TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

ACTA TECHNICA NAPOCENSIS

Series: Applied Mathematics, Mechanics, and Engineering
Vol. 69, Issue Special III, July, 2026

OBJECT-ORIENTED DESIGN APPLIED TO THE FIELD OF PLANAR MECHANISMS IN JAVA

Tiberiu Alexandru ANTAL, Bogdan GHERMAN

Abstract: *Object-oriented design and the object-oriented programming paradigm are related fields that can be applied in an integrated manner to create applications in which code reuse or extension becomes natural and simple. The paper aims to present the conditions under which this integration is successfully achieved when using AutoCAD as an interface tool.*
Keywords: *AutoCAD, Automation, COM, Java, mechanism, trajectory.*

1. INTRODUCTION

A programming paradigm is a fundamental style or approach to programming that provides an underlying structure for organizing code. It is characterized by:

- a set of concepts, principles, and patterns;
- a way of thinking about problem-solving;
- constraints and guidelines for program structure;
- often a specific methodology for computation.

The creation of a programming paradigm is not a singular act of invention by a language, but rather emerges from a deliberate community of design choices, enforced constraints, and conceptual innovations that, together, shape how programmers structure and reason about code. As a programming language fosters a paradigm by hardwiring a particular computational philosophy into its language constructs (or more philosophically in its DNA) - through constraints that discourage some old patterns and provide new patterns - it is useful to analyze the evolution of programming language generations.

2. A BRIEF OVERVIEW OF PROGRAMMING LANGUAGE GENERATIONS AND THE CONNECTION TO MECHANICS

Hardware capabilities and programming paradigms exist in a state of mutual causation; each one directly shapes and is shaped by the other. This relationship is defined by reciprocal innovation: advancements in one domain are both a consequence of and a catalyst for progress in the other. Without claiming to be an exhaustive classification, the hardware generations and paradigms inspired by it is:

- **Generation 1** (1940s) was about hardware (vacuum tubes) that could only understand binary (machine language).
  Hardware: Vacuum tubes, plug boards, manual switches;
  Programming: Machine code by hand by wiring the machine directly.
- **Generation 2** (1950s) was about transistors and batch processing. Increased memory could store symbol tables while the CPU speed improved and could afford compilation overhead.
  Hardware: Transistors, magnetic core memory, punched cards;
  Programming: Assembly led to first compilers (FORTRAN, COBOL);

Connection: Assembly reflected CPU instruction sets directly.

- **Generation 3** (1960s) was about Integrated Circuits (ICs) and Timesharing OSs that allowed multiple users creating the need for better abstraction. Memory became cheaper and could store compilers and runtime systems.
  Hardware: ICs, faster CPUs, terminals replacing cards;
  Programming: High-level languages, interactive debugging;
  Connection: ALGOL's **structured programming** matched new CPU capabilities.
- **Generation 4** (1970s-80s) was about Microprocessors and Personal Computing. The limited memory on early PCs required efficient languages (C) and GUI development needed event-driven programming.
  Hardware: 4-bit to 64-bit CPUs, RAM expanding, graphics;
  Programming: C for systems, Pascal for education, OOP emerges;
  Connection: C's pointers and manual memory management matched hardware memory layout.
- **Generation 5** (1990s) was about RISC, Parallelism, Internet. The Internet created the need for portable code (Java's VM), faster networks allowed the client-server architectures appear.
  Hardware: RISC architectures, network cards, CD-ROMs;
  Programming: OOP dominates (Java, C++), Scripting (Python, Perl), Web (JavaScript);
  Connection: Garbage collection became viable with more memory.
- **Generation 6** (2000s-2010s) was about Multicore, Mobile, Cloud. The Multicore needed tools for parallel programming (Go, Rust, Erlang), the new resource-constrained mobile hardware, demanded optimized languages (Swift, Kotlin).
  Hardware: Multicore CPUs, GPUs, smartphones, SSDs, cloud data centers;
  Programming: Concurrent/parallel languages, functional revival, scripting explosion;
  Connection: GPU computing needed data-parallel languages (CUDA, OpenCL)
- **Generation 7** (2020s+) is about AI Accelerators, Quantum, Edge. AI chips implemented tensor operations as primitives and Quantum computing needed an entirely new computational model.
  Hardware: TPUs, quantum bits, IoT devices, neuromorphic chips;
  Programming: Domain-specific languages, probabilistic programming, differentiable programming
  Connection: Hardware specialization leads to software specialization.

Hardware and software are locked in a cycle of symbiotic advancement. Each generation of hardware enables new programming paradigms, whose widespread adoption creates the imperative for the next hardware revolution.

3. SUCCESSFUL PARADIGMS IN MECHANICS

A normal question would be why, among so many paradigms, the imperative (Generation 2) and the object-oriented (Generation 4) are the most used. The imperative paradigm's relation to mechanics is fundamental as it provides the sequential, algorithmic logic that makes mechanical simulation and control possible. At the same time, mechanical systems are physical objects where the objects interact. Software objects model physical objects. Parts become classes. Assemblies contain parts. Inheritance creates taxonomies. The linkage is an object. The calculation of its motion is a structured procedure inside it. Structured programming lives inside OOP methods. While OOP models what the system is, structured programming defines how each component behaves step-by-step.

4. OBJECT-ORIENTED PROGRAMMING AND DESIGN IN JAVA

Without claiming that the field of Object-Oriented Programming (OOP) is completely presented, it is related to the following principles that organize software around objects that combine [3]:

- Data (properties/state)
- Behavior (methods/functions)
- Relationships (how objects connect)

The core principles of this software design are:

- Encapsulation - Hide complexity inside objects. Only expose what's necessary;
- Inheritance - Create new objects based on existing ones;
- Polymorphism - Different objects can respond to the same message in their own way;
- Abstraction - Simplify reality by modeling only relevant details.

OOP structures code as a set of objects. Each object is an independent entity with its own knowledge and capabilities. Objects communicate by sending messages. Not all languages implement these principles the same way. As the following code is in Java, I will refer the specific implementations of the ideas related to this language. Java is class-based [1], [2]. You must first define a `class`, which is a template. From this class, you create **objects** using the `new` keyword. A class defines **fields** (for data) and **methods** (for behavior). **Encapsulation** in Java is enforced with access modifiers: `private`, `public`, `protected`. `private` fields are locked inside the object. You expose controlled access via public methods, often called getters and setters. **Inheritance** in Java uses the `extends` keyword. A class can inherit from one parent class only (single inheritance). The child class gains all the non-private fields and methods of the parent. **Polymorphism** in Java works through inheritance and interfaces. A parent class reference can point to a child class object. When a method is called, Java executes the child's version (if overridden). This is runtime or dynamic polymorphism. A core Java feature is the interface. An interface is a pure contract - a list of method signatures without implementation. A class `implements` an interface and must provide code for all its methods. A class can implement multiple interfaces, which is Java's way around single inheritance. **Abstraction** is achieved using `abstract` classes and interfaces. An abstract class is a partial blueprint that cannot be instantiated; it must be extended. It can contain both implemented and abstract (unimplemented) methods. Java handles memory automatically through garbage collection. When no references to an object remain, it becomes eligible for removal, preventing memory leaks. Java's OOP provides a disciplined, structured, and safe framework for building programs as interacting objects, enforcing clear rules for encapsulation, inheritance, and polymorphism through its syntax and type system.

5. SOME WORDS ON OBJECT-ORIENTED DESIGN

The Object Oriented Design (OOD) phase happens before any code is written. It is about the process of conceptual modeling [5], [10]. You look at the problem and you ask questions like: What are the key things? What do they know? What can they do? How do they relate? You identify nouns as potential objects (Joint, Linkage, Motor). You identify verbs as potential methods (step(), rotate(), translate(), move(), stop()). You group common features into hierarchies (Revolute Joints, Prismatic Joints and Fixed Joints are kinds of Joints). You define how objects collaborate (Linkage objects use Joints object to process/produce the motion/movement). Without domain knowledge and the ability to identify and use the domain's concepts that constitute the problem, one cannot make a correct design of the problem to be solved because one cannot represent the concepts and behavior. This means that someone who only knows programming but is not familiar with the domain cannot meet the requirement because one cannot program in an object-oriented way correctly if one cannot design in an object-oriented way - which requires knowing the domain.

6. AN OOD TEMPLATE FOR PLANAR MECHANISMS

A link is a single rigid body (a physical part). It is the building block. It has mass, shape, and length. A link connects to other links at points called joints. A binary link (Figure 1) has two connection points (a ternary link has three, a quaternary link has four). In a diagram, a link (binary) is one line or shape (ternary, quaternary).

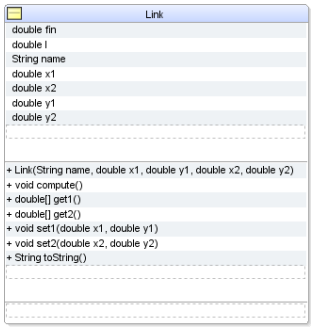


Fig. 1. - The binary link conceptual representation.

The two points are stored by their cartesian coordinates and they define a fixed length (`l`) and an initial angle (`fin`). The getters (`get1()` and `get2()`) and setters (`set1()` and `set2()`) extract and change the two points' coordinates.

A joint is a connection between two links. A joint allows specific relative motion. It constrains some degrees of freedom while permitting others. A joint is where movement happens. It is a mechanical connection point. In planar mechanisms, the revolute joint (hinge, pin) is one of the fundamental connectors. It is the simplest and most common joint for creating rotating motion in a plane. It allows pure rotation about a single axis. One link rotates relative to the other. The axis of rotation is perpendicular to the plane of motion in a planar mechanism. It removes all other motions: no sliding, no lifting. The conceptual definition (Figure 2) is based on abstraction where `JointR` is the root for two concrete revolute joints, the active revolute joint (`JointRA`) and the passive revolute joint (`JointRP`). Active means it imposes motion. Passive means it follows motion. An active revolute joint is a revolute joint where the rotation is controlled. A motor, an actuator, or an external force directly sets its angle or its angular velocity. It is the input to the mechanism. It is the source of motion. A passive joint rotates freely in response to forces from connected links. Its motion is determined by the system's geometry and the active input.

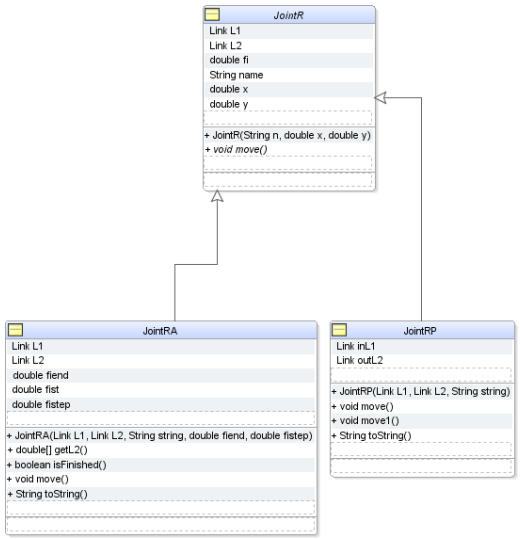


Fig. 2. - The revolute joint conceptual representation.

A motor chain (Figure 3) is a kinematic chain with one or more active joints. It is the complete mechanical path from the power source (the motor) to the end effector. The motor applies force or motion at an active joint. This motion then propagates through the entire connected series of links and passive joints - the chain - to produce the desired final movement.

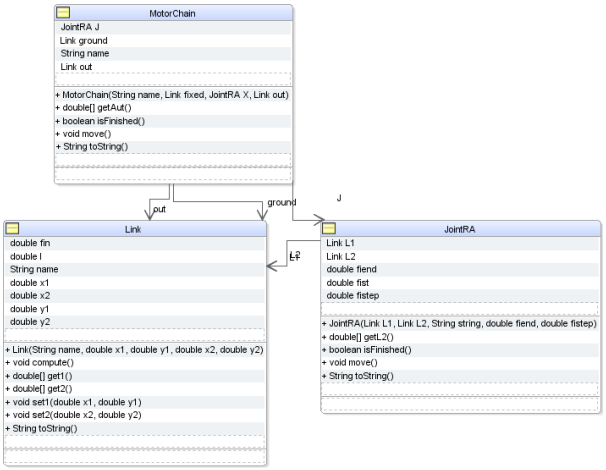


Fig. 3. - The motor chain conceptual representation.

The motor chain from Figure 3 has one active joint and is created by the composition of an active joint, a fixed (and reference) link called ground and a moved link by the active joint. An Assur group is a fundamental, rigid building

block in the structural theory of mechanisms. It is a kinematic chain with the specific properties:

- It has zero degrees of freedom on its own (it's rigid if disconnected).
- When attached to a mechanism, it adds zero degrees of freedom.
- It cannot be decomposed into smaller groups with the same properties.

It is the smallest, non-breakable unit you can add to a mechanism without changing its overall mobility.

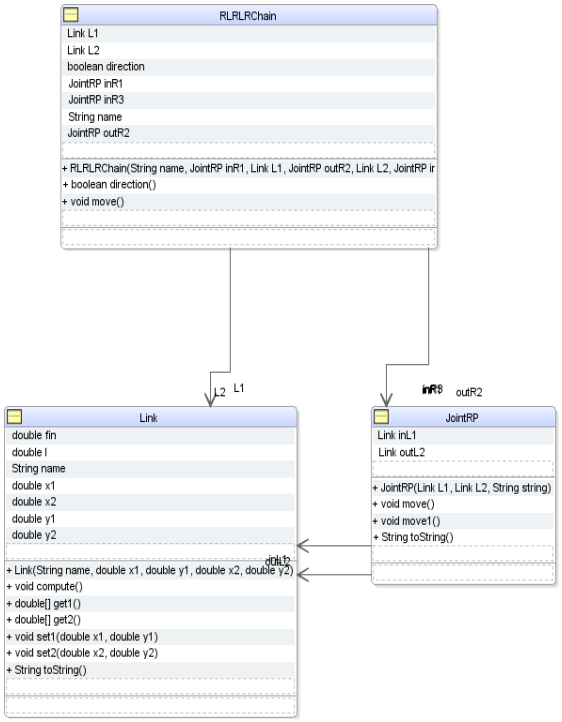


Fig. 4. - The RLRLR chain conceptual representation.

The Assur group in Figure 4 has a structure of two links (`L1` and `L2`) and three joints (`inR1`, `outR2`, `inR3`) that is internally rigid (has zero mobility on its own). Internally, this is solved using the law of cosines. From the quadratic equation, which yields two possible intersection points for the circles centered at A (radius `L1`) and C (radius `L2`), the correct one is selected. This specific group is a core component in the structure of many multi-loop planar mechanisms (or linkages). The last conceptual representation from Figure 5 is about a joint that is able to pick the state of a link (position and orientation of `Lin`) and transmit it to another link (`Lout`) from where can be collected to form the endpoint in trajectory or an input for some other element.

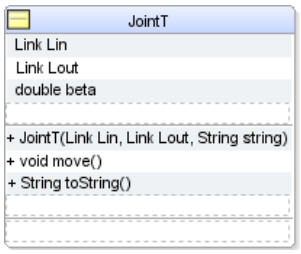


Fig. 5. - The transmitter joint conceptual representation

7. EXPERIMENTAL RESULTS AND CONCLUSIONS

A four-bar linkage mechanism can be modeled [4], [7] using these classes with the following code. The `Link` class is used to create the following objects: `G0`, `L1`, `L2`, `L3` and `L4`. The `G0` (fixed or reference link) and `L1` (moving link) are coupled using the `O` (of `JointRA` type) active joint that the connected link ends stay together [8,9]. Using these objects inside a constructor a new object `MO` (of `MotorChain` type) is created by composition. By moving the actuator in the `MO` from the start position 360^0 and with a 1^0 step the position of `L1` will change. This is connected to the `ABC` (of `RLRLRLChain` type) object composed by the `A`, `B`, `C` revolute passive joints and the `L2` and `L3` moving links. The endpoint of `L1` is the start point for `L2` and the endpoint of `L3` is fixed. The `FourBL` object has a transmitter joint object E attached to the `L4` link (of `JointT` type) link at the start of `L2`. Each time the `L2` updates, the `L4` object extracts from `L2` its position and angle, calculates its own world coordinates, and adds them to create a path followed by the endpoint of `L4`. The final `FourBL` object is created by the **composition** [10] of three other objects `MO`, `ABC` and `T`. Using OOP **encapsulation** hides the solving complexity. Consider the `ABC` object that determines the new positions of `B` when the positions of `A`, `C` and `L2` with `L3` are given. From the quadratic equation which provides two solutions for `B`, the right one is selected (closer to the previous `B`). **Inheritance** models joint types, `JonitRA` and `JointRP` inherit from a base `JointR` class. Each overrides methods like `move()` differently. **Polymorphism** enables uniform updates even though every joint runs its own constraint satisfaction logic. The links are objects that know geometry. The joints are objects that enforce constraints. The linkage is

the composite object that coordinates them. Motion propagates through method calls, and the system's behavior emerges from object collaboration. Figures 6 and 7 present a four-bar linkage mechanism and a simulation of the movement while Figure 8 to 12 present a five-bar mechanism simulated under different constraints.

```
public class FourBL {
 JointRA O;
 JointRP A, B, C;
 JointT E;
 Link G0,L1, L2, L3, L4;
 MotorChain MO;
 RLRLRChain ABC;

 public FourBL() {
//motor chain
  G0 = new Link("0 - fixed", 0, 0, 0, 0);
  L1 = new Link("1 - OA", 0, 0, -1, 3);
  O = new JointRA(G0, L1, "O", 360, 90);
  MO = new MotorChain(">> motor O <<", G0,
O, L1);
//RRR Assur group
  L2 = new Link("2 - AB", -1, 3, 10, 7);
  L3 = new Link("3 - BC", 10, 7, 10, -2);
  B = new JointRP(L2, L3, "B");
  A = new JointRP(L1, L2, "A");
  C = new JointRP(L3, G1, "C");
  ABC = new RLRLRChain(">> RRR <<", A, L2,
B, L3, C);
  //Output link L4
  L4 = new Link("4 - E'E", -1, 3, 0, 5);
  E = new JointT(L2, L4, "E");
 }

 public static void main(String a[]) {
  FourBL m1 = new FourBL();
  do {
   m1.MO.move();
   m1.A.move();
   m1.ABC.move();
   m1.E.move();
  } while (!m1.MO.isFinished());
 }
}
```

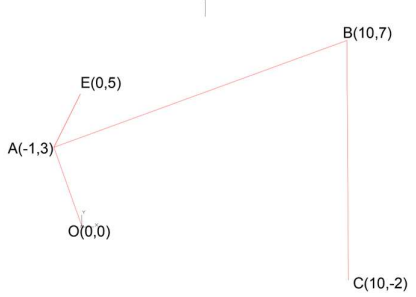


Fig. 6. - The four-bar linkage mechanism representation.

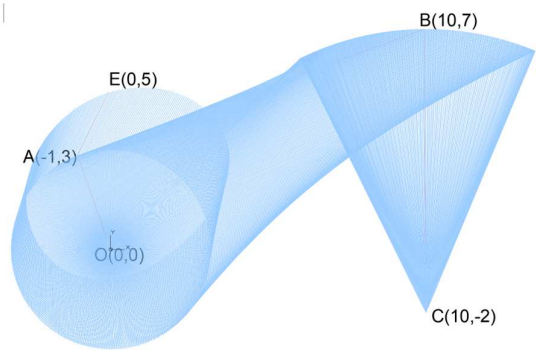


Fig. 7. - The four-bar linkage mechanism simulation.

A five-bar linkage mechanism like the one from Figure 8 can be modeled and simulated using the following model.

```
public class FiveBL {
 JointRA O, O1;
 JointRP A, B, C;
 JointT E;
 Link G0, G1, L1, L2, L3, L4, L5;
 MotorChain MO, MO1;
 RLRLRChain ABC;

 public FiveBL() {
//motor chain 0 (0,1) > (-1,3)
  G0 = new Link("0 - zero - ground", 0, 0,
0, 0);
  L1 = new Link("1 - OA", 0, 0, -1, 3);
  O = new JointRA(G0, L1, "O", 360, 0.5);
  MO = new MotorChain(">> motor O0 <<", G0,
O, L1);
//motor chain 1 (8,1) > (12,4)
  G1 = new Link("0 unu - fixed", 8, 1, 8,
1);
  L4 = new Link("4 - O1C", 8, 1, 12, 4);
  O1 = new JointRA(G1, L4, "O1", 360, 1);
  MO1 = new MotorChain(">> motor O1 <<",
G1, O1, L4);
//RRR Assur group(-1,3), (5,7), (12,4)
  L2 = new Link("2 - AB", -1, 3, 5, 7);
  L3 = new Link("3 - BC", 5, 7, 12, 4);
  B = new JointRP(L2, L3, "B");
  A = new JointRP(L1, L2, "A");
  C = new JointRP(L4, L3, "C");
  ABC = new RLRLRChain(">> RRR <<", A, L2,
B, L3, C);
//Output link L5
  L5 = new Link("5 - aE", -1, 3, 5, 10);
  E = new JointT(L2, L5, "E");
 }

 public static void main(String a[]) {
  FiveBL m1 = new FiveBL();
  do {
   m1.MO.move();
```

```
    m1.A.move();
    m1.MO1.move();
    m1.C.move1(); //L2(x2,y2)
    m1.ABC.move();
    m1.E.move();
   } while (!m1.MO.isFinished());
  }
}
```

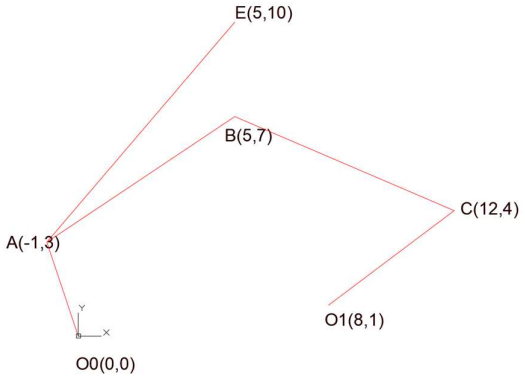


Fig. 8. - The five bar linkage mechanism representation with L5 fixed to L2.

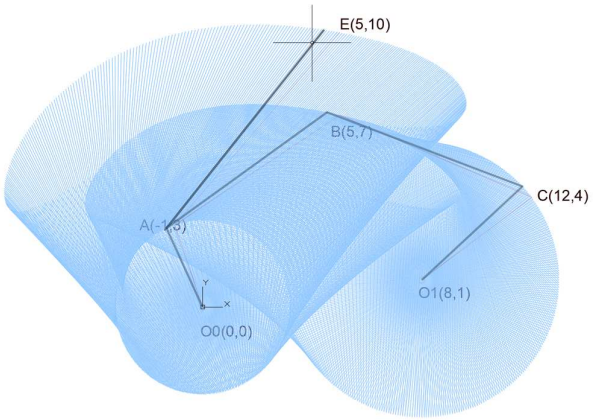


Fig. 9. - The five bar linkage mechanism simulation with L5 fixed to L2.

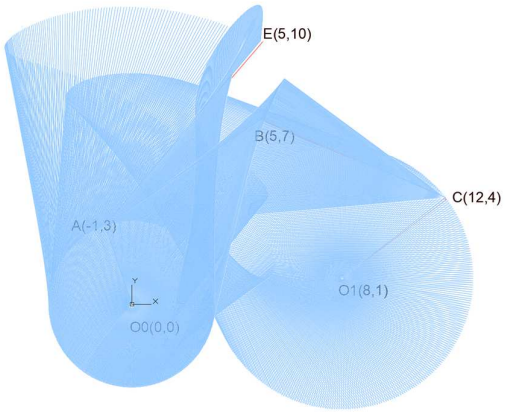


Fig. 10. - The five bar linkage mechanism simulation with distinct steps (`MO` with double step compared to `MO1`).

```
//Output link L5
  L5 = new Link("5 - BE", 5, 7, 5, 10);
  E = new JointT(L3, L5, "E");
```

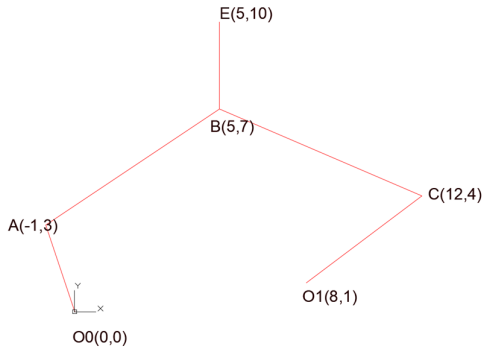


Fig. 11. - The five bar linkage mechanism representation with L5 fixed to L3.

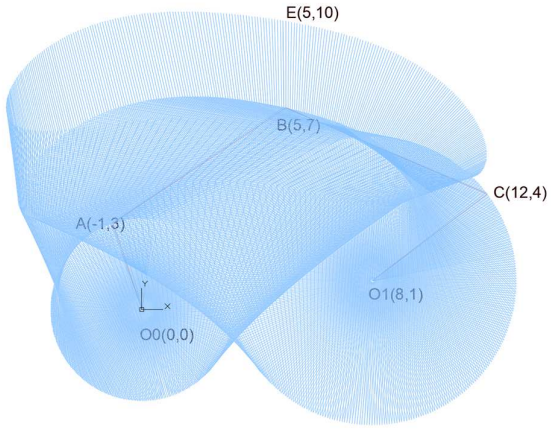


Fig. 12. - The five bar linkage mechanism simulation with L5 fixed to L3.

The OOP paradigm using OOD in Java enables modular, collaborative development where complex mechanical simulations can be built from independently developed, tested, and validated components - exactly mirroring how real mechanical systems are engineered from standardized parts.

8. ACKNOWLEDGMENT

This research was supported by the project New Frontiers in Adaptive Modular Robotics for Patient-centered Medical Rehabilitation - ASKLEPIOS and was funded by the European Union - NextGenerationEU and Romanian Government under the National Recovery and

Resilience Plan for Romania, contract no. 760071/23.05.2023 and code CF 121/15.11.2022, with the Romanian Ministry of Research, Innovation, and Digitalization within Component 9, Investment I8.

9. REFERENCES

[1] Antal, T.A., *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 Mathematics and Mechanics, ISSN 1221-5872, 53(3), pp. 481-486, Cluj-Napoca, 2010.

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

[5] Kecskeméthy, A., *Object-Oriented Modelling of Mechanical Systems.* In: Angeles, J., Kecskeméthy, A. (eds) Kinematics and Dynamics of Multi-Body Systems. CISM International Centre for Mechanical Sciences, vol 360. Springer, Vienna, 1995. https://doi.org/10.1007/978-3-7091-4362-9_6.

[6] Harry H. Cheng Æ Dung T. Trang, *Object-oriented interactive mechanism design and analysis*. Engineering with Computers (2006) 21: 237–246, DOI 10.1007/s00366-005-0008-4.

[7] 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.

[8] Pisla, D.; Birlescu, I.; Vaida, C.; Tucan, P.; Pisla, A.; Gherman, B.; Crisan, N.; Plitea, N. *Algebraic modeling of kinematics and singularities for a prostate biopsy parallel robot.* Proc. Rom. Acad. Ser. A 2019, *19*, 489–497.

[9] José M. Garrido, *Object Oriented Simulation*. Springer-Verlag US 2009, p. 431, ISBN 978-1-4419-0515-4.

[10] Reed, John A., AFJEH, Abdollah A. *An Object-Oriented Framework for Distributed Computational Simulation of Aerospace Propulsion Systems.* In: COOTS. 1998. pp. 149-164.

Proiectarea orientată pe obiecte aplicată în domeniul mechanismelor plane utilizând Java

Rezumat: Proiectarea orientată pe obiect și paradigma de programare orientată pe obiecte sunt domenii înrudite ce pot fi aplicate integrat pentru crearea de aplicații în care reutilizarea sau extinderea codului devine naturală și simplă. Lucrarea își propune să prezinte care sunt condițiile în care aceasta integrare este realizată cu succes în situația utilizării ca instrument de interfață AutoCAD-ul.

Tiberiu Alexandru ANTAL, Prof. Eng., PhD, Technical University of Cluj-Napoca, Department of Mechanical System Engineering, antaljr@mail.utcluj.ro, 0264-401667, 103-105, Muncii Blvd, , Cluj-Napoca.

Bogdan GHERMAN, Assoc. Professor, Eng., PhD, Technical University of Cluj-Napoca, Department of Mechanical System Engineering, Bogdan.Gherman@mep.utcluj.ro, 0264-401684, 103-105, Muncii Blvd, , Cluj-Napoca.