ChatGPT

Text-Based Mechanical Description Languages and Co-Coding for Mechanical Design

Executive Summary

Text-based mechanical design description languages are emerging as powerful complements to traditional CAD tools. They enable engineers to **programmatically define** mechanical structures, assemblies, and system behavior using code or scripts, analogous to how hardware description languages (HDLs) describe electronic circuits. This approach brings benefits in parametric precision, automation, and version control. While historically mechanical design has been dominated by interactive GUI-based CAD software, a growing ecosystem of **code-centric CAD tools** (e.g. OpenSCAD, CadQuery, Onshape FeatureScript) and **system modeling languages** (e.g. Modelica) has gained traction. These allow complex designs to be described with textual **commands, equations, and algorithms**. A key driver of renewed interest is the potential for **AI-assisted co-coding**: large language models (LLMs) and tools like GitHub Copilot can understand and generate code, making text-based design more accessible. Recent research and industry experiments show LLMs generating CAD scripts from natural language, auto-completing parametric models, and assisting in design space exploration. This report surveys the state-of-the-art in text-based mechanical design languages, comparing their capabilities to HDLs, highlighting current tools (open-source and commercial), and examining how they can be leveraged in co-coding with AI. We present case studies from both hobbyist maker communities and industry (e.g. automotive use of Modelica), analyze trends such as increased adoption of **programmatic CAD for customization** and **LLM-driven design**, and discuss challenges like the steep learning curve and integration with traditional workflows. Finally, we identify open research questions and future directions, and provide hands-on tutorials for two representative examples – **OpenSCAD** (scripted 3D CAD modeling) and **Modelica** (equation-based mechanical system simulation) – to help new users get started.

Historical & Conceptual Background

Mechanical design has traditionally been performed with drawings and interactive CAD software, but the concept of describing designs **textually** has deep roots. As early as the 1970s, researchers explored *Part and Assembly Description Languages* (PADL) for solids. For example, in 1978 Herb Voelcker developed PADL, a formal language for solid modeling that influenced early CAD systems [1]. This idea – capturing geometry in a **human-readable code format** – was ahead of its time. Mainstream CAD evolved with GUIs (e.g. Sketchpad in 1963 [2]), emphasizing visual interaction over textual programming. However, the rise of **parametric CAD** in the 1980s–1990s reintroduced programming concepts: users set up dimensional parameters and constraints that function akin to code variables. Still, the underlying model was typically stored in proprietary binary formats, hidden from the user [3] [4].

By the 2000s and 2010s, open-source communities began embracing *script-based CAD*. **OpenSCAD**, first released in 2010, exemplifies this approach: it reads a script file describing constructive solid geometry (CSG) operations and compiles it into a 3D model [5]. This approach gives the designer full control and parametric power – the design's "source code" can be edited, versioned, and reused. The concept is analogous to how a hardware engineer might write VHDL/Verilog for a circuit instead of drawing it. In both cases, a **textual source** is the master representation of the design intent.

The conceptual appeal of text-based mechanical design lies in **precision and repeatability**. A text description is unambiguous and captures design logic (loops, conditionals for variants, mathematical formulas for geometry) that would be cumbersome to manage through clicks in a GUI. It also means the designs can be **parametric templates** – easily adjusted by changing a few values – which fosters reuse. Another advantage is **transparency**: instead of an opaque binary CAD file, text can be read and understood (by humans or machines) to know exactly how a model is constructed [4] [6].

However, historically these benefits came with trade-offs. Text-based modeling was a niche skill, used mostly by programmers or researchers, while most mechanical engineers stuck to familiar interactive CAD tools. Unlike digital logic (where HDLs quickly became standard for complex circuit design), mechanical design involves continuous geometry and often organic shapes that seemed less amenable to coding. Only highly regular or parametric shapes (gears, enclosures, lattice structures, etc.) were practical to define in code. **Complex free-form surfaces**, by contrast, were easier to sculpt in a GUI. Thus, for decades, textual mechanical design lived on the fringes – used in CNC toolpath scripts, finite element model scripts (e.g. ANSYS APDL), or by hobbyists with OpenSCAD. A recent survey indicated that code-based CAD users represented only about *1%* of the overall CAD market [7].

Survey of Current Solutions

Today there is a rich landscape of text-based mechanical description languages and tools. They range from geometry modeling languages to system dynamics modeling languages:

- **OpenSCAD – "The Programmer's Solid 3D CAD Modeller":** An open-source scripting language and tool focused on *constructive solid geometry*. OpenSCAD uses its own simple language to define primitives (cubes, cylinders, etc.) and boolean operations (union, difference, intersection) to build models [8]. It is *declarative* in nature (you describe what to create, not how to drag it), and renders a 3D model from the code. OpenSCAD excels at parametric designs – you can easily change a few variables (width, height, number of holes, etc.) and recompile the model [9]. It is popular in the 3D printing and maker community for creating customizable parts. (MakerBot's Thingiverse platform even integrated a **Customizer** that exposes OpenSCAD parameters via a GUI, allowing users to tweak community-shared models without coding.) OpenSCAD's limitations are that it only produces static geometry (no simulation of motion) and uses a simplistic CSG approach (no native sketch constraints or complex fillets, though creative coding or libraries can approximate these). It requires a dedicated app (with GUI preview) to compile the script. Nonetheless, it's widely used for open hardware designs and teaching computational design.

- **CadQuery:** A Python-based CAD scripting framework that wraps the powerful OpenCASCADE geometry kernel. CadQuery provides a higher-level, *more design-intent-driven* API compared to OpenSCAD [10]. Instead of writing a custom language, designers write Python code using CadQuery's library: for example, one can start a workplane, extrude a profile, make holes, fillet edges, etc., all in code. The goal (as the documentation says) is a script that reads close to an English description of the object [11]. CadQuery allows relative placements and constraints (e.g. "cut this hole on face top-center") which preserve design intent, unlike OpenSCAD's purely coordinate-based approach [10]. It can export to standard CAD formats (STEP, STL) and has a growing ecosystem (e.g. a GUI editor, Jupyter notebook integration). Being Pythonic, it leverages a full programming language for loops, math, and logic. CadQuery is open-source and has gained popularity among engineers who want more power and familiarity (Python) in script-based CAD. It's considered an evolution of code-CAD and is increasingly used in tandem with AI since Python code generation by LLMs is quite mature [12] [13].

- **OpenJSCAD and others:** Similar to OpenSCAD but using JavaScript. OpenJSCAD runs in a web browser, allowing script-based modeling with JS syntax. There are also other language-specific tools like **ImplicitCAD** (Haskell-based, aiming to improve on OpenSCAD's kernel), **RapCAD**, and new experimental ones like **Build123d** (a Python CAD library offering an alternate paradigm to CadQuery). These are mostly open-source and cater to small communities. **Fornjot**, for example, is a young project implementing a CAD kernel in Rust, designed for code-first modeling – a sign of ongoing innovation in this space.

- **FeatureScript (Onshape):** Onshape, a cloud-based CAD platform, introduced FeatureScript as a built-in language to create custom parametric features. It's a domain-specific language in which users can program new feature types (like a custom gear generator or complex pattern) that integrate into Onshape's GUI. Unlike OpenSCAD, FeatureScript doesn't describe entire parts from scratch but rather *augments* interactive modeling with custom-coded building blocks. It is text-based and open to all Onshape users (the language is open, though the platform is proprietary). This is a unique example of a commercial CAD vendor embracing text descriptions: Onshape stores models in a database but allows full regeneration from FeatureScript code for custom features. Notably, FeatureScript is powerful enough that one could define entire parts via code if desired. It's been used to standardize designs and automate repetitive tasks in industry [14]. Onshape also exposes a general-purpose API, which developers (or AI) can use to create geometry via scripts from external applications [15] [16].

- **Modelica:** In mechanical design, not all problems are purely geometric. Modelica is a high-level *modeling language for multi-domain physical systems*. It allows engineers to describe **mechanical systems** (along with electrical, thermal, hydraulic, etc.) using equations and object-oriented component models. For instance, one can model a suspension by connecting masses, springs, and dampers, or model a robot arm's kinematics and control. Modelica is *acausal* and equation-based – you write differential equations and constraints that the solver will work with, rather than step-by-step simulation code. It's become a standard in system simulation, with open implementations like OpenModelica and commercial tools like Dassault Systemes Dymola, Wolfram SystemModeler, etc. Modelica does not produce 3D CAD geometry (though some tools visualize mechanisms), but it's crucial for mechanical design *behavior*. It's text-based (usually `.mo` files) and highly structured (with formal syntax for models, connectors, etc.). The **adoption in industry is significant**: companies like Audi, BMW, Ford, Toyota and even Formula 1 teams use Modelica to design and optimize vehicle systems [17]. It addresses the dynamic side of mechanical design which pure CAD does not – making it complementary. In context of co-coding, Modelica code could in principle be written with AI assistance (for example, suggesting equation forms or connecting standard library components).

- **Others:** There are various specialized languages worth noting. *VHDL-AMS* is an analog hardware description language that can describe mechanical analog behaviors (seldom used for complex mechanical design, though). **Simscape Language** (by MathWorks) is a text-based language to create custom components for Simscape, similar in spirit to Modelica (equation-based). In robotics, **URDF (Unified Robot Description Format)** is an XML-based language to describe a robot's links and joints – essentially a text format for a mechanical assembly's structure (though geometries are referenced, not defined algorithmically). **G-code** used in CNC machining is a low-level description of tool paths (not parametric modeling, but an important text format in manufacturing).

This survey shows two broad classes: **geometry modeling languages** (like OpenSCAD, CadQuery, FeatureScript) and **behavioral/system modeling languages** (like Modelica, Simscape). Both are text-

based but serve different purposes. Increasingly, projects aim to unify aspects of both – for example, generating geometry with embedded knowledge of physics or manufacturing.

Comparative Overview of Key Languages

To summarize, the following table compares some representative text-based mechanical design languages/tools:

Language/ Tool	Domain & Purpose	Paradigm (Design Approach)	Availability	AI Co-Coding Potential
OpenSCAD	3D solid modeling (CAD geometry)	CSG-based scripting (custom DSL)	Open-source app	Moderate – simple syntax, some LLM training data; Copilot can suggest code, used in text-to-CAD research [12].
CadQuery	3D CAD modeling (parametric features)	Python EDSL, design intent focus	Open-source lib	High – leverages Python code generation; LLMs fine-tuned to output CadQuery achieve high accuracy [12] [13].
FeatureScript	Parametric CAD features (Onshape CAD)	Declarative feature definitions	Proprietary cloud (Onshape)	Medium – not widely in LLM datasets, but Onshape's API used in AI experiments (e.g. OnshapeGPT) [18] [19].
Modelica	Mechanical system simulation (multi-domain)	Equation/object-oriented modeling	Open standard (tools: OpenModelica, Dymola, etc.)	Low/Medium – complex syntax, but possible for AI to assist in writing equations or connecting components (requires domain knowledge).
KCL (KittyCAD)	3D CAD modeling (full designs)	Python-derived DSL with constraints (Zoo's approach)	Proprietary (in development)	High – explicitly designed to leverage LLMs; text-based geometry intended for AI generation [20] [21].
ImplicitCAD	3D modeling (alternate kernel)	Haskell DSL (can also parse OpenSCAD)	Open-source	Low – niche language, minimal training data for AI.

Language/ Tool	Domain & Purpose	Paradigm (Design Approach)	Availability	AI Co-Coding Potential
Fornjot	3D modeling (code-first CAD in Rust)	Rust library (programmatic CAD)	Open-source (early stage)	Potential – future integration with code assistants as it matures.

Table: A comparison of various text-based mechanical design languages, illustrating their focus and suitability for AI-assisted coding.

Comparison with Hardware Description Languages (HDL)

It is instructive to compare mechanical description languages to the well-established **hardware description languages** used in electronics (VHDL, Verilog, etc.). HDLs and MDLs (mechanical description languages) share the idea of using text to formally describe a design, but the nature of what they describe differs fundamentally:

- **Abstraction Level:** HDLs describe digital circuits in terms of logic gates, state machines, and interconnections. These are discrete and have clear boolean/algebraic definitions. Mechanical languages describe shapes or physical behavior, which are continuous and geometric or differential in nature. For instance, an HDL might say *"wire output of flip-flop A to input of gate B"*, whereas a mechanical geometry script might say *"extrude this profile 10 mm and subtract a cylinder to make a hole"*. The mechanical equivalent of connecting components is either geometric assembly (constraining parts together) or physical connection (in a simulation sense). Modelica, for example, connects components via ports which is analogous to connecting circuit components – highlighting a similarity where mechanical systems *can* be described modularly like circuits when focusing on system dynamics.

- **Determinism and Execution:** HDLs can be executed/simulated deterministically and can be synthesized into physical hardware automatically. Mechanical design descriptions (geometry) are not *executed* in that sense – they are evaluated by a CAD kernel to produce a shape. There is no direct "compile to machine" equivalent; manufacturing a part from the description involves separate CAM (or 3D printing) processes. However, mechanical simulations (like those written in Modelica) *are* executed to produce time-domain behavior, akin to how an HDL is simulated. One big difference is that HDLs often include the notion of time and parallel events (especially for digital logic simulation), whereas a static CAD model has no time dimension. For dynamic mechanical models (Modelica), time and continuous integration are key, but those languages must handle continuous mathematics (differential equations) rather than event-driven logic.

- **Constraints vs. Logic:** Mechanical designs often involve geometric constraints (perpendicular, concentric, etc.) and numeric parameters. Traditional HDLs don't have an analog of geometric constraints – instead, they have logical conditions and timing constraints. In mechanical description, a major challenge is specifying constraints that should be solved to realize the design. For example, one might want to specify that a part fits within a volume or that two holes align – a mechanical DSL might need a solver to satisfy those constraints. Indeed, research languages like AIDL incorporate a constraint solver to assist LLMs in generating valid geometry [22] [23]. In HDLs, constraint solving isn't needed in the same way; the "solver" is essentially the synthesis tool mapping logic to gates.

- **Maturity and Adoption:** HDLs have been industry standard for decades in electronics – no chip is designed without them. They have robust toolchains (synthesis to silicon, formal verification, etc.). Mechanical text-based design is far less ubiquitous. Most mechanical engineers still use interactive CAD and would find writing code for a complex shape unusual. The adoption gap is huge: writing a 100,000-gate chip *requires* an HDL, but designing a car or airplane is still mostly done with interactive CAD, with textual methods used in niche areas (analysis scripts, generative design algorithms, etc.). That said, there is a trend to bring software-like methodologies into mechanical design for **automation and collaboration**. For instance, text files are version-controllable and diffable, which is a big plus for collaborative engineering: it's easier to track changes or merge contributions in code than in binary CAD files [24]. There is no mechanical equivalent of automatic synthesis (you can't "compile" a functional spec into an optimal mechanical design easily, aside from generative design optimization which is a different AI-based approach). This means mechanical coding often remains at the level of automating geometry creation, rather than declaring a high-level spec and letting the computer synthesize the design – a contrast to HDLs.

In summary, while **HDLs provided a blueprint** for how powerful a textual design language can be, mechanical design languages must contend with geometry's complexity and the need for continuous math. They are evolving to incorporate solver aids, as seen with research into DSLs like AIDL that attempt to mimic the success of HDLs by formalizing mechanical design for AI generation [25]. The aspiration is that mechanical design might one day have its analog of "logic synthesis" – e.g. an AI that takes a set of requirements and outputs a CAD model – and having a textual representation is a step in that direction.

Capabilities for Co-Coding with AI

One of the most exciting aspects of text-based mechanical design is how naturally it can interface with **AI coding assistants**. Large Language Models like GPT-4 (as used in GitHub Copilot, Cursor, etc.) have been trained on vast amounts of code and text. When mechanical designs are represented in code, we essentially *enable LLMs to work with them*. This is far easier than trying to have an AI manipulate a binary CAD file or a complex GUI. As one developer of a new CAD language noted, *"by representing geometry as text, KCL can reuse the rapidly-growing ecosystem of machine text tools – LLMs, summarizers, AI coding assistants, etc. LLMs can generate KCL from a human prompt, explain the purpose behind some line of code, or improve your KCL code for you"* [20] [21]. In other words, once design = code, all the advances in AI for code can be applied to design.

Co-coding in this context means an AI assistive partner that can help write or refine mechanical design code. There are several capabilities and emerging examples:

- **Text-to-CAD generation:** Researchers have directly prompted LLMs to create CAD models from natural language descriptions. One study had GPT-4 generate OpenJSCAD (JavaScript CAD) code when asked for a furniture piece – e.g. "design a simple cabinet with a shelf" – and the AI produced a script with reasonable structure, including meaningful variable names and comments [26]. The AI even made educated guesses for dimensions not specified, demonstrating *spatial reasoning* to some extent [27]. Another project, *Text2CadQuery*, fine-tuned open-source LLMs specifically to output CadQuery code. By training on pairs of text prompts and CadQuery scripts, they achieved about **69% exact match** on a test set, a notable jump from prior approaches [12] [28]. The advantage was leveraging the LLM's strength in Python to directly generate the CAD code, avoiding intermediate formats [29] [13]. These experiments show that AI

can *propose initial designs in code form*. A human can then review and iterate on that code (potentially with further AI help).

- **Interactive CAD assistants:** Rather than one-shot generation, an AI co-coder can engage in a loop: the engineer gives a command or question in natural language, the AI produces code or suggestions, the engineer (or CAD system) executes it and provides feedback (via geometry preview or error messages), and the loop continues. There have been prototypes of this in both academia and industry. For example, a student project integrated ChatGPT with Onshape's API – the user could describe a mechanism (like "Newton's cradle with 5 balls and an A-frame") and the AI chatbot generated Python code to call Onshape's API and create that geometry [18] [16]. While the geometries were simple, it proved the concept that an AI can drive a modern CAD system through code. The AI essentially wrote the "glue code" to use Onshape's feature library, which is something Copilot or Cursor could also assist with for users writing such scripts.

- **AI-assisted parametric exploration:** LLMs can help modify and explore variations of an existing coded design. Because the design parameters and logic are explicit in text, an AI can be instructed to "make the part taller and add a pattern of holes" and it could edit the code accordingly [30]. This is like having a conversational partner who understands CAD code. For instance, one could have a GPT-based assistant for OpenSCAD where you say "increase the diameter of the holes and array 5 of them instead of 3" and it outputs the modified code snippet. Early results show LLMs can handle such transformations in a chat setting, effectively acting as a CAD co-pilot that accepts high-level instructions and translates them to code [27] [30].

- **Combining formal grammars with AI:** A key challenge is ensuring AI-generated code is **valid and error-free**. CAD languages can have strict requirements (e.g. solids must be manifold, constraints solvable). Researchers have begun constraining LLM outputs with formal grammar or DSLs to improve reliability [31] [25]. For example, MIT's AIDL introduced a hierarchical language that an LLM can output, which then relies on a geometry solver to fill in exact placements [22] [32]. By offloading heavy constraint solving to a backend, the LLM is guided to focus on high-level structure (like "place a hole centered on face A and align part B to it"). In co-coding terms, this means the AI and the language are co-designed: the AI writes in a helper language that a second tool interprets, catching mistakes and refining geometry. This approach is promising for complex designs where pure end-to-end generation would often fail on geometry details. In general, the use of **domain-specific languages or APIs** can funnel the AI's suggestions into safe channels. We see this in practice with tools like *PartCAD* (an extension to VS Code) which provides an AI "Generate CAD model" command supporting OpenSCAD or CadQuery [33] [34]. Under the hood it likely uses prompt engineering and perhaps templating to ensure the AI's code will run.

- **Code completion and error fixing:** Even without custom integrations, using GitHub Copilot in an editor while writing mechanical code can be helpful. For popular languages (Python for CadQuery, or even OpenSCAD which has some presence on GitHub), Copilot may suggest code as you type. It might auto-complete a loop to place multiple holes, or recall syntax for a cylinder in OpenSCAD. This speeds up coding and lowers the barrier for beginners. Furthermore, if the code fails (say the model doesn't compile), an AI can help debug. For example, if an OpenSCAD script has a misplaced bracket or a non-manifold operation, an assistant could analyze the error and suggest a fix in code. This is analogous to Copilot helping with a Python bug – except the "bug" might be a geometric inconsistency.

It's important to note that while AI co-coding is powerful, it's not flawless. LLMs sometimes **lack deep spatial understanding** – they might place a part floating in space or make two parts intersect

unintentionally. They also have limited knowledge of physics or manufacturability unless specifically trained. That said, the rapid improvement in multimodal models hints that future AIs might directly interpret 3D geometry along with code. Even now, tools like the aforementioned PartCAD are exploring using both text and image feedback (the Reddit post mentioned using Gemini and ChatGPT with interactive model previews [33] [34]). The AI could "see" the rendered model and adjust its code, a sort of vision-language feedback loop.

In summary, the **capabilities for co-coding with AI** in mechanical design are expanding: from generating initial design code from scratch, to assisting in incremental edits, to validating and optimizing designs. The fact that mechanical designs can be expressed in text is what makes this possible – as one commentary put it, *"LLMs are far ahead in reading/writing text than specialized CAD formats, so text representation ironically lets AI understand geometry better"* [20] [21]. The synergy of text-based design and AI is driving a resurgence of interest in these tools.

Use Cases & Industry Adoption

While still an emerging approach, text-based mechanical design and co-coding with AI are finding use in various domains:

- **Maker Community & Mass Customization:** Hobbyist designers and the 3D printing community have widely adopted OpenSCAD for making customizable models. On platforms like Thingiverse and Printables, one can find countless OpenSCAD files for things like enclosures, brackets, puzzles, or parametric gadgets. A user can open the code and tweak parameters to get a personalized variant (or use a Customizer UI to do so). For example, customizable **mounting brackets** allow changing the angle, number of screw holes, dimensions, etc., through parameters – something practically implemented with OpenSCAD scripts shared online. This use case democratizes design: even without deep CAD knowledge, one can adjust a few numbers in code and print a part that fits their needs. In education, this teaches programming concepts alongside geometry.

- **Open-Source Hardware Projects:** Some open hardware projects use script-based designs to ensure reproducibility. For instance, an open-source robotic hand or a microscope might be designed in OpenSCAD or CadQuery so that others can easily alter dimensions (for different hardware or adjustments) and regenerate all parts. This is akin to how open-source software can be compiled for different platforms – open-source *hardware* can be "compiled" (via CAD scripts) for different specifications. It also means any contributor can propose improvements by editing the code and sharing a git patch. Version controlling a text design is far cleaner than comparing binary CAD files. We see this in projects like the OpenSCAD MCAD library (parametric common parts) [35] and community-driven designs of machine elements.

- **Industrial Automation and Configurators:** Some companies have internal tools or scripts for automating repetitive design tasks. For example, a company making custom enclosures might maintain a CadQuery or FeatureScript template where sales engineers input client requirements (board size, connector locations) and the script generates a tailored 3D model. This reduces manual CAD work for each order. Similarly, in industrial machinery, engineers sometimes use scripting (e.g. Python with FreeCAD's API or MATLAB scripts with CAD APIs) to auto-generate designs like gear sets, piping layouts, or cable harness routes that follow algorithmic patterns. These text-based solutions might not be publicly visible, but they are essentially domain-specific mechanical description languages in action. The adoption is typically driven by need for

standardization and speed – once a script is written, generating variants is trivial, whereas doing each variant by hand would be error-prone and slow.

- **Automotive and Aerospace (System Modeling):** As noted, Modelica is heavily used in automotive for modeling vehicle dynamics, engines, cooling systems, etc. [17]. Engineers write models that capture the mechanical and control aspects of, say, a hybrid powertrain, and simulate performance. This **reduces physical prototyping** and helps in design decisions. Companies like Daimler created entire Modelica-based environments for developing control software via simulation [36]. In aerospace, Modelica and similar tools are used for aircraft energy systems, environmental control systems, and more. The adoption here is because no single CAD model can capture how a complex mechanical system behaves – one needs a *textual simulation model*. The co-simulation of mechanical and electrical (and software) is crucial in modern products, and Modelica shines there. While this is not "co-coding with AI" yet, one can foresee AI helping to suggest model structures or tune parameters (some researchers have explored using AI to search a Modelica model space for optimal designs).

- **AI-Assisted Design Prototypes:** We have examples like the OnshapeGPT prototype where an intern showed feasibility of AI-driven geometry creation in a professional CAD tool [18] [19]. While that was exploratory, major CAD companies are certainly investigating AI. PTC (which owns Onshape) is looking at AI plugins; Autodesk has demonstrated generative design (though typically topology optimization via simulation, not language) and recently there are startups (e.g. the company behind the "Zoo" Text-to-CAD approach) focusing on letting users talk or prompt and get designs. We can consider these as case studies in *augmenting professional workflows*: a designer could speed up initial concept generation by asking an AI for a starting point in code, then refine it using traditional tools. This hybrid approach may increase over the next few years.

- **Libraries and Standardization:** Another form of adoption is building **standard libraries of code-based components**. For instance, OpenSCAD's MCAD and BOSL libraries provide ready-made functions for screws, gears, bearings, etc., that can be included in scripts. This is analogous to electrical engineers using standard IP blocks. It indicates a maturing ecosystem where common mechanical elements are abstracted in code for reuse. On the system side, the Modelica Standard Library provides tons of mechanical components (springs, bodies, joints, sensors) that users drag/drop or textually connect, showing that a textual representation can be rich and user-friendly when packaged well.

Despite these use cases, challenges remain that temper adoption (discussed next). Not every domain finds text-based design the best fit. For example, artistic and ergonomic design (car body styling, consumer product aesthetics) still rely on sculptural CAD techniques. Also, many mechanical engineers are not trained in programming, so there is a cultural and skill gap – which ironically AI assistants might help bridge by generating code for non-coders.

Challenges & Research Gaps

Several challenges have limited the widespread adoption of textual mechanical design, and they present research opportunities:

- **Learning Curve and Mindset:** Many engineers find it non-intuitive to design via code. Spatial thinking doesn't always translate easily to lines of text. The immediate visual feedback one gets by dragging geometry in a CAD program is absent when writing code (though live preview windows partially address this). Overcoming this requires better *interactive coding environments* –

for example, live parametric previews, or bidirectional editing (modify either the code or the 3D view and keep them in sync). Research into *interactive programming for CAD* aims to merge the benefits of code and GUI [37]. Lowering the barrier might also involve educational efforts: teaching computational thinking alongside CAD in engineering curricula.

- **Visual Feedback & Debugging:** In code-CAD, if the model comes out wrong, debugging can be difficult. Error messages might be sparse (e.g. "CGAL error: computation failed" in OpenSCAD is infamous). Unlike software, where you can log values or use a debugger, diagnosing a geometric error requires visualization. One research gap is better **debugging tools for CAD scripts** – perhaps stepping through the construction sequence, or automatically isolating the part of code causing a non-manifold result. AI could assist here, by analyzing the code and the resulting geometry, but that's cutting-edge territory.

- **Performance and Scalability:** Complex mechanical models (with thousands of features or parts) can be computationally heavy to generate via script. OpenSCAD, for instance, can bog down with many boolean operations. CAD kernels like OpenCASCADE (used by CadQuery) handle complexity better, but then code may become very lengthy for large assemblies. How to manage **large-scale designs in text**? One area for improvement is enabling *modular design in code* – e.g. break a project into multiple files, have clear assembly constraints specified textually, etc. In HDLs, hierarchical design is standard; mechanical design could borrow that (and indeed, languages like AIDL emphasize hierarchical decomposition [38] [39]). There is room to research languages that handle assemblies elegantly, not just single parts.

- **Geometry Constraints & Solving:** Traditional CAD excels at constraint-based design: you sketch a few lines and add constraints, and a solver finds the exact geometry. Textual languages like OpenSCAD don't have constraint solvers; you must compute coordinates manually or via math. This makes some designs harder to script, especially if the geometry is defined implicitly (e.g. "fit a curve through these points"). A major research area is embedding *constraint solving capabilities into mechanical DSLs*. Efforts like AIDL that allow referencing geometry and adding constraints that a solver resolves are a start [32] [40]. In practice, one might want a language where you can write something like "place bolt holes equally spaced on this curved surface" without calculating each position by hand – the system would solve that. Integrating a solver also ties into AI: an AI could propose a design where some parameters are unspecified and let the solver fill them in optimally.

- **Integration with Existing Workflows:** Engineers won't abandon their trusted CAD platforms overnight. Research and development are needed on *hybrid workflows* where code and GUI coexist. Onshape's approach with FeatureScript is one example; another could be plugins for SolidWorks or Fusion 360 that allow generating features from scripts. Ensuring that changes in code update the 3D model and vice versa (round-tripping) is challenging. There's also the question of data exchange: how to convert a manually created CAD model into a textual form for further editing? Reverse-engineering a binary model to code might be possible only in limited cases. Thus, practical adoption may rely on companies providing official support or export to textual representations (e.g. an "export to FeatureScript" or "to Python script" function).

- **Validation and Verification:** For critical designs (like aerospace components), verifying that a design meets requirements is vital. Textual representation could enable formal verification techniques (as in software). Early work in this direction could involve checking that a generated design code conforms to design rules (geometry constraints, strength criteria, etc.). This is partly an AI problem (interpreting requirements and checking code) and partly a formal methods

problem. We might see the development of *design rule languages* that go alongside the design code.

- **Lack of Rich Dataset for AI Training:** While AI shows promise, one gap is the scarcity of large public datasets of text-to-CAD pairs. Unlike software where billions of lines of code are available, CAD models with corresponding scripts are fewer. The Text2CadQuery paper had to augment a dataset with 170k instances by generating them [41] [42]. As a community, encouraging sharing of parametric models and their descriptions will help. Perhaps repositories of CAD scripts (on GitHub or specialized sites) need to grow, which might happen as interest grows. Additionally, multimodal datasets that include 3D geometry alongside code are needed to train AI that truly "understands" the shape it's coding.

- **Handling of Complex/Organic Geometry:** Many mechanical parts are not just primitive-based solids; they have free-form surfaces (for aerodynamics, ergonomics, biomimicry). Describing a car hood or a turbine blade in text is extremely complex if done manually. Research could explore high-level textual primitives for such shapes (e.g. "NURBS surface through these profiles"), or AI techniques that integrate shape synthesis with code (perhaps an AI could generate a coarse code structure and a mesh for the free-form part). Until this gap is closed, textual design will complement, not fully replace, traditional modeling for these cases.

- **User Interface and Experience:** There's an opportunity to design better UIs for text-based design. For example, live sliders bound to variables in code (tuning a parameter and seeing update), or block-based coding (for those not comfortable with text, maybe a Scratch-like interface that outputs code). Also, collaborative editing (multiple engineers working on the same design code) introduces needs for locking and merging that are well-known in software dev but new to CAD teams.

In summary, while **text-based mechanical design has clear benefits, it faces challenges in usability, capability (constraints, complex forms), and integration**. These challenges are active areas of research and development. We see academia tackling some (like solver-aided design languages, AI integration), and industry tackling others (like better UIs and partial adoption via features and APIs). Overcoming these gaps will likely involve interdisciplinary effort – mechanical engineering, computer science, HCI, and AI all contributing to the next generation of CAD tools.

Future Directions

Looking ahead, the convergence of trends suggests that text-based design and AI will play a much larger role in the future of mechanical engineering. Some anticipated future directions include:

- **CAD Co-Pilots as Standard Tools:** Just as code developers now often have AI autocomplete and chat assistants, we can expect mechanical designers to have AI co-pilots within their CAD software. These assistants might accept natural language queries ("Make this part 20% lighter but keep stiffness") and respond by adjusting the parametric model or suggesting topology optimizations. The textual representation acts as the intermediary – the AI might edit the underlying script or feature tree to achieve the goal. By 2025, we're already seeing prototypes; by 2030 this could be a mature feature in leading CAD suites.

- **Unified Design Representations:** We may see efforts to unify geometric and behavioral design in one textual representation. For instance, a **Mechanical Description Language (MDL)** could describe not only the shape of a part but also its material properties, intended loads, and even

testing procedures, all in one code file. This would facilitate using the same source for CAD modeling, FEA simulation, and maybe CAM instructions (with appropriate tool-specific generators). It's analogous to how a hardware designer might simulate a circuit and also generate its layout from the same HDL source (with different tool flows). Early versions might simply link together formats (embedding a Modelica model in a CAD script file for instance), but eventually maybe something more integrated could emerge.

- **Open-Source CAD Platforms:** The success of languages like Python in AI was partly due to open ecosystems. In CAD, historically proprietary systems dominated. Future might bring open-source CAD kernels and languages (e.g. perhaps a fully open "Reference CAD Language" standardized by groups like the Modelica Association but for geometry). KittyCAD's KCL indicates a push in that direction, although it's currently proprietary, it shows a model for storing models as text in a cloud platform [4] [6]. An open standard text format for CAD models (beyond limited ones like STEP which is not human-friendly) could revolutionize collaboration – imagine if any CAD model could be opened as code in a text editor or manipulated via scripts, regardless of the originating software.

- **AI-Driven Optimization and Generative Design:** In the future, specifying the problem might be more important than specifying the solution. For example, instead of coding the exact shape, an engineer of 2035 might write a high-level script: "objective: minimize weight, constraint: support 100N load, space: this bounding box, material: ABS plastic" and an AI/solver system will iterate and produce a design (potentially described in a modifiable language format for further tweaking). This is an evolution of today's generative design, but more accessible via text prompts and code-like constraint definitions. The human can then refine the result by editing the constraints or adding additional features via code. We see glimpses of this as researchers integrate formal grammars (for the AI to respect) [31] [25] and let the AI's creativity handle the rest. The language in this scenario becomes less about specifying geometry directly and more about specifying *rules and goals*, which is an exciting frontier.

- **Enhanced Collaboration and Version Control:** As more of the design gets captured in text, mechanical engineering teams might adopt software-like collaboration models. We could see widespread use of git for versioning CAD code, automated CI pipelines that regenerate models and run simulations/tests whenever code changes (ensuring a change doesn't break a downstream assembly or analysis). In the future, a **mechanical DevOps** could emerge – repositories of parts with code, continuous integration building them, maybe even continuous fabrication (with automated printing or machining triggered by repository updates!). This level of integration demands robust text-based representations and error handling.

- **Education and Democratization:** Future mechanical engineers may be expected to code as part of their skill set. Conversely, programmers may find it easier to contribute to hardware design via these languages. This cross-pollination could democratize product development – for instance, a software developer with an idea for a gadget could script a prototype design and have it made, without needing years of CAD training. As educational institutions incorporate computational design into the curriculum, we'll see a generation of engineers comfortable moving between solid modeling and scripting. This will naturally push industry to accommodate their workflows, possibly accelerating the adoption of text-based methods.

- **Standardization and Interoperability:** Bodies like ISO or ASME might define standards for parametric design exchange. Just as STEP and IGES were standards for exchanging geometry, we might get a "STEP-Code" format that allows exchanging the procedural history of a model in a neutral way. If one company shares a design with another, they might send the code which can

be built in the recipient's environment. This would mark a big change from today's reliance on dumb solid exports. Achieving this requires consensus on languages or at least a common intermediate representation. The OpenCASCADE community, or the Modelica community, might be drivers here, since they already focus on openness and standards.

- **Integration with manufacturing and IoT:** With Industry 4.0, there's talk of "digital twins." A text-based design that is directly connected to manufacturing parameters could allow dynamic updates – e.g. an IoT sensor on a machine detects a change, updates a parameter in the design code, and the production prints a slightly adjusted part to compensate. It's speculative, but having designs in code opens possibilities for automation across the product lifecycle (production scripts adjust design scripts on the fly).

Ultimately, the future likely holds a **blend of human creativity and AI automation** in design. Text-based description languages will serve as the contract between human intent and machine execution of design. They make design knowledge explicit and transferable. As one Medium article mused, this could "democratize 3D design by letting engineers converse with CAD tools in plain English" [43] [44] – the code is the bridge between plain English and precise CAD. With advancing AI, that bridge will be increasingly seamless.

Case Studies

Let's look at a few case studies illustrating the concepts discussed:

Case Study 1: Parametric Shelf Bracket – From Niche Code to Widely Shared Design – A designer creates a parametric shelf bracket using OpenSCAD and shares it on a community site. The bracket's code allows adjusting side lengths, angles, and number of mounting holes. Initially, this might seem like overkill to code something a CAD user could draw in minutes. However, by coding it, the designer unlocked *mass customization*: hundreds of users have downloaded the OpenSCAD file (or used Thingiverse's Customizer) to generate brackets of various sizes for their specific needs (e.g. different shelf widths) without needing to redesign from scratch. The parametric model ensures that as long as the input parameters are reasonable, the bracket will generate correctly – hole spacing is auto-computed, material thickness is consistent, etc. This reliability and flexibility are a direct outcome of the textual description. Furthermore, when a user requested a new feature (like a center reinforcement rib), the original designer (or even another community member) could modify the code and re-share it, improving the design for all. This case highlights how text-based design can enable collaborative improvement and adaptation of a mechanical part.

Case Study 2: Automotive Manufacturer's System Modeling – An automotive R&D team needs to design a new electric vehicle's thermal management system. They use Modelica to model the cooling circuit (pumps, radiators, battery, motor) as well as the cabin HVAC. The entire vehicle's thermal network, including mechanical components like fans and valves, is coded in Modelica. This textual model is then simulated to evaluate temperatures, flows, etc., under different conditions. One big advantage they found is that once the text model is verified, it becomes a *single source of truth* for multiple departments: controls engineers use it to develop their control logic, mechanical engineers use it to size the radiators and decide placement (coupling with a CAD layout), and system engineers use it to run what-if scenarios (like failure of a pump). The text-based approach meant they could version control the model, compare changes (say, "what changed between v1 and v2 of the cooling system?" via a diff), and even generate documentation from the code comments. When AI became available, they experimented with an assistant to fine-tune parameters to meet performance goals (by wrapping an optimization around the Modelica simulations). This case demonstrates the power of a textual

description in a complex, multi-disciplinary mechanical system – it improved clarity, collaboration, and allowed advanced tools (like AI optimization) to be applied.

Case Study 3: AI-Driven Prosthetic Hand Design – In a research project, a team aimed to rapidly iterate designs of a prosthetic hand. They defined the hand's finger link dimensions and joint angles in a CadQuery script, along with parameters for the grip size. Using an AI co-pilot (GPT-4 integrated in VS Code), a biomechanics expert who wasn't a programming guru managed to adjust the design by simply stating goals: "make the thumb opposable at 60 degrees" or "ensure the palm can accommodate a cylindrical object of 50mm diameter." The AI translated some of these requests into code changes in the CadQuery script (like repositioning the thumb's base and adjusting finger lengths). Not all AI suggestions were perfect, but it accelerated the non-programmer's ability to refine the model. Additionally, because the design was in code, they set up a script to generate variants (e.g. different hand sizes) automatically for different patient profiles. This case underscores the potential for AI + code to empower domain experts who are not CAD specialists, and how coding a design makes automation and personalization far easier.

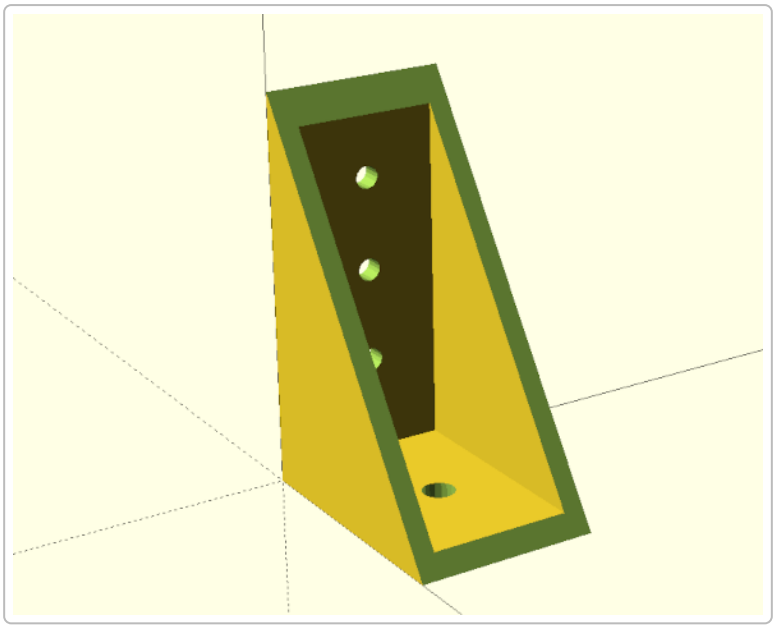
Each of these case studies reflects a facet of the text-based approach: community sharing and customization (bracket), cross-domain integration and rigorous development (vehicle system), and human-AI collaboration in creative design (prosthetic hand). They show that even though text-based design may not yet be the mainstream default, it can provide significant advantages in the right contexts.

Practical Examples & Onboarding

To make these concepts concrete, let's walk through two beginner-friendly examples: one using **OpenSCAD** for geometry modeling and one using **Modelica** for system simulation. These will illustrate how to get started with text-based mechanical design, step by step.

Example 1: Script-Based CAD Modeling with OpenSCAD

Imagine we want to design a simple **mounting bracket** – essentially an L-shaped bracket with holes for screws. We'll create a parametric bracket where we can easily adjust dimensions and the number of holes. This example will introduce basic OpenSCAD syntax and workflow.



An example of a parametric L-bracket created in OpenSCAD (rendered view). The bracket's dimensions and hole placements are driven by variables in the code, allowing easy modifications.

Step 1: Install OpenSCAD. Download and install OpenSCAD from the official website (available for Windows, Mac, Linux). It's a lightweight program. When you open it, you'll see a code editor on the left and a 3D view on the right (which will be blank until we compile some code). Ensure **Design → Automatic Reload and Compile** is enabled, so the 3D view updates whenever you save changes [45].

Step 2: Define parameters. In OpenSCAD, we start by defining top-level variables for key parameters. OpenSCAD units are generic (often assumed as millimeters by convention). Let's define our bracket's primary sizes and the hole specs at the top of the script:

```
// Bracket parameters
bracket_height = 60;  // height of the upright section
bracket_depth = 40;  // depth of the base section
bracket_thickness = 5; // thickness of the material
hole_diameter = 5;   // diameter of screw holes
num_holes = 3;     // number of holes on the upright
```

Each line ends with a semicolon. The `//` denotes a comment. We have a height and depth for the two legs of the L, a thickness, and we plan to put `num_holes` holes on the upright part (for screws). We also define hole size.

Step 3: Construct the solid geometry. We will build the bracket in two parts: the upright plate and the base plate, then join them (since it's an L shape). In OpenSCAD, you create primitives like cubes or cylinders and use transformations to position them. Finally, you can **union** them together.

Let's create the upright plate using a rectangular solid (cube) and then subtract cylindrical holes from it using **difference()** operation:

```
upright_length = bracket_height;       // just for clarity
upright_width = 20;              // width of the bracket (out of plane)
base_length  = bracket_depth;
base_width   = upright_width;

module bracket_upright() {
  // Create upright rectangular plate
  cube([upright_length, upright_width, bracket_thickness], center=false);

  // Subtract holes if any
  for(i = [1:num_holes]) {
    translate([ (upright_length/(num_holes+1)) * i, upright_width/2, -0.1 ])
      cylinder(d=hole_diameter, h = bracket_thickness + 0.2, center=true);
      // each cylinder is positioned through the plate thickness
  }
}

module bracket_base() {
  cube([base_length, base_width, bracket_thickness], center=false);
  // (we could also add holes to the base if needed, similarly)
}
```

Here we defined two modules: `bracket_upright()` and `bracket_base()`, which encapsulate the creation of each sub-part. Inside `bracket_upright`, we first make a cube of size `[length, width, thickness]`. In OpenSCAD, by default the cube's one corner is at the origin (0,0,0) if `center=false`. We then do a **for-loop** to create holes. The loop variable `i` goes from 1 to `num_holes`. For each, we position a cylinder. The `translate` moves the cylinder to a computed location: along the length of the upright, we divide it into equal intervals using `(upright_length/(num_holes+1)) * i` (this spaces the holes evenly along the height) [46] [47]. We put the hole in the middle of the width (`upright_width/2`). We also translate it slightly in Z (`-0.1`) so that the cylinder goes through the plate – we gave the cylinder a height slightly larger than the thickness (with center at its middle) to ensure it fully cuts through (this trick avoids exact flush edges that sometimes cause rendering issues).

The base module is simpler, just a cube (we could similarly add a hole or two in the base if we wanted, for screwing the bracket down, but let's keep it simple).

Step 4: Assemble the bracket. Now we need to position the upright and base relative to each other and join them. The bracket will have the upright standing on one end of the base. Suppose we align them such that the upright's bottom is flush with the base's top, at one end. We can do:

```
// Assemble upright and base
union() {
  // upright: standing upright, base lying flat
  bracket_upright();
  translate([0, 0, bracket_thickness])
    rotate([0, 90, 0])
```

```
    bracket_base();
}
```

We wrap with `union()` so the two solids are merged (though merely placing them without a difference or so would also merge because OpenSCAD defaults to union of sequential shapes at top level). We place `bracket_base()` by first translating it by `[0,0,bracket_thickness]` – this lifts it up so that its top surface aligns with the bottom of the upright (because the upright was built at Z=0 of thickness height). Then we rotate it 90 degrees around the Y-axis to make it lie perpendicular (an L shape). We might also need to translate it along X so that the base extends under the upright. In this code, since the upright was at origin spanning in +X direction, the base, after rotation, will also extend in +X from origin, overlapping with the upright's lower edge – that's actually what we want: an L where one leg's end touches the other's side. If we needed to adjust, we'd use an additional translate.

Step 5: Preview and render. If you copy all the above code into OpenSCAD's editor and hit F5 (preview), you should see an L-bracket in the 3D view. It might appear in a single color (yellow by default). You can orbit the view to inspect. If everything looks good, hit F6 (render) which does the full computation (needed before exporting STL for example). The result is our bracket model.

Try changing `num_holes` or `bracket_height` at the top and hit F5 again – the model updates parametrically. If `num_holes=0`, our loop will not run and no holes will be made (leaving a solid plate). If you set `num_holes=5`, you'll see 5 holes appear along the upright.

Onboarding tips: The OpenSCAD cheat sheet [48] is a handy reference for syntax. Common pitfalls for beginners include missing semicolons and forgetting that OpenSCAD uses *coordinates in mm by default* (so if something is off by a factor of 10, check units). Also note that OpenSCAD doesn't allow variables to be reassigned; they are more like constants (if you try to do `x=5; x=6;` it will ignore the second assignment). In our approach, we used modules to encapsulate parts – this helps manage complexity as your designs get bigger, and you can instantiate modules multiple times for repeated parts.

The bracket we made is simplistic (sharp edges, etc.), but one could refine it (e.g., fillet the edges – though OpenSCAD has no native fillet, one can approximate or use libraries). Nonetheless, this example demonstrates the core workflow: define parameters, build geometry with code, and adjust as needed. It showcases the **full control** you have – want 100 holes? just change a number; want it twice as thick? change a number; the code logic takes care of placement etc.

You can save the file as `bracket.scad` and generate an STL by clicking *File → Export → STL* after rendering. That STL can be 3D printed or used in analysis software. If you version control the `bracket.scad` text file, any changes can be tracked. For instance, if you later decide to add a diagonal rib, you could code that and the diff would clearly show the new lines added.

Through this OpenSCAD example, you have effectively learned how to **code a mechanical part**. With practice, one can create quite complex assemblies using these principles. And as mentioned earlier, this textual approach opens the door to using AI assistants – you could experiment with prompting Copilot while writing such a script (e.g., typing `for(i = [1:` and seeing it suggest the rest), or even ask ChatGPT "How do I subtract multiple holes in OpenSCAD?" to get pointers.

Example 2: Mechanical System Modeling with Modelica

For the second example, let's shift to mechanical **behavioral** modeling. We'll create and simulate a simple **damped oscillator** – imagine a mass attached to a spring and damper (shock absorber). This is a classic mechanical system example that showcases Modelica's strengths with differential equations.

Step 1: Setup a Modelica environment. If you don't have a Modelica tool, one open option is to install **OpenModelica** (it provides OMEdit, a GUI, as well as a command-line). Alternatively, commercial tools like Wolfram SystemModeler or MapleSim offer free trials. For this example, OpenModelica is sufficient. Once installed, open OMEdit (OpenModelica Connection Editor). You'll see a modeling workspace where you can either drag components or write Modelica text. We'll focus on textual modeling, so open a new Modelica class as a text file.

Step 2: Write the Modelica model. In Modelica, everything is organized in classes (which can be models, records, blocks, etc.). We'll make a model called `DampedOscillator`. Type the following in the text view:

```
model DampedOscillator
 parameter Real m = 1 "Mass (kg)";
 parameter Real k = 20 "Spring constant (N/m)";
 parameter Real c = 2 "Damping coefficient (N·s/m)";

 Real x(start=1) "Displacement of mass (m)";
 Real v(start=0) "Velocity of mass (m/s)";
initial equation
 // Initial conditions
 // x is already set to 1 at t=0 via start, alternatively:
 // x = 1;
 // v = 0;
equation
 // Dynamics: m*acceleration + c*velocity + k*displacement = 0
 m*der(v) + c*v + k*x = 0;
 // Kinematic: velocity is derivative of displacement
 der(x) = v;
end DampedOscillator;
```

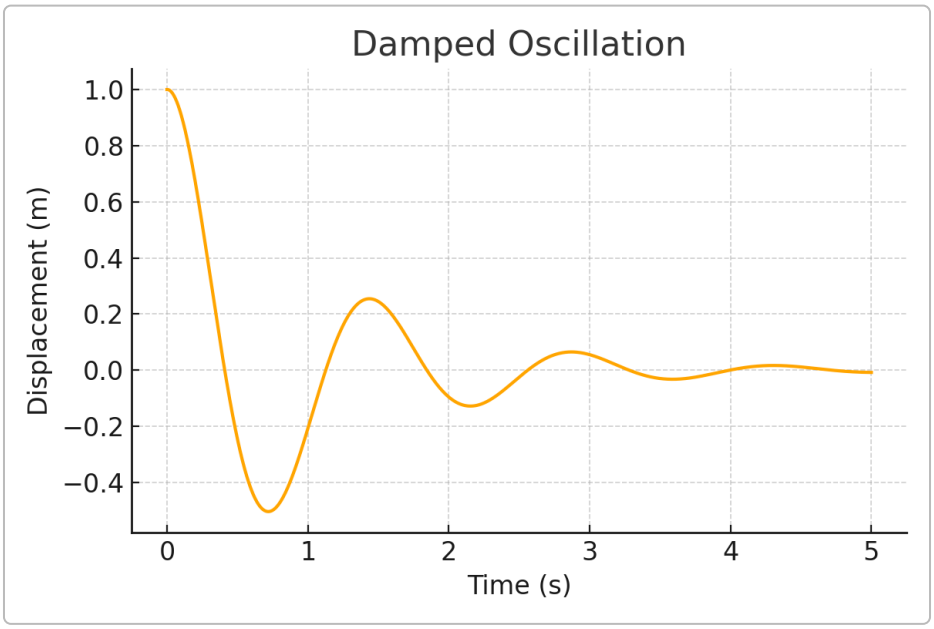
Let's break this down. We declare three parameters: `m`, `k`, `c` with default values (and units in comments). Then we declare two dynamic variables `x` and `v` for the mass's displacement and velocity. We set `start` values for them (initial condition: at time 0, x=1 meter displaced, v=0). We could also enforce initial conditions in an `initial equation` section (which we left mostly empty because the start attributes suffice here).

The `equation` section is where we put the system's equations. We write the second-order ODE as two first-order equations: the force balance `m*der(v) + c*v + k*x = 0` (which is Newton's law: mass * acceleration + damping force + spring force = 0) and `der(x) = v` (definition of velocity). Modelica knows `der(v)` means time derivative of v. These equations define the system's behavior.

One thing to note: we didn't have to write an explicit simulation loop. Modelica is *declarative* – we just state equations. The simulation engine will turn this into ODEs to solve [49]. Also, we didn't have to structure it like a program; it's more like writing physics laws.

Step 3: Simulate the model. In OMEdit, click the simulate/play icon. If the model is error-free, it will compile and run a simulation (by default, perhaps 0 to 1 second). In OMEdit, you can then plot variables. Plot `x` and `v` over time. You should see an exponentially decaying oscillation. Because we gave some damping (c=2), the mass will bob up and down and gradually come to rest at x=0 (equilibrium). The frequency of oscillation is about $\sqrt{}$(k/m) = $\sqrt{}$20 ≈ 4.47 rad/s (natural freq), and the damping ratio is c/(2$\sqrt{}$(mk)) ~ 0.224 (underdamped). So it oscillates at a slightly lower frequency and decays.

If you run the simulation for longer (you can set stop time to, say, 5 seconds), the motion will die out. The result might look like this displacement-vs-time graph:



Simulation result of the DampedOscillator Modelica model. The displacement starts at 1 m and oscillates around zero with decreasing amplitude due to damping.

You can play with parameters: try increasing `c` (damping) to 5 – the oscillation will decay much faster (maybe even become overdamped if c^2 > 4mk). Or increase `k` (stiffer spring) to see a faster oscillation. You can also try different initial displacements or give it an initial velocity.

Step 4 (Optional): Using components instead of equations. Modelica also allows a *component-oriented* approach, where you'd drag in a `Mass` component, a `SpringDamper` component, connect them, etc. Textually, that would look like:

```
model DampedOscillatorComponent
 Modelica.Mechanics.Translational.Components.Mass mass(m=1);
 Modelica.Mechanics.Translational.Components.SpringDamper sd(c=2, d=20);
 Modelica.Mechanics.Translational.Components.Fixed fixed; // fixed reference
equation
 connect(mass.flange_b, sd.flange_a);
```

```
  connect(sd.flange_b, fixed.flange);
end DampedOscillatorComponent;
```

This does essentially the same thing (the library's SpringDamper combines k and c, note I used c for damping and d for spring constant as per library's naming). The `connect` lines wire the mechanical ports: mass's flange to one end of spring-damper, the other end of spring-damper to ground. Modelica's library internally has the equations, but you could assemble systems like lego blocks. We show this to emphasize that textual doesn't mean you always write equations from scratch – often you connect predefined components by writing `connect()` lines. The end result simulation would be similar. Beginners might find the component method easier once they know the library, but it's important to understand the underlying equations too.

Step 5: Interpret results and next steps. The output of the simulation can be used to make design decisions. For instance, if this mass-spring-damper was supposed to represent a car suspension for a quarter car model, you'd interpret how quickly it settles after a bump (time to halve amplitude, etc.). If it's not satisfactory, you adjust parameters and rerun. This iterative loop is very quick in Modelica – much faster than building a real prototype! For onboarding, one can experiment with more complex models by extending this example: add a second mass (two-mass oscillators), add a forcing function (Modelica has source components to apply a sine or step force), or add a nonlinearity (like a hard stop or friction, which Modelica can handle with conditional equations or when-events).

Running Modelica models with AI: While not a built-in feature, you could use something like Jupyter Notebook with OMPy (OpenModelica Python API) to programmatically simulate models. This opens the door to using Python-based AI libraries to, say, tune parameters. For example, one could use an optimization loop to adjust `c` and `k` to achieve a target settling time. This is beyond manual onboarding, but it's worth mentioning because it hints at how code-based models integrate with software tooling.

In conclusion, this Modelica example showed how a mechanical dynamic system can be described in a **few lines of text** and simulated to gain insight. Compared to writing a differential equation solver yourself, Modelica did the heavy lifting. Compared to building a physical rig, it's orders of magnitude faster and cheaper. You also saw how units and documentation can be embedded (the strings in quotes after variables), which can be auto-generated into nice documentation – another perk of textual models. For beginners, the key takeaways are: identify the state variables, write the governing equations, and let the Modelica environment solve it. As you grow more comfortable, you can tackle very complex systems piece by piece (Modelica encourages that via subsystems). And as with OpenSCAD, an AI assistant can help here too – for instance, if you're unsure how to model friction, you could ask a code assistant and it might suggest using a conditional equation or a standard library component, even provide a code snippet.

By going through these two examples, you should have a taste of both ends of the spectrum: geometry modeling and system modeling. In both cases, the process involved **writing text that encodes the design** and using a tool to interpret that text into a result (3D model or simulation). This is the essence of text-based mechanical design. From here, you can explore more: perhaps try designing another part in OpenSCAD (there are many tutorials and examples online), or building a multi-body mechanism in Modelica (like a pendulum, or a double-spring system). As you do, remember that you're effectively programming the behavior or shape of your design – and this means you can leverage all the power of programming (loops, re-use, logic) and increasingly, the power of AI assistance, to supercharge your mechanical design workflow.

Conclusion

Text-based mechanical description languages transform the way we capture and communicate engineering designs. They bring the rigor and versatility of programming to the realm of CAD and CAE. This deep dive has highlighted that while historically a niche, these approaches are now buoyed by modern needs – **parametric automation, collaboration, and AI integration**. We surveyed the landscape of tools ranging from OpenSCAD and CadQuery for geometric modeling to Modelica for system dynamics, noting how each contributes unique capabilities and how they compare to the more mature field of hardware description languages. The comparative analysis underscored both the potential and current limitations: we have powerful examples and libraries at our disposal, but also gaps in ease-of-use and broader adoption.

Crucially, the advent of AI co-coding is a potential game-changer. It lowers barriers for newcomers and amplifies productivity for experts, as seen in early experiments of AI generating CAD scripts or assisting in design modifications. The **tight coupling of LLMs with text-based design** suggests that the future engineering workflow could be a seamless dialogue between human intent and machine-generated solutions, all mediated through a textual design language that both can understand [20] [21].

Our trend analysis noted that interest in these methods is rising, evidenced by new languages (like KCL) and research (like AIDL, Text2CAD) emerging in just the last couple of years. Industry is paying attention too – from PTC's OnshapeGPT prototype to major automotive firms using Modelica at scale. It's reasonable to predict that in the next decade, **hybrid GUI-code environments** will become common, and engineers fluent in both traditional CAD and code will be highly sought after. We may see the establishment of standard textual representations for CAD models, improved integration of simulation and design, and more democratized innovation as coding brings design to a wider audience.

However, challenges remain as roadblocks to overcome: improving user experience, training engineers in these tools, ensuring reliability of AI suggestions, and handling complex geometry elegantly in code. These are fertile grounds for research and development. If these are addressed, the payoff is significant – mechanical design could undergo a renaissance akin to how software development was revolutionized by high-level languages and open-source collaboration.

In wrapping up, we emphasize that text-based approaches are not here to replace traditional CAD outright, but to augment and enhance it. They offer **new ways of thinking** about design – more abstract, more automated, and more integrative. Combined with AI co-coding, they hint at a future where engineers can focus more on the *what* and *why* of design, and let code (assisted by AI) handle much of the *how*. The hands-on tutorials provided a glimpse of how accessible this can be: with just a simple text editor and some curiosity, anyone can start coding a bracket or simulating a spring-mass system. From there, the sky's the limit – one could code entire product assemblies or virtual test rigs.

The field is evolving quickly, and those who embrace these tools early will help shape the future standards and best practices. Whether you are a mechanical design professional, a student, or a software engineer stepping into hardware, learning these text-based methodologies opens up a new dimension of creativity and efficiency. The marriage of mechanical design and code – and by extension, AI – truly has the potential to be "a marriage made in heaven" [50], uniting the precision of engineering with the power of computational innovation.

Sources

- Kulkarni, Yogesh H., “Thoughts on Text-to-CAD: An AI Co-Pilot for Mechanical Design,” *Technology Hits*, Aug 9, 2025. [51] [44]

- Xie, Haoyang, and Feng Ju, “Text-to-CadQuery: A New Paradigm for CAD Generation with Scalable Large Model Capabilities,” *arXiv preprint arXiv:2505.06507*, 2023. [12] [13]

- Jones, B.T. *et al.*, “A Solver-Aided Hierarchical Language for LLM-Driven CAD Design (AIDL),” *arXiv preprint arXiv:2502.09819*, 2023. [22] [38]

- Dominguez, Xavi, “CAD with Code – HackMD Tutorial,” 2024. [7] [45]

- Rowntree, Dave, “CadQuery Comes of Age,” *Hackaday*, Feb 4, 2022. [11] [10]

- Chalmers, Adam, “KCL: A Programming Language for Parametric CAD,” *Zoo Research*, May 20, 2025. [20] [21]

- Onshape Blog (Valerio, Rachel), “Is OnshapeGPT on the Horizon?...”, Onshape, June 12, 2024. [18] [16]

- Reddit post by OpenVMP team, “Generate OpenSCAD with AI,” *r/openscad*, 2024. [33] [34]

- Golubev, Pavel, “Language models, parametric design spaces... in CAD,” *Medium*, Jul 9, 2025. [26] [25]

- Marco Bonvini, “All about Modelica,” personal blog, June 29, 2020. [17]

- FreeCAD Wiki, “CAD History,” (re: PADL) [1]

- Mastering OpenSCAD, “Project 1: Shelf Bracket,” 2022. [46] [47]

---

[1] [2] FreeCAD - FreeCAD / CAD History
https://www.freecad.sk/en/cad-history

[3] [4] [6] [20] [21] [24] KCL: A Programming Language for Parametric CAD | Zoo
https://zoo.dev/research/introducing-kcl

[5] [7] [8] [9] [45] [48] CAD WITH CODE - HackMD
https://hackmd.io/@xavidominguez/HkqzwFe4N

[10] [11] CadQuery Comes Of Age | Hackaday
https://hackaday.com/2022/02/04/cadquery-comes-of-age/

[12] [13] [28] [29] [41] [42] Text-to-CadQuery: A New Paradigm for CAD Generation with Scalable Large Model Capabilities
https://arxiv.org/html/2505.06507v1

[14] What Can CAD with FeatureScript Do? - Onshape
https://www.onshape.com/en/blog/standardization-automation-complex-design-cad-featurescript?author=emma+lin

15 16 18 19 Student Explores Text-to-CAD AI Chatbot with Onshape
https://www.onshape.com/en/blog/cad-integration-ai-featurescript-api

17 home - Marco Bonvini
https://marcobonvini.com/modelica/2020/06/29/all-about-modelica.html

22 23 32 38 39 40 A Solver-Aided Hierarchical Language for LLM-Driven CAD Design
https://arxiv.org/html/2502.09819v1

25 26 27 30 31 50 Language models, parametric design spaces, L-systems and formal grammars in CAD — a marriage made in heaven? | by Pavel Golubev | Jul, 2025 | Medium
https://medium.com/@paul.golubev/language-models-parametric-design-spaces-l-systems-and-formal-grammars-in-cad-a-marriage-made-24332c38ea11

33 34 Generate OpenSCAD with AI : r/openscad
https://www.reddit.com/r/openscad/comments/1bynbdh/generate_openscad_with_ai/

35 openscad · GitHub Topics
https://github.com/topics/openscad

36 [PDF] Simulation-Based Development of Automotive Control Software with ...
https://www.synopsys.com/content/dam/synopsys/verification/presentations/mbenz-sil-modelica.pdf

37 [PDF] CAD Scripting And Visual Programming Languages ... - CumInCAD
https://papers.cumincad.org/data/works/att/ijac201210108.pdf

43 44 51 Thoughts on Text-to-CAD. An AI Co-Pilot for Mechanical Design | by Yogesh Haribhau Kulkarni (PhD) | Technology Hits | Aug, 2025 | Medium
https://medium.com/technology-hits/thoughts-on-text-to-cad-c11998d06afd

46 47 Mastering OpenSCAD
https://mastering-openscad.eu/buch/example_01/

49 Solving Modelica Models - OpenModelica
https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/solving.html