCAD using JAWIN from Java in JDeveloper, Acta Technica Napocensis*, Series: Applied Mathemathics and Mechanics, ISSN 1221-5872, 53(3), p.481-486, Cluj-Napoca, 2010.

[3] ANTAL, T. A., *Java - Iniţiere - îndrumător de laborator*, Editura UTPRE, *ediţia a II-a*, Editura RISOPRINT, 2006, p.264, ISBN 973-751-349-5.

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

[5] ANTAL, T. A., *Mechanism displacement analysis with AutoLisp in AutoCAD. Acta Technica Napocensis*, Series: Applied Mathematics and Mechanics, ISSN 1221-5872, 45, p. 19-24, Cluj-Napoca, 2002.

[6] ANTAL, Tiberiu Alexandru. *About the transformation of a structured code to a library in Java.* ACTA TECHNICA NAPOCENSIS - Series: Applied Mathematics, Mechanics, and Engineering, v. 67, n. 1, mar. 2024. ISSN 2393–2988.

[7] Musa Abdul, Lawal Dalhatu Eneyemire, & Abubakar Muhammad. (2024). *Design and development of a java-based cad environment for builders and architects using the mvc pattern*. International Journal of Engineering Processing and Safety Research, 5(5), 2024.

[8] ANTAL, Tiberiu Alexandru. *A machine learning approach in planar mechanism trajectory approximation*. ACTA TECHNICA NAPOCENSIS - Series: APPLIED MATHEMATICS, MECHANICS, and ENGINEERING, v. 68, n. 2, 2025. ISSN 2393–2988.

[9] Moreno Cazorla, Ricardo & Bazán, A. *Design Automation Using Script Languages. High-Level CAD Templates in Non-Parametric Programs.* IOP Conference Series: Materials Science and Engineering. 2017. 245. 062039. 10.1088/1757-899X/245/6/062039.

Programarea AuoCAD prin Java

AutoCAD este un software CAD comercial utilizat în multiple industrii pentru desenare și modelare 2D și 3D. Java este un limbaj de programare orientat pe obiecte, de nivel înalt și o platformă de calcul utilizată în numeroase domenii. Articolul descrie modul în care AutoCAD poate fi programat din Java folosind COM Automation prin intermediul bibliotecii Jacob.

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

MORARIU-GLIGOR Radu Mircea, Assoc. Professor, dr. eng., Technical University of Cluj-Napoca, Department of Mechanical System Engineering, radu.morariu@mep.utcluj.ro, 0264-401654, B-dul Muncii, Nr. 103-105, Cluj-Napoca, ROMANIA.

CHELARU Julieta Daniela, Associate Professor, dr. eng., Babeș-Bolyai University, Department of Chemical Engineering, julieta.chelaru@ubbcluj.ro, Arany Janos, Nr. 11, Cluj-Napoca, ROMANIA.