AI BEFORE THERE WAS AI

What People Pretend AI Will Do

Jeremy Howe

Complete Working Draft v0.1

Contents

Is This Book for You?

There has probably never been a more exciting time to build things. For most of my career, turning an idea into a permanent solution meant learning to program, writing thousands of lines of code, testing it, maintaining it, and convincing other people that it was worth using. Today, artificial intelligence has removed much of that barrier. The difficult part is no longer building the solution. The difficult part is recognizing that a solution should exist in the first place.

If you have ever looked at a repetitive task and thought, “There has to be a better way,” this book is for you. It does not matter whether you are an engineer, accountant, teacher, designer, machinist, programmer, manager, or someone trying to run a household. Every field contains work that is genuinely new and work that has already been solved. Those two kinds of work should not be treated the same.

You do not need to know how to program. You do not need to understand artificial intelligence. Those are implementation tools. The important skill is learning to look at your work and ask a more useful question: Is this new, or am I repeating something the system should already know?

Once you begin making that distinction, repetitive work becomes difficult to ignore. Every repeated task quietly asks, “How could this become reusable?” Small answers accumulate. A file opens correctly. A name is already right. A color carries meaning. A routine check happens without being requested. A complicated process collapses into one instruction that describes the outcome instead of the procedure.

There is an uncomfortable part. Many people measure their value by how much work they complete. If their work consists largely of repetitive detailed tasks, removing those tasks can feel like removing their value. The opposite is possible. Eliminating repeated work can release the imagination that routine has been occupying. It can move a person from proving that they can survive tedious work to discovering what they are capable of creating when the tedious work no longer owns the day.

This book is not a collection of programming tricks or AI prompts. It is a record of a way of working that developed over decades: learn, solve, capture, reuse, improve, and keep moving the boundary between what requires a person and what belongs in the system.

Preface

This book is the sum of more than twenty years spent working as a designer and programmer while technology changed underneath me. I had the unusual luxury of seeing several generations of tools arrive, mature, and become ordinary. The tools mattered, but the more important change was what they allowed me to stop doing.

I did not begin as a programmer. I was a high school dropout who became a draftsman. On a drafting board, even the smallest repeated motion mattered. I carved details into my drafting triangles because the same detail had to be drawn again and again. The modification was crude, but the idea was already there: if a task repeats, the tool should remember it. It made the work faster, but the larger reward was that it removed another opportunity to make a mistake.

I have always hated being wrong. I hated the feeling of getting a drawing back covered in red marks. A missing dimension, incorrect text, or forgotten detail felt childish. The error was rarely interesting. It was simply evidence that my attention had slipped while I was performing something the process already knew how to check.

I also hate tedious work. I hate it. I hate it. I hate having my life depend on whether I can remain perfectly attentive while repeating details that have already been decided. That hatred turned out to be productive. It pushed me to carve the triangle, write the macro, build the template, combine the commands, and eventually create systems that pulled work into themselves like a vortex.

The first victories were small. A command would run and I would think, “Oh shit. That worked.” The relief mattered more than the excitement. I would never have to do that particular task again. Then came a different realization: “Oh shit. This is working.” The system was no longer eliminating isolated chores. It was absorbing the process itself. Mistakes began to disappear because the opportunities to make them disappeared.

The last realization was the one that changed everything: “Oh shit. This is going to be good.” Once the system had removed enough sludge, imagination had room. The question stopped being how to finish the assigned work and became what else could now be built. That spark continued to grow for twenty years.

I tried to learn three things every day. They did not have to be important things. Most were tiny. A command, a naming convention, a better way to

structure a file, a surprising property of geometry, a shortcut, a mistake that revealed a rule. Three small things per day do not feel significant. Over years, they become an operating system.

This book is not an account of perfect decisions. Some of the most valuable moments came from discovering that I was wrong. The useful response was not to defend the old answer. It was to update the system once and allow all future work to follow. That is one of the central advantages of building systems: the system can change without requiring every old decision to be repaired by hand.

The stories here come from product development, CAD, programming, and large engineering organizations, but the subject is broader. The subject is what happens when knowledge stops living only in people and begins living in the work itself.

Chapter 1
What People Pretend AI Will Do

A friend who was an executive at a large automotive supplier told me that one of the big AI companies had been brought in to study how people worked. The plan was to record mouse clicks, identify patterns, and look for opportunities to improve productivity. At first, that sounded impressive. Then the more I thought about it, the more it sounded like the same mistake large organizations had been making for decades, only with more expensive software.

I do not care how many times someone clicks a mouse. I care why the click exists. If a person performs the same sequence of fifty clicks every day, the valuable information is not the speed of the clicking or the order in which it happens. The valuable information is that fifty clicks have become a stable sequence. The work already contains enough knowledge to be described. Once it can be described, the system should be challenged to absorb it.

The conventional answer is to observe the person, document the steps, improve the training, measure the performance, and perhaps use software to tell the person which step was missed. That approach preserves the work and optimizes the worker. It treats repetition as permanent and asks humans to become better at carrying it.

The other approach is to question the work. Why should the person remember which file to open, which background to load, which sections to create, which colors to apply, which parameters to type, which checks to run, and which report to prepare? If the correct answers are known, they do not belong in a person’s memory. They belong in the system.

That distinction is the reason the title of this book mentions AI. People imagine artificial intelligence will arrive and remove busywork. They picture a system that understands intent, handles the procedure, prevents mistakes, and quietly delivers the desired result. That is a reasonable expectation. It is also something that could be pursued long before modern AI existed.

For years I built commands that did not merely shorten a procedure. They removed the procedure from the user’s awareness. The user selected the work and described the outcome. The system knew the environment, the document, the active part, the saved path, the data type, the naming rules, the logging requirements, the cleanup, and the next operation. If the selection was valid, the command processed it. If nothing was selected, the command prompted

for the right thing. If several operations were always clicked in the same order, they became one command.

The difference is easy to miss. A macro repeats clicks. A system carries knowledge. A macro says, “Here is how to perform these steps.” A system says, “Tell me the result you want. The journey belongs to the platform.”

Large organizations frequently spend enormous sums on software that checks work after a human creates it. A CAD user types values into dozens or hundreds of parameters. A data coordinator runs a checker. The checker finds errors. An email goes to everyone involved. Managers become angry. The user corrects the fields, reruns the process, and waits to discover what remains wrong.

The same knowledge that checks a field can often fill it. Program number, part number, user name, date, document type, file path, standard naming, and required metadata are not creative decisions. If the computer can verify the value, the burden should be on the organization to explain why the computer did not create the value.

This is the mistake hidden inside many AI projects. They begin by preserving the existing process. They record the work, imitate the work, score the work, and promise to help the person perform the work. The more important opportunity is to ask whether the process deserves to survive.

The question that starts this book is therefore not, “How can AI automate my job?” It is simpler and more disruptive:

> *Why does this work still exist?*

Everything that follows is an attempt to answer that question without sacrificing the part of work that makes people valuable.

Chapter 2
Where Engineering Lives

New is where engineering lives. Repeating is not.

One of the most persistent confusions in product development is the belief that engineering is everything engineers do. It is not. Engineering lives in a specific place: the unknown.

When a problem has never been solved, someone must explore it. The inputs may be incomplete. The constraints may conflict. The first idea may fail. The answer may arrive while staring at a blank screen, sketching on paper, moving geometry in a rough model, or vacuuming a pool while the conscious mind is occupied somewhere else. That work requires imagination, judgment, experience, and the willingness to be wrong.

That is discovery. It is valuable because the answer does not yet exist.

Once the answer is known, the nature of the work changes. Applying the same clearance, creating the same feature, naming the same object, checking the same rule, or reproducing the same report is no longer discovery. It may still be required, but it is execution. Calling both activities engineering hides the boundary between creating knowledge and repeatedly applying knowledge.

In mature automotive product development, that boundary is much further upstream than most organizations admit. A vehicle interior or exterior begins with styling surfaces, cut lines, established manufacturing processes, regulatory requirements, standard attachments, known materials, and known clearances. The visible product changes, but much of the engineering knowledge does not.

A glue track around a headlamp is not an artistic invention made fresh on every program. Its route follows known clearances to surrounding structure and attachment flanges. A clip is not merely a piece of geometry. It brings a clip tower, spacing logic, a mating cutter, a hole in the adjacent part, draft, manufacturing limits, and rules for when a doghouse is required. A doghouse brings its own undercut and wall-thickness relationships to prevent sink marks on the visible surface. Ribs carry draft, spacing, structural purpose, and manufacturing limits.

When those relationships are embedded in the template, the vehicle program no longer needs people to rediscover them. Styling changes the surface. The

system generates or updates the dependent features. The model reports its own condition through geometry and color. A feature turns red when a constraint is violated and returns to its designated color when the design moves back into the allowable space.

Traditional product development separates those events. Styling changes a surface. Someone loads background data, cuts sections, measures clearances, prepares a report, attends a meeting, explains the numbers, waits for a decision, and sends the issue back. The surface changes again and the cycle repeats.

A constraint-driven model collapses that loop. The surface moves. The result is visible immediately. The design is validated while it is being created.

This does not mean that human intelligence has no place. It means human intelligence should not be consumed by work whose answer is already known. The important boundary is not between people and machines. It is between unknown and known.

When I say that engineering lives in the unknown, I am not trying to diminish the people performing known work. I am trying to protect the part of their ability that routine crowds out. Discovery cannot be scheduled with the same certainty as a checklist. It needs room. It needs mistakes. It needs time for the subconscious to keep working after the person walks away from the screen.

The tragedy of many engineering organizations is that the people most capable of solving new problems spend their days proving they can repeat old solutions. Their attention is consumed by setup, metadata, documentation, checking, meetings, and the administrative residue of knowledge that was never compiled into the platform.

The first responsibility of a good system is therefore not to replace creativity. It is to protect it. Every known task removed from the person creates a little more space for the unknown. Every repeated decision embedded into the platform returns a little attention to discovery.

That is where engineering lives.

Chapter 3
The Framework

The framework began without a formal name. It grew from irritation, laziness, fear of mistakes, and the simple observation that doing the same thing twice felt like evidence that the tool had failed to learn.

The first rule is straightforward: the first time, do the work. The answer is not known yet, so learn. Make the ugly version. Find the commands. Discover the sequence. Notice what matters and what does not.

The second time, pay attention. Repetition has appeared. The task may still be changing, but there is now enough information to ask what can be captured. A reusable CAD feature, a small command, a template fragment, a naming rule, or a documented input may be enough.

By the third time, the work should be moving into the system. The third repetition is no longer an isolated event. It is a pattern. If it continues to depend on memory and manual execution, the organization has chosen to keep paying for knowledge it already owns.

Frequency determines the delivery method. Work used occasionally may live as a reusable feature or script. Work used daily deserves a command. Work used several times per day deserves an icon where it can be reached instantly. Commands always used in sequence become one command. The system evolves around actual behavior rather than an imagined master plan.

This is where the five-minute rule enters. The five-minute rule is not a precise accounting formula. It is a threshold for attention. If a repeated task consumes enough time or irritation to become noticeable, it is a candidate for removal. The command does not have to save hours each time. A thirty-second annoyance performed thousands of times may deserve more attention than a rare hour-long task.

The important point is not to stop all work and launch a giant transformation project. The system grows while the work continues. One task is done manually. The next version is reusable. The next version is easier to access. The next version includes the neighboring task. The next version checks the selection, handles multiple items, restores the user’s state, logs the result, and disappears into the normal flow of the day.

The architecture I learned to trust was inputs, process, outputs. Shared setup belongs in one place. Task-specific logic belongs in small files. Shared

cleanup, logging, security, and closeout belong in one place. A compiler assembles them in the correct order.

The physical files do not matter much. The same architecture can live in text files, VBA, Visual Studio, a website generator, firmware, or any other toolchain. Text is text. Reusable text plus specific text plus ordered assembly produces a finished artifact.

That separation allowed hundreds of active commands to share the same infrastructure. A change to security, logging, initialization, selection handling, or cleanup happened once. The commands were rebuilt, and the improvement propagated everywhere. The organization did not have to revisit hundreds of independent programs.

The framework also separates outcome from process. Users should describe the destination. The platform should own the journey. “Update the drawing” is an outcome. “Open File, browse to this folder, locate the correct drawing, open it, inspect every view, refresh sections, check dimensions, save, and close” is a procedure. The user should not be required to carry the procedure if the system can infer it from the working part.

This principle produced one of the most important command names in the system: Synchronize. Earlier versions exposed many individual update operations because that was how the software worked internally. Users did not want a tour of the machinery. They wanted the active work to become correct. The command name described the outcome, and the system decided which operations were required.

Another part of the framework is learning to be wrong effectively. I hate being wrong. That is precisely why the system must make wrong answers cheap to correct. A rule stored once can be changed once. Every future command, template, and project inherits the correction. A belief defended because changing it would be painful is not a best practice. It is technical debt protected by emotion.

Best practices are temporary. They must be retested. At one company, time studies showed that one CATIA method was dramatically faster than another for complex shapes. I carried that result into later work and defended it when challenged. Eventually I tested again. The result had flipped. The method I believed was slower had become much faster. I was disappointed to be wrong, but the better result mattered more than preserving the old conclusion. After that, I periodically turned off “I care” and retested. The number of times the answer changed was staggering.

The framework is therefore not a fixed collection of correct answers. It is a way to convert current knowledge into reusable execution while keeping the knowledge easy to replace. It assumes tools change, rules change, standards change, and yesterday’s best method may become tomorrow’s obstacle.

Knowledge should be created once. Value is created every time it is reused.

Chapter 4
Once You See It, You Can't Unsee It

Before the framework becomes visible, work appears as one large pile. Some tasks are difficult, some are easy, some are interesting, and some are tedious, but they all feel like parts of the job. Once the split between new and repetitive becomes clear, the pile never looks the same again.

Every task begins to sort itself. Is this discovery, or is this repetition? If it is repetition, the next question arrives automatically: how could this become reusable?

That mental change is what I call Leverage Time. It is not a separate project that requires everything else to stop. It is another way to view the work already happening. You do not disappear for six months and return with a perfect platform. You allow the platform to grow from the work.

Skills ratchet upward because each solution becomes the starting point for the next one. Intellectual property accumulates. Mistakes vanish when the tasks that create them vanish. Quality rises because the same proven behavior appears everywhere instead of being recreated by different people under different conditions.

The change is not only numerical. Work becomes more delightful. A file opens and everything is clean, named, colored, and complete. The user understands where they are immediately. The distance between “file opened” and “I know what I am looking at” becomes very short.

One command taught me how much value can hide inside a seemingly meaningless detail. A very talented engineer had a pet peeve: he wanted every name in the CAD file converted to capital letters. I thought the request was trivial. I made the command partly as a joke and used a capital T, the first initial of his name, as the icon.

The command was used more than half a million times in its first month. I had the data. The click count was impressive, but it was not the largest lesson. The surprise was that the files genuinely became easier to read. Thousands of names in inconsistent capitalization create visual noise. Capital letters made the list cleaner. Opening another person’s file became less irritating and faster to understand.

The command did not perform glamorous engineering. It standardized non-value-added work. That category deserves aggressive attention. Names,

colors, formatting, metadata, file organization, routine setup, repeated checks, and known updates should not be allowed to remain obstacles merely because each one is small.

Once standardized, these tasks begin to feel like the slides in old board games. What used to produce “Ugh, not that” becomes “Oh sweet, next.” The system carries the person past the sludge.

This is where “do not underestimate my laziness” applies. Constructive laziness is not refusal to work. It is refusal to spend human attention on work that does not deserve it. A lazy person willing to build a permanent solution may outperform a diligent person who repeats the same five-minute task forever.

Some people find repetitive tasks comfortable. The work is predictable. It is easy to measure. It can feel like a glorious sea of job security. A book that argues those tasks should disappear may feel threatening. It is threatening to the belief that value is proven through endurance.

The alternative is more interesting. When repeated work stops occupying the day, the person is forced to confront the unknown. That can be uncomfortable because the unknown allows failure. It also allows imagination, growth, and the possibility of work that did not exist yesterday.

Once you see the difference, you cannot unsee it. Every repeated action becomes evidence that the system has not yet learned.

Chapter 5
Eliminate the Tiniest Details

Large automation projects attract attention because the savings are easy to describe. A process that once took a week now takes an hour. A report that required three people now appears automatically. Those projects matter, but they are not where a system becomes pleasant to use.

Pleasant systems are built by removing tiny details.

A missing dimension, incorrect text string, wrong parameter, inconsistent name, stale background, hidden object, or unnecessary dialog box may consume only seconds. Individually, each one appears too small to justify investment. Together, they dominate the user experience.

The enterprise response is often to add checking. A checker verifies hundreds of parameters after the work is complete. It identifies every wrong field and produces a list. The process appears rigorous because errors are visible and people can be held accountable.

The better question is why the user typed the fields. Part number, program number, user, date, document type, file path, standard attributes, and naming rules already exist somewhere. Asking a person to re-enter known information converts a data relationship into a memory test.

The solution is not a more intimidating report. It is to eliminate the opportunity for error. Known values should be populated from known sources. Required parameters should be created automatically. Names should be generated. The file should be placed correctly. The checker should find nothing because the creation process already contains the checking rule.

This principle extends beyond metadata. If software can verify a deterministic condition, the organization should ask why the same knowledge is not present when the work is created. A rule used only for inspection arrives too late. It detects waste after the waste has been produced.

The first version of a command may simply perform a task. The second version should handle obvious edge cases. The third version should become difficult to misuse. Selection filters prevent the wrong data type from entering. Multiple selected objects are processed consistently. If nothing is selected, the command asks for one valid item. The user’s selection and screen state are restored afterward.

These details are rarely celebrated, but they determine whether a tool becomes trusted. A command that occasionally destroys the user's context will be avoided even if it is fast. A command that behaves politely becomes invisible.

The goal is not to make the user careful. The goal is to make care unnecessary where the answer is known. Human attention should be reserved for judgment, not spent guarding the platform from predictable mistakes.

Eliminating tiny details also changes the emotional tone of work. The day no longer contains hundreds of little traps. The user is not bracing for the red-marked drawing, the error email, the missing section, or the parameter that someone forgot to populate. Confidence comes from the process rather than from hoping every person remained perfectly alert.

This is how mistakes begin to vanish. Not because people become flawless, but because the work stops asking them to perform tasks in which a momentary lapse creates failure.

Chapter 6
Color Coding and Naming

Color and naming are often treated as cosmetic. In a system, they are language.

A large CAD model contains thousands of objects. Without standards, the user spends time translating the file before any engineering begins. What is background? What is active work? Which objects are warnings? Which geometry belongs to crash conditions? Which surfaces are styling? Which features are tooling, clearance, or construction?

A color standard answers those questions before they are asked. Body-in-white can be white. Clearance zones can be green. Critical or failed conditions can be red. Reusable construction features can carry a known color. The exact palette matters less than consistency. Once the meaning is stable, the screen begins reporting the state of the work.

The value is difficult to capture with a stopwatch because it lives in recognition. The user pans, zooms, rotates, and understands. No meeting is required to explain what is visible. No legend must be reconstructed from memory. The signal survives across projects and across users.

At one company, a particular RGB value I used for reusable features spread through the process. After leaving for a year and returning, I walked through the office and saw that color on screen after screen. The system had continued without me. The visual language had become part of how people worked.

Naming has the same effect. A clean tree of objects reduces the time required to find the working area and understand another person’s intent. The capitalization command described earlier was trivial in code and enormous in cumulative use because it improved the surface through which every future user encountered the data.

Names should describe purpose rather than history. A feature called “Join.47” tells the next user almost nothing. A feature whose name describes the result becomes a small piece of documentation that stays attached to the work.

The standard should not depend on every user remembering to comply. The system should create the name and color when it creates the object. If the object changes state, the system can change the color. If a feature violates a rule, red should appear without a meeting, report, or interpretation.

This turns the model into a communication device. The model is not only geometry. It is the report. The important conditions become immediately visible because non-value-added noise has been standardized and removed.

A well-prepared meeting can then become remarkably short. The work is complete, easy to understand, and already validated. The presentation is not a tour through open issues and explanations. It is: here is what was completed, here is the remaining decision, okay, next.

That simple meeting is not a lack of discussion. It is the result of a system that handled everything that did not deserve discussion.

Chapter 7
Building Blocks

The smallest reusable pieces eventually become more than commands. They become building blocks: self-contained capsules of knowledge that carry geometry, rules, behavior, and validation.

A building block is not merely copied geometry. A clip feature can know its standard hardware, tower, draft direction, spacing limits, mating cutter, hole, and conditions that require a doghouse. A rib can know wall-thickness relationships, draft, direction, spacing, and the structural or packaging purpose it must serve. A glue track can know the surrounding flanges and required clearances.

The block therefore contains more than shape. It contains an object's relationship to the product-development world.

This is where repeated value-added work goes through the same first, second, and third stages as small automation. The first time, an expert creates the feature. The second time, the repeated knowledge becomes visible. The third time, the feature should return as a reusable object rather than a fresh assignment.

The benefit is not only speed. The block arrives with proven behavior. Draft is not remembered. The correct die direction is not reinterpreted. The mating condition does not wait for a later check. The object knows enough about itself to participate correctly in the model.

Building blocks also preserve the history of solved problems without forcing future users to read that history. The expert's judgment becomes executable. A new user receives the correct feature and sees the correct behavior without first surviving years of mistakes.

This changes training. Instead of teaching every person how to recreate the standard, the platform teaches through the object. The model itself becomes the most current process document.

A mature building-block library is not designed from imagination in advance. It grows from actual work. Each repeated object earns its place. Rarely used or ineffective blocks can disappear. Frequently used blocks become easier to access and more deeply integrated.

The 5-minute rule helped prune a system that had once grown beyond a thousand commands. Emotional attachment initially made removal difficult.

Over time, attachment shifted from individual tools to the overall user experience. A command did not deserve to survive because it had once been clever. It survived if it continued to improve the work.

That is the discipline required for reusable knowledge. Capture aggressively, but curate without sentiment. The goal is not to accumulate the largest library. The goal is to make the user’s next action obvious, correct, and increasingly unnecessary.

Chapter 8
Dynamic vs. Parametric Design

Design does not happen in one mode. Trying to use the same level of precision from the first idea through final release makes early work slow and late work fragile. Three modes are useful: hack and slash, dynamic design, and parametric design.

Hack and slash belongs at the beginning. The screen is blank, the problem is not fully understood, and there is nothing to react to. The correct response is often to slop something onto the screen. Create a rough volume. Move a surface. Use placeholder geometry. Make something ugly enough to reveal what the real questions are.

This is not careless final design. It is a deliberate refusal to pretend that precision exists before the idea exists. The first model starts a conversation with the problem.

Dynamic design begins once there is something to manipulate. Commands with arrows, handles, and on-screen controls allow the designer to push, pull, rotate, stretch, and rough in a complete solution. Immediate visual feedback matters more than exact dimensions. The design remains fluid.

This is where proportions are discovered. What happens if the part moves over here? What if the radius grows? What if the surface is taller, the opening narrower, or the attachment shifted? The answer appears on the screen faster than it can be described numerically.

Parametric design belongs after the design intent becomes clear. Numbers, relationships, constraints, tolerances, manufacturing limits, and exact dimensions refine the solution. Parametrics are excellent for controlling a known design and terrible for discovering an unknown one.

Starting parametrically can trap the designer inside premature decisions. Time is spent entering dimensions for geometry that may not deserve to survive. The model becomes difficult to explore because every change requires negotiating with relationships created before the product was understood.

The progression is natural:

> *Hack and slash gives you something to look at. Dynamic design discovers the shape. Parametric design makes it exact.*

The same pattern appears outside CAD. A rough written outline is hack and slash. Rearranging sections and testing the flow is dynamic. Final language and formatting are parametric. A prototype circuit, a working webpage, or an early business offer can follow the same sequence.

The important skill is recognizing the mode. Precision is not always quality. At the wrong stage, precision is delay. Exploration is not always sloppiness. At the right stage, it is the shortest path to understanding what should eventually become precise.

Chapter 9
Templates

Templates are often approached as a giant planning exercise. Someone attempts to predict every future project, define the perfect master model, and force all work into it. That is not the kind of template described here.

A useful template emerges from building blocks. Reusable capsules begin connecting because the work repeatedly requires them together. The wall forms from the bricks. No committee has to guess the complete building before the first useful object exists.

A vehicle template can begin with placeholders for missing styling surfaces and zones showing where cut lines may exist. As stable inputs arrive, the placeholders update. Standard attachments populate according to spacing rules. Mating cutters reach into adjacent parts. Ribs, doghouses, draft, tooling directions, gap and flush, clearances, and validation come along because the building blocks carry those relationships.

The template does not wait for a person to remember every rule. It applies the current architecture to the current input.

This is why the same template can survive across programs and even across different companies. The styling surfaces and visible product change. The product-development patterns remain remarkably stable. The number and location of buttons may change. The established clip, rib, draft, clearance, and manufacturing knowledge does not need to be reinvented.

Master sections are a good example of the weakness of manual application. A required gap may be checked at a convenient midpoint and appear perfect there, while the condition drifts toward each end. The resulting shape can bow like parentheses because the process sampled a few locations rather than evaluating the complete relationship.

A template can check the condition continuously, at intervals far smaller than a person would reasonably inspect. The margin remains correct because the rule exists along the whole path, not because someone happened to measure the right point.

A mature template changes the role of the user. The user is no longer assembling the engineering environment. The user changes design intent and watches the system respond. The template contains the solved knowledge. The human explores the remaining unknowns.

Templates should also remain replaceable. A new standard clip, changed regulation, updated material rule, or better feature method belongs in one definition. The template updates, and every future use follows. The engineering effort scales with the platform rather than with the number of vehicle programs.

That is the central economic advantage. Engineering moves from every project into the reusable architecture.

Chapter 10
Systems

Building blocks combine into templates. Templates begin filling gaps between one another. At that point, a system appears.

The system is not a pile of automation. It is an operating model. Shared inputs, task-specific processes, shared outputs, reusable knowledge, deployment, feedback, and revision all work together.

The CATIA command framework used simple text files for a reason. One file carried common setup. Small command-specific files contained only unique logic. Another file carried shared cleanup, logging, security, and closeout. A compiler assembled complete commands. Some commands were not even stored as full source files; the compiler generated their code in memory.

The sophistication was not in the visual toolchain. The sophistication was in keeping the architecture visible. A plain-text system was portable, editable, recoverable, and easy to deploy. Fancy interfaces had been tried. The simple compiler remained durable.

A system also needs a clearly defined outcome. This is the point where “begin with the end in mind” becomes practical rather than motivational. The user must be able to state what correct looks like. Without a destination, the platform can only automate wandering.

Once the outcome is defined, the platform can own the steps. It can identify the active part, locate related files, determine what needs to be created or updated, apply standards, validate results, preserve context, and record usage.

The strongest systems remove state management from the person. The user no longer remembers which file, folder, order, view, update, and check belongs to the current task. The platform carries that state.

After returning to a previous workplace following a year away, I experienced my own system as a new user. I walked through the office and saw the standard feature colors on screen after screen. At the desk, someone explained the process: three buttons. One processed all background data. One updated all 3D CAD, including sections. One updated the drawing.

The drawing update was not a shortcut for opening a file. The system knew which drawing belonged to the active working part, where it lived, whether required content already existed, and what should be created or refreshed while the file opened. The procedure had disappeared behind the outcome.

That experience mattered because I had forgotten enough of the old process to feel the improvement. I was in awe of how smooth it had become. The creator returned as a user and discovered that the platform carried more knowledge than he did.

That is the standard for a system: it should not merely preserve expert work. It should make the expert unnecessary for routine execution, including the person who built it.

Chapter 11
Data Mining

Clean, consistent systems create clean, consistent data. At that point, data mining becomes surprisingly easy.

Every command in the CATIA framework logged usage. The system could record who ran it, when, on which workstation, in which document, on which part, and under what conditions. That logging was not primarily surveillance. It was a feedback loop.

Usage revealed which tools became essential, which were ignored, which departments adopted improvements, and which sequences should be combined. A rarely used command could be removed. A popular command could be simplified or promoted to a toolbar. A sudden change in behavior could reveal a problem before phone calls began.

The loop was simple: people used the platform, the platform produced data, the data revealed friction, and the next version removed the friction. Every improvement propagated to the entire user base.

At a large organization, that propagation creates enormous leverage. A few keystrokes can improve the output quality of hundreds or a thousand users. The creator is no longer fixing individual files. The creator is changing the conditions under which the organization produces files.

Consistent data also makes technical analysis easier. If names, parameters, colors, structures, and relationships follow standards, the system does not need to interpret a different local dialect in every file. Searches become reliable. Trends become visible. The data can answer questions that inconsistent work hides.

The phrase “single source of truth” is often used carelessly. Live data is not automatically trustworthy data. A model that changes continuously while people react to one another can create chaos. A manual update improves control but still does not reveal whether the latest save represents completed work or a temporary state.

One disciplined approach defined “latest” as the newest data that everyone could trust: the published state from the close of the previous day. Files were lightweight, intentionally prepared for others to see, and shared consistently. Everyone worked from the same set.

That story matters because systems need operational definitions, not slogans. “Live,” “current,” “complete,” and “approved” must mean something the process can enforce.

Once definitions are stable and data is consistent, mining the system is not an exotic AI task. It is a natural byproduct of structured work. The difficulty was not writing the query. The difficulty was building a process that produced information worth querying.

Chapter 12
Building Your Future

The early stages of this framework are defensive. Remove the tedious work. Prevent the mistakes. Stop the emails. Eliminate the repeated setup. Protect the day from sludge.

Eventually the purpose changes. The platform has absorbed enough known work that time and attention return. The question is no longer what can be eliminated. The question becomes what should be created next.

This is where fear of being wrong matters. Many people avoid unfamiliar work because a new field exposes them as beginners. Repetitive work feels safe because success is already defined. Discovery carries the possibility that the answer will fail in public.

A reusable system lowers that emotional cost. Past knowledge is not abandoned when the person explores something new. It remains available in the platform. The person can take a risk without returning to zero.

Learning three small things per day creates the same effect. No single lesson transforms a career. The accumulation changes the starting point of every future project. Tools, commands, patterns, standards, and mental models become a larger set of building blocks.

The transition can be heard in the internal conversation. “I am not going to do that because I might be wrong” becomes “I do not know, but I am going to have fun trying.”

That sentence is not motivational decoration. It describes a practical change in risk. When routine work no longer consumes the entire schedule and when mistakes can be corrected once in a reusable system, experimentation becomes cheaper.

New work should begin with hack and slash. Put something on the screen. Build a crude prototype. Create enough reality to react to. Dynamic exploration follows. Precision arrives later. The framework is not only a method for automating mature work; it is a method for moving new work toward maturity without demanding perfection at the beginning.

As new projects develop, their solved portions join the platform. The future is not a sequence of unrelated starts. It is a growing base of reusable knowledge that makes increasingly ambitious starts possible.

At that point, building becomes less about proving competence and more about choosing which unknown is worth pursuing.

Chapter 13
Living With Your Knowledge

The long-term reward of reusable knowledge is difficult to describe because it arrives quietly. There is no single day when the system becomes complete. Small solutions accumulate until the person can no longer remember all of them.

Then hidden, delightful surprises begin appearing.

A file opens in the right state. A command handles an edge case forgotten years ago. A color reveals a problem before anyone asks. A drawing updates itself. A feature appears with draft, cutters, and validation already attached. Someone else uses a tool correctly without ever meeting the person who captured the knowledge.

Every surprise is pleasant because it has already survived at least three stages. The idea was tried. It was reused. It was refined. The platform is not surprising the user with an untested experiment. It is returning proven knowledge at the moment it is needed.

This changes the relationship with work. The day becomes less about remembering everything and more about trusting the architecture. The system carries details that a person could not reliably hold after years of growth.

There is also an unusual emotional experience in encountering your own work after enough time has passed. Returning to the three-button process described earlier felt like learning someone else’s elegant system. I could appreciate the result without being trapped inside the memory of every line of code that created it.

That is one reason to design for the user experience even when you are the user. A system built only to demonstrate cleverness ages badly. A system built to make tomorrow easier continues paying dividends after the cleverness is forgotten.

Living with accumulated knowledge does not mean becoming rigid. The system must remain easy to update. Best practices change. Tools improve. Rules are revised. The architecture must allow one correction to propagate rather than turning the past into a museum.

The lifestyle to aspire to is not one in which every task is automated. It is one in which solved work stays solved. The person walks through the day surrounded by proof that earlier effort continues to work on their behalf.

That is leverage in its most human form: not a chart showing hours saved, but the repeated feeling of opening a file and thinking, “Oh sweet, this is already done.”

Chapter 14
People vs. Process

Organizations reveal what they believe by the way they respond when something goes wrong.

A process-focused environment examines the process. What allowed the failure? Which input was missing? Which rule was unclear? Where should the knowledge live so that the problem does not repeat? The process is under constant scrutiny because the process is considered the primary production asset.

A people-focused environment looks for the offender. Who made the mistake? Why was the work not complete? Who needs correction, training, escalation, or embarrassment? The system remains largely unchanged while the person is expected to perform the same fragile work more carefully next time.

The difference becomes obvious in meetings. In a process-driven environment, discussions are often reports of completed work: here is what we did, here is how we solved the problems, here is the remaining decision. The answer may simply be “okay,” followed by the next topic.

That short meeting is a holy-grail result. Non-value-added details are not part of the discussion because the system already handled them. Value-added work is named, colored, organized, and easy to understand. Critical issues stand out because the noise has been removed.

In a people-driven environment, the meeting often moves down an open-issues list. Ownership is assigned. Blame is distributed. The group designs by committee or identifies the future meeting in which the issue will be discussed again. Without clear process goals, every question becomes “Why wasn’t this done?”

A company can copy the names and milestones of a successful process without copying the operating logic. The result is a schedule of punishment dates rather than a sequence of clearly defined outcomes. The appearance of process hides continued dependence on people improvising their way toward vague checkpoints.

This distinction is not an abstract management preference. It determines whether improvements scale. Correcting one person may affect one future

task. Correcting the platform affects every future user and every future project.

The resistance is understandable. In a people-focused organization, knowledge, status, headcount, and responsibility are tied to individuals. Moving knowledge into the platform changes the location of value. A manager whose organization is defined by people executing known tasks may experience an efficient system as a threat rather than an improvement.

The system builder can still succeed inside that environment. Build personal systems. Standardize your own non-value-added work. Capture your own solved knowledge. Make your files, tools, and outputs easier to understand. Allow completed work to speak clearly enough that the meeting can move on.

The goal is not to win an argument about culture. The goal is to create a small process-focused environment around your own work and let the result become visible.

Chapter 15
How AI Makes This Easy

For most of the period described in this book, implementation was the barrier. Recognizing a repetitive task was not enough. Someone had to understand the application, object model, programming language, deployment method, error handling, security, versioning, and maintenance.

The architecture could be simple and the work to create it could still be substantial. A person might see exactly what should disappear and lack the programming skill or time required to make it disappear.

AI changes that relationship. It does not eliminate the need for the framework. It makes the framework accessible.

A person can now describe an outcome in ordinary language, show examples, provide files, explain the repeated sequence, and ask for a script, command, template, or small application. The first version may be imperfect. That is acceptable. Hack and slash applies here too. Put something on the screen. Test it. Correct it. Move toward dynamic control and then precise behavior.

The critical skill remains seeing the work. Asking AI to reproduce a bad process may produce a faster bad process. Recording clicks and imitating them may preserve dozens of unnecessary decisions. The useful prompt begins one level higher: what outcome is the person trying to produce, and which parts of the journey are already known?

This is why the book is called AI Before There Was AI. The philosophy did not begin with language models. It began with carving a drafting triangle, writing a macro, combining repeated commands, embedding rules in templates, and allowing the platform to remember what people had already learned.

AI did not invent reusable knowledge. It lowered the cost of expressing it.

That creates a remarkable opportunity for people who have never considered themselves programmers. The domain expert can describe the rules. The system can help translate those rules into code. The first, second, and third stages can happen faster. A small annoyance can become a reusable tool before it has time to harden into “the way we have always done it.”

There is still work. Inputs must be clear. Outcomes must be defined. The generated result must be tested. Rules can be wrong, code can be wrong, and

valid rules can conflict. The advantage is not magical perfection. The advantage is that once the correction is understood, it can be made once.

The most important use of AI is therefore not asking it to appear intelligent while repeating solved work. It is using it to help move solved work into deterministic execution and return human attention to the unknown.

People have spent years pretending AI will do that someday.

The tools now exist. The remaining question is whether we are willing to stop protecting the work that should disappear.

Working Notes for the Next Draft

This manuscript is intentionally complete enough to read from beginning to end, but it remains a first full assembly. The recovered source contains additional stories, quotations, examples, and corrections that should be mapped into later revisions. The next draft should preserve the current structure while strengthening transitions, expanding the most important firsthand stories, removing repeated explanations, and verifying every quantitative claim.

The central test for revision is simple: the reader should finish each chapter seeing a category of work that was previously invisible. The book should not tell the reader to feel inspired. It should show what happened, why it mattered, and allow the result to carry the emotional weight.