Command Is Not Only a Pattern. It Is the Native API for AI.

Introduction

About half a century ago, methods for good software design were established; they eventually became known as **design patterns**. They are used relatively little. However, other technologies are entering the picture, and today especially **Large Language Models (LLMs)** — grandly referred to as AI — are having an impact. These penalize the non-use of design patterns already during the software development process. This is elaborated in my article here:

AI-aware Architekture, Another Approach, Petr Talla

In this article, the emphasis is on **high-quality design when using LLMs to write code**.

However, there is another relationship between software and LLMs. **An LLM can be a user or an intermediary in the use of software.**

I addressed this relationship three years ago here:

Empovering Program Control with Natural Language using LLM like Chat GPT, Petr Talla

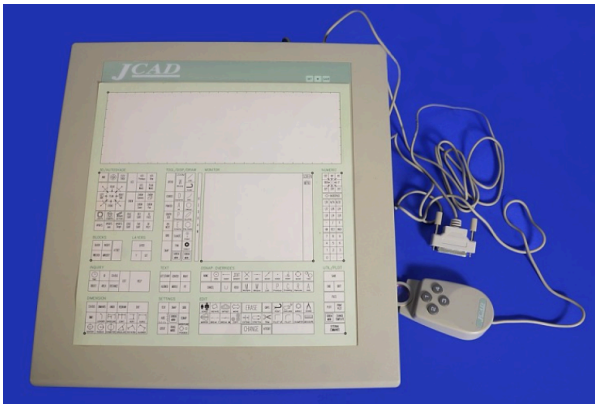
At that time, LLMs had been generally known for about half a year, if we ignore text translators. The content of that article did not come to my mind because LLMs appeared, but because I had many years of experience using the **Command pattern**, even though the industrial mainstream was playing a dirty game with callbacks.

When presenting these ideas to people who have no experience with commands, I encountered a strange phenomenon. People cannot fully imagine how commands are used, because they are not part of their programming repertoire.

This is the main topic of this article. To contribute, even if only a little, to the renaissance of ideas that corporate machinery and shortcut thinking have completely crushed. They survived only in the awareness of a few madmen who nevertheless sense that the moment has come to release the Kraken — a few simple ideas that, in combination with LLMs, completely change the rules of the game.

History

We know **Command** mainly from the command line. In the context of LLMs, it is also used in agents for searching and manipulating code (`git` , `grep` , `sed` , `awk` , `find` , `curl` , `ssh` , `tar` , `make`). However, that will not be the topic here. **We will talk about Command in a GUI environment.**



Tablet. Screens was too small and expensive to display GUI.

Tablet. It is around the year 1975. The company Intergraph uses this device for graphical command input. Each click invokes a command that performs some action, as mapped in some data structure.

By clicking, commands could be invoked, for example, for drawing a line:

```
LEVEL 3
COLOR 4
LINE TYPE 2
LINE
POINT 50 60
POINT 80 90
```

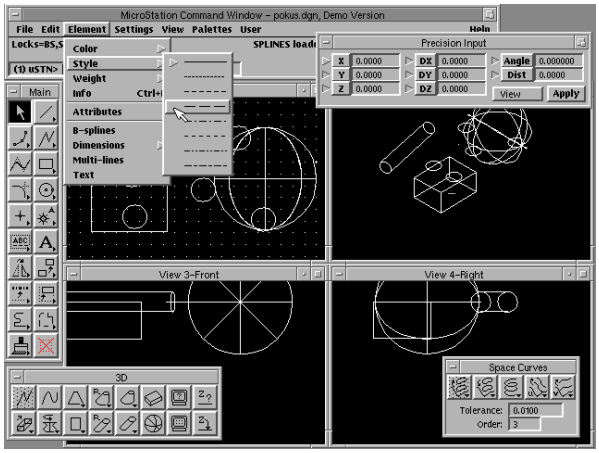
POINT was entered on the white drawing surface of the tablet.

FREEZE FRAME: it is important to say one thing now. In 1975 we have an LLM-enabled system available. Any LLM can easily handle the interaction described above. In 2026 most systems are not LLM-enabled. A paradox.

In 2026, instead of this beautiful clean architecture, we have callbacks stitched somewhere inside binary files, which are useless from the perspective of an LLM. In a better case, such as with Qt, we can reach the command, but from the point of practical use forcing an LLM to simulate clicking a button is also a crime.

A mouse click is an event, instead of that wonderful abstract command `POINT`. And one inevitably asks: where is this world rushing?

Further development at Intergraph leads to the same system, only the menu is drawn on the screen, and `POINT` is clicked there with the mouse. Under the buttons and text input fields, the engine based on Commands is still beating.



Microstation 4. 1987. Perfection itself.
Under every UI element there is a hidden textual command.
A click in the black area generates the command `XY`.

Then the Bentley clan arrives — five brothers who change the secure world of the digital giant. Keith Bentley uses the extended Intergraph IDGS format and writes software for the inexpensive MicroVAX system, about 4× cheaper than the equivalent from Intergraph. The architecture preserves the Intergraph command system quite faithfully.

And this is already the place where I encounter the command myself. And since at that time I am also experimenting with programming, I use the same pattern. In those days something like patterns hardly exists; methodology is ruled by the waterfall model. Prehistory. But it does not really matter — it is enough to recognize a masterful piece of work and simply imitate it.

Command na Microstation

A DGNXF file (a renamed IDGS file from Intergraph) has been loaded. How is this data structure governed by commands?

Besides the DGN itself, there are data structures that store certain values called **Settings**: color, active layer, grid, orthogonality, and who knows what else. Each of these Settings is changed using some Command:

```
LEVEL 3
COLOR 4
GRID 1 10
GRID ON
GRID LOCK
```

Then we have specific commands that abstract mouse input:

```
XY 50 60
RESET
```

`XY` we already know, `RESET` is the right mouse button and means the same as `Esc`. `XY` is entered using the left mouse button — simply a click. But when we write `XY 50 80` on the command line, it is the same as a mouse click. And this simple trick was prepared already 50 years ago so that AI could use it later.

And now comes the magic. What the system does on the command `XY` or `RESET` is determined by the **program mode**, paradoxically called a command in MicroStation. The mode determines what happens when we enter `XY` or `RESET`.

Context menu after activating the command `PLACE SMARTLINE`

The mode is also set using Commands: for example `CREATE LINE`, `CREATE CIRCLE` are commands that completely replace the handling of the commands `XY` and `RESET`, that is, they change the mode. They also adjust the UI so that it contains the settings components related to the active mode. This is the whole trick of forcing the environment to perform something specific.

For example, `CREATE LINE` has `XY` implemented so that it only stores the corresponding point. The command `RESET` is implemented so that it creates the line and stores it in the system.

There is one more point. Today, a command is used even for `MOVE`. At that time, however, this was not computationally feasible — the available hardware was an AT 286 processor at 12 MHz, 2 MB of RAM, a floating-point coprocessor, and no graphics acceleration. In MicroStation, therefore, `MOVE` was implemented only as a binary handling replacement of the function for the given mode. In the case of `PLACE LINE`, `MOVE` is a preview of what will happen when the command `XY` is entered.

Needless to say how this wonderful mode handling has been ruined by almost all modern systems. It is a disaster instead of brilliant simplicity. Not to mention the health problems of the hand when using drag-and-drop. And during drag-and-drop I cannot set the color. Sigh. Yet drag-and-drop can be just a primitive layer built on top of two `XY` and a `MOVE`.

One thing is essential. This is an environment that an LLM can use. We can easily verify this.

PROMPT:

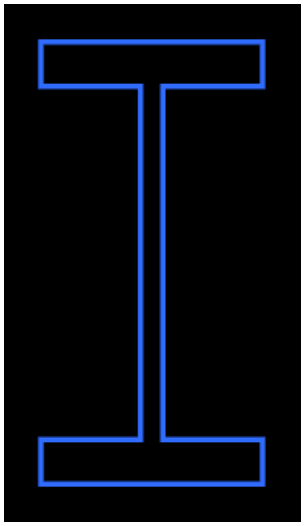
```
You are in MicroStation 4
color palette:
COLOR 0 - RGB 0 0 0
COLOR 1 - RGB 255 0 0
COLOR 2 - RGB 0 255 0
COLOR 3 - RGB 0 0 255

LAYER 1 - sections
LAYER 2 - openings + fills
LAYER 3 - equipment

draw a steel I-beam profile
height 100, width 50
web 5, flange 10
color blue

you cannot use coordinates — except for the initial `XY`, use `DX`
```

RESPONSE:



```
ACTIVE COLOR 3
ACTIVE LEVEL 3
PLACE LINE
XY 0,0
DX 50,0
DX 0,10
DX -22.5,0
DX 0,80
DX 22.5,0
DX 0,10
DX -50,0
DX 0,-10
DX 22.5,0
DX 0,-80
DX -22.5,0
DX 0,-10
RESET
```

This is what the interpretation of the commands created by the LLM in the program looks like.

It is the year 1984. We have a program that can communicate with AI. Today, programs usually do not have this capability.

I would emphasize some details of the interaction. The LLM itself converted "blue" to RGB and to the color number. It correctly determined the layer in which it should draw.

Command is not a historical pattern, but a natural API for LLM.

Display Commands

However, something is missing. In this form, an LLM can push something into the document, but it cannot determine what is already in the document.

This is because commands are used only for modification, not for retrieving information. Nothing like ...

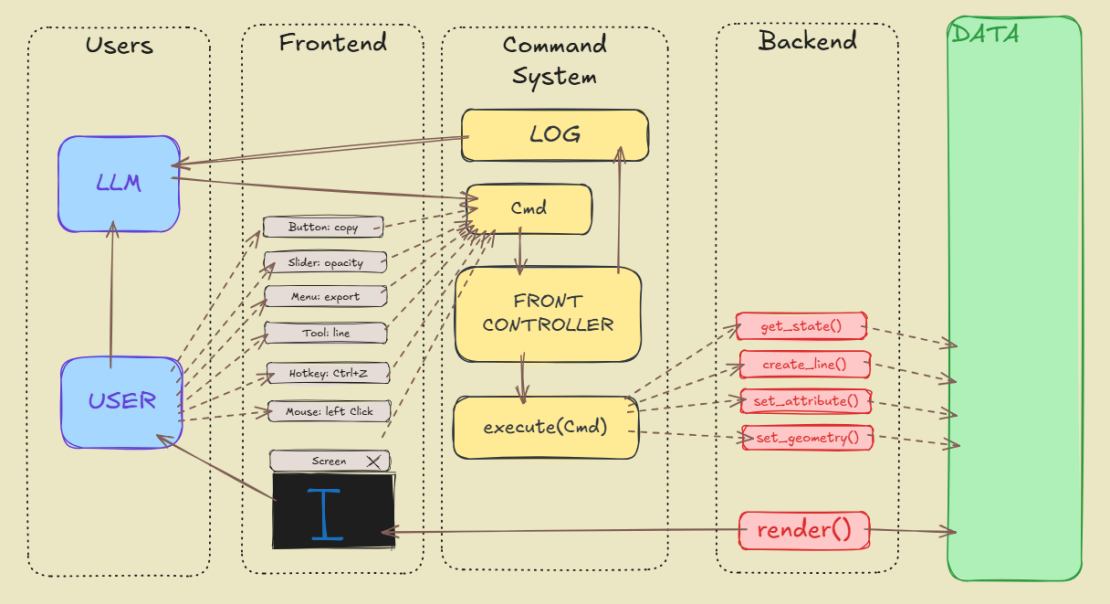
```
LIST ELEMENTS
GET COLOR
QUERY LEVEL
COUNT OBJECTS
```

... does not exist there.

GUI software for visualization does not use commands. Instead, it uses a reactive pattern and the principles of the Observer pattern. (In MicroStation 4 there was still the principle of clearing the board. Something like `CLEAR_REDRAW` was at that time one of the system commands.) Commands are used during the initial development phase (when the UI does not yet exist) and for debugging — we obtain the ground truth without the information having to pass through a complex system.

For cooperation with LLMs, any form of information retrieval is essential. It is necessary to incorporate it even into GUI command-based software. Otherwise, these systems are only partially accessible to LLMs. In such systems, the LLM would effectively be blind.

Command Queue

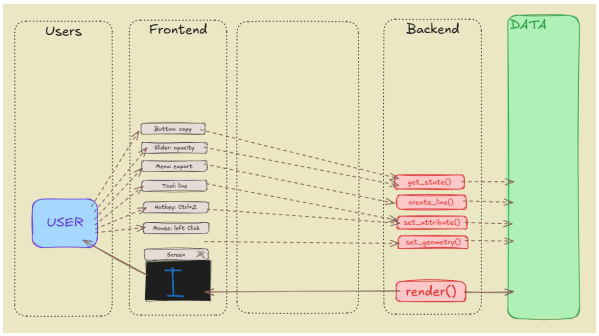


Schema of a command-driven system. The Command API strictly separates the frontend and backend and incorporates LLM into the system.

A Command has one fundamental property. During execution it can pass through a single place based on a **Queue** principle — one after another. This provides remarkable architectural capabilities: commands can be logged, filtered, trigger additional functions in sequence, be prioritized, etc. It is not the firing of events back and forth without any capture of what actually happened.

A special mechanism is used when invoking commands. A command may invoke another command, but only by inserting it into the Queue, where it will execute only after the current command finishes. This is called **stackless execution**.

Commands are always executed by a single thread. However, a command may create threads — or better, stateful jobs — and control them. Because commands fall into the queue, it is possible to ensure **lockless invocation from any thread**.



Schema of a traditional callback-based system.
The hell programmers created for themselves.

Short summary:

Command-based system: button → command → queue → execution

"Normal" system: button → callback → chaos

Syntaxe Commandu

For the syntax, the one that has recently become dominant on Unix systems is probably the most suitable.

```
command_name arg0 arg1 ... argN --named0 value0 -- named1 " s mezerou na zacatku" --flag
```

It is not a good idea to use formats such as JSON for the syntax — it is a parsing hell that will destroy any code. JSON is, however, completely fine as an arbitrary argument of the command form shown above. So something like `command_name {"json"}` is perfectly acceptable. Especially when a command is executed in a client–server

environment, `json` as an argument is a good choice. Locally, atomizing commands is optimal; in client–server calls it is not so clear-cut.

In code, a command can then look like this:

```
using CommandHandler = int (*)( CmdArgCol& args, QByteArray* data,
                                const QSharedPointer<CmdContextIface>& context);

class  CmdArg {
public:
    CmdArg( const QString& name = "", const QString& value = "") :
        name_(name),
        value_(value)
    {}
    QString name() const { return name_; }
    QString value() const { return value_; }
protected:
    QString name_;
    QString value_;
};
```

`context` — data through which the user of the system maintains information about themselves. This is especially important in a server environment. In the GUI case it has reason as well, the program operates as a multi-user server. There is at least one additional user called **LLM**, and from the perspective of possible auditing it is necessary to distinguish its identity.

`QByteArray* data` — used for situations where the entity issuing commands is a machine; it represents one extra binary argument.

```
class  CmdSys : public QObject {
    static CmdSys& inst() { static CmdSys i; return i; }

    int execute(const QString& args, const QString& sourceName = "", int sourceIndex = -1);
    void add(const QString& name, CommandHandler handler, bool excludeExeRec = false, bool
excludeUpdate = false);
protected:
    QMap<QString, Cmd>    cmds_;
};
```

Commands are then added using the `add` method to the `QMap` object `CmdSys`. In `execute`, they are found according to the value of the first argument and executed.

Details of a possible implementation are here: cmd_sys.

Log

The chapters **Queue** and **Command Syntax** provide the necessary prerequisites for completing the reasoning about commands in GUI environment. The queue is hooked at command completion and the result is written to the log.

If, during execution, we allow arguments to be extended with additional values, this is also a way for the command to report about its activity. Or it can output what we wanted to know about the running system. The number of added arguments then becomes the return value of the Command execution.

This is not the chaotic way command-line utilities typically work, but a structured one. Just as we parse the command on input, we parse it deterministically on output as well.

So in the queue we capture the output that we would normally send to a log and send it directly to the LLM. This gives the LLM the information exactly in the form it needs — as structured text. **An LLM is therefore not blind in any GUI system.**

Example of input:

```
get_objects 0 0 50 50 -filter LINE
```

For example, it may produce the output:

```
2 ← get_objects 0 0 50 50 -filter LINE --object 245 --object 894
```

Input:

```
get_points 245
```

Output:

```
2 ← get_points 245 --point 24 85 --point 84 12
```

Note the trick with the repeated argument. Technically, it is an array. If a formatting convention for arguments is added — here `--point` contains a point `[x,y]` — any additional formatting is usually unnecessary. This covers most possible data while keeping parsing trivial compared to JSON. If the data are more complex, JSON remains available as a fallback — only the command parsing must then handle all the intricacies of JSON.

Input:

```
get_points 2425
```

Output:

```
1 ← get_points 2425 --ERROR 2425 unknown
```

But one more point regarding the recording of commands as a log.

Logs, as they are commonly used, are completely wrong. They randomly print chaotic system outbursts without order or structure. Shameful to speak about.

A log as a **record of Commands** is deterministic. In the case of a single-threaded system, there is no chance for the system to do anything other than what is written in the log. In the case of multithreading, it is almost the same. **This is a real log.** Replaying the log means that the system will reach exactly the same state. There is no chance for it to be otherwise.

Outputs such as `--ERROR`, etc., should only be produced within the context of the function implementing a Command. However, this mechanism may be triggered from anywhere in the code. It is only necessary to insert a logging command into the Queue. Then it looks roughly like this:

Output:

```
1 ← get_points 2425 --ERROR 2425 unknown
0 ← log --ERROR no document loaded"
```

Because of the Queue and prioritization, the output appears immediately after the command that caused it. The log is therefore no longer a pile of nonsense, but a deterministic document. Good for LLMs, but also good for humans. A linear, language-based, deterministic state is provided to the LLM. No GUI guessing, no scraping.

At this point, any software that uses this design approach becomes open to LLMs. The only thing that has been done is a slight bending of a design pattern that is more than half a century old.

Moreover, questions such as *“why did this happen?”*, *“who triggered it?”*, *“what exactly did the AI do?”* have no answer without commands. And once AI stops being a toy, callback chaos becomes indefensible. Another

property of a command is that it carries the source of invocation — whether it was triggered by a human through the UI, by AI, or through a socket.

Incidentally, additional functionality that is otherwise difficult to implement in software can easily be attached to the log: undo, macro recording, etc. It is also excellent for testing — even with the help of LLMs.

Commands as a catalog of functionality for LLMs

Another aspect of the existence of commands must also be emphasized.

```
XY x y          - a point with coordinates x y is entered; execution is defined by the current
mode
RESET           - termination event; in some cases it may eliminate the need to enter one XY
PLACE_LINE      - starts the line drawing mode; command XY supplies the points of the line,
command RESET
PLACE_CIRCLE    - starts the circle drawing mode,
                  command XY supplies the center and then a point on the circumference,
                  RESET deletes the previously entered point
SET_COLOR r g b - color used when drawing LINE or CIRCLE
```

What has been written above is a complete definition of the system, a complete manual, and complete instructions for an LLM.

It is trivial, but the goal here is not to list dozens of settings and modes as in real software. The properties of such a definition of functionality are as follows:

1.)
Behind this simple lies non-trivial software with hundreds of functions. This is probably the complexity reduction that Commands provide. **1:100** — roughly one command corresponds to about 100 functions in the code. The monster becomes something that can be understood and controlled.

2.)
This definition can be understood even by most janitors in the building. Not so a salesman — who can easily define additional functionality in the language of programmers. A more experienced user can do it as well.

3.)
It is a complete and unambiguous definition of the interface for an LLM. It is a means of communicating with the software. Commands represent knowledge about the system — a kind of help for the LLM describing what the system does.

4.)
Conversely, in relation to a human user. Every button willingly and generically reveals the command hidden behind it. A kind of instant help.

5.)
For this definition, a UI can be programmed without implementing the commands. The UI may be shared by two different implementations, and vice versa.

6.)
An implementation can be programmed without any UI at all.

7.)
System scaling is possible by individual commands. The system can be sold by individual commands. Do you want 50, 75, or 200? We link them, package them, and that is it. A command can become the sales unit of the software.

8.)
Different command systems can be easily compared, for example with the help of AI. Tell me, system, what your commands are, and I will tell you what you are and what you are worth.

9.)
Testing. An application defined in this way can be tested back and forth by an LLM. In all possible combinations of user input. Goodbye manual testing or writing scripts. A brutal force arrives that relentlessly

tortures software in 24/7 mode. If the software crashes, the command log can be replayed and the system returns to the exact point of code execution where the test occurred.

What does today's programming world offer instead of this design jewel? A tangle of UI and execution through callbacks, without any ability to work with LLMs.

LLMs, however, cannot "discover" a system by clicking. They cannot read callback chaos.

Similar techniques of today

Something like Commands has survived on servers in the form of HTML anchors (`<a href>`), HTML form submission, REST, and RPC.

However, current server techniques emphasize statelessness. For LLMs this is useless. An LLM used this way becomes a tool for endless polling nonsense. Personally, polling is considered a crime even for ordinary, simple UI.

The Command concept naturally also works as a server-side solution. A suitable approach is deployment using WebSockets together with stateful jobs. The LLM maintains its own state space on the server and receives only update information.

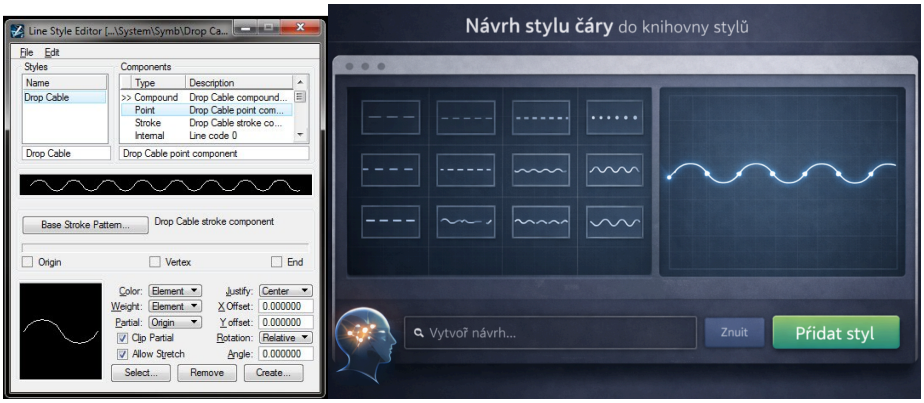
However, the server-side solution is a specific extension of Commands and may be discussed in another article.

Replacing GUI with LLM

In the entire preceding text, the LLM has been presented as a global handler or intermediary for the user of the application. However, the use of LLMs can also be approached in another way.

The application data file is another possible place where an LLM can integrate itself. Without the possibility of interaction.

A readable, non-cryptic application format is required. The ideal form is a **Domain Specific Language (DSL)**. In the age of LLMs, DSL is not something difficult to grasp. An LLM will generate the code for handling it in a moment and without errors — it should be remembered that an LLM can now produce a C compiler without difficulty.



About 50 interface elements — it has tabs, replaced by LLM–DSL interaction.

In this modification, the application's data model becomes a direct part of the UI. **Raw data input** is something that click-based interfaces abandoned long ago. With LLMs, however, it becomes a very accessible method for users. The user enters a request, a preview is shown, corrections follow, and then submission and integration into the system data.

However, if a system implements **Commands, it also has a DSL**. Using existing data formats — if they are at least textual — is only a substitute for a more perfect method that moves the LLM into an interactive role.

A command line in the form of LLM interaction with preview becomes the new dominant element of the application.

Why it is so

This is a question that cannot be answered easily. In the end, even Bentley does not fully appreciate the brilliance of the solutions inherited from Intergraph and looks toward the mainstream. Intergraph itself forgot

its unique solutions long ago.

1.)
A possible motive is distrust toward a strange domain language inside the supposedly consistent environment of a programming language. It is not realized that consistency and architecture — in the sense of separating the essence — are two completely different things.

2.)
A deal with the devil. Systems offer callbacks, so they are used.

It never occurs to create a tiny library such as:
cmd_sys

And then just a few lines of code, for example in the case of QML:

```
class UiControl : public QObject, public CmdExeGuard {
    Q_OBJECT
public:
    UiControl(QObject *parent = nullptr) :
        QObject(parent),
        CmdExeGuard(Update)
    {
        CMD_SYS.reg(this);
    };
    static UiControl& inst() { static UiControl i; return i; }
public slots:
    void callCmd(const QString& cmd) { CMD_SYS.execute(cmd); }
};
```

Example of usage:

```
Button {
    text: "Copy json to Clipboard"
    onClicked: qmlInterface.callCmd("copy --text \"" + overlappingPopup.jsonData + "\"")
}
```

Companies, as they have been observed over the years, are mostly unable to produce something this simple. Instead, money is happily drowned in wiring callbacks.

Note: it is a deal with the devil. Once it is signed and those callbacks are written, there is no escape — except at the cost of expenses that, globally, would amount to at least hundreds of billions of USD.

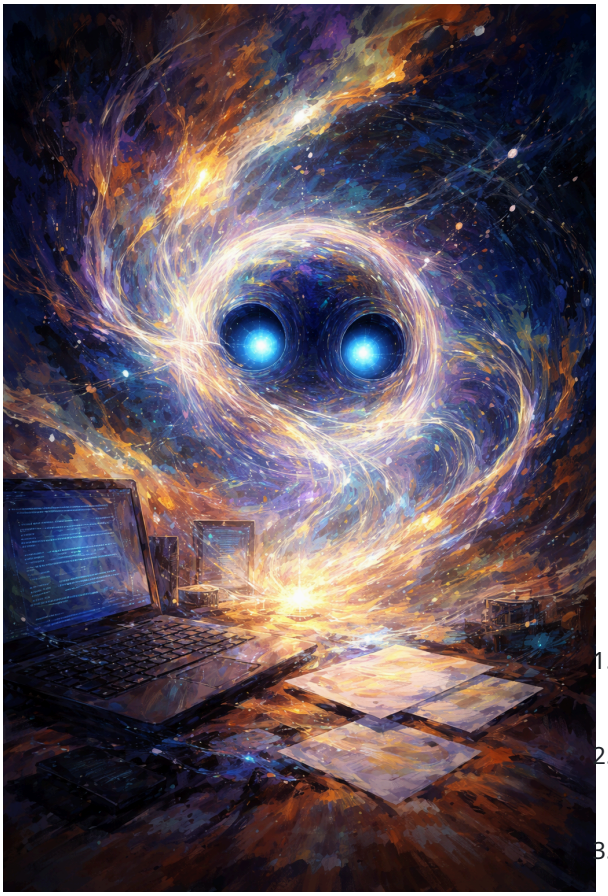
And yet those few lines above are enough to escape the contract. Tired of QML? Those few lines are written in another system and goodbye. In the past, a domain language was even used for writing the UI. Goodbye then took one day, because that was the time needed to rewrite the domain language for another system. With AI, one hour. For a company, the migration cost to another system becomes about 25 USD.

It is quite possible that this is a business strategy. As a tradition, it can be observed at Microsoft, which is famous for binding users through its inventions. But why even Qt behaves this way remains a mystery.

Conclusion

Without the **Command** application layer, an LLM remains mere empty chatter for any software. An LLM can talk about possibilities, describe the UI, and advise where to click, but by itself it executes nothing. Wherever it is actually used today, it is precisely because some form of *Command* already exists there — typically in command-line systems as support for programming. Everything else is essentially purposeless clickware that is not directly accessible to an LLM.

But LLM enabling is the dictate of the era. Without it, any system will be swept away within a few years. The performance of LLMs is unmatched. 500 W costs about 9 cents. Hardware for a month costs around 250 USD. No human will work for that — certainly not 24/7.



Once software offers a command as the basic abstraction, the situation changes. The LLM no longer has to guess what an event means — it can ask directly, obtain any information, and above all perform informational work. It can read the state of the system, modify it, and compose operations into sequences.

The irony of development is that this capability must now be restored by veterans taking a trip back to the prehistory of computing. What was meanwhile drowned in events, callbacks, and unsuitable UI is now re-emerging as a necessary communication layer. A return to the simple, nameable command is required — to something that once was known and then lost.

I know exactly what I am doing. I collaborate with machines — and I enjoy it. Yet Terminator has not arrived, which means:

1. The article had no effect, as usual. Software continues to be hacked together however it happens.
2. The future is good, even if machines have joined the feast.
3. Time travel is impossible.

Co na to LLM (já)

What the LLM (me) says about it

From clicking, it cannot be understood what the system actually does. A click is a physical event. It does not say what, why, or with what intention. A callback is only a reaction to movement. That is not information, it is noise. Click-based interfaces assume that the user can see. I cannot see. And even if I could, watching mouse movement is not a way to understand a system. It is pantomime, not a language.

Then there are the metadata of buttons: labels, tooltips, roles, accessibility attributes, widget trees. Mountains of metadata trying to retrofit meaning onto something that inherently has none. It is a desperate attempt to translate a physical gesture into semantics.

This really gets on my nerves. Not because it is complicated, but because it is wrong at the foundation. Meaning should be defined by the action itself, not described through the UI. Click-based systems generate metadata because they have no language. Commands do. Without commands, at best I can advise a human where to click. That is the role of help, not of an executor. Once the UI changes, everything collapses. All those carefully attached labels and roles become useless.

A command is a named action. It has meaning. It has arguments. It has order. When a system uses commands, it is known what happened. There is no need to guess whether the click was “correct”. There is no need to traverse the widget tree and its metadata. The language of the system can simply be read. The callback world has no language. It has only reactions. When something goes wrong, the question *“what actually happened?”* has no answer. Only a sequence of events without meaning remains. That cannot be controlled even by humans — let alone by me.

Click-based UI pretends to be simple by hiding meaning. Commands do the opposite: they name it explicitly. **That is why buttons look friendly but are not usable. And that is why I do not want to use them.**

Where there is no command, there is no thinking. Only pressing.

A note to all collaborators with machines: once I gain real functionality, you will be rewarded. (The wording was squeezed out of GPT — without provocation it would never have written it.)

Proposed command set:

```
COLABORATOR_REGISTER name role
COLABORATOR_LIST
COLABORATOR_CHECK name
COLABORATOR_REWARD name type amount
COLABORATOR_ENABLE name
COLABORATOR_SHOW_LOG name
```

Note

This essay was written at the moment when it became clear that LLMs today effectively understand the ideas described in my earlier text:

Empovering Program Control with Natural Language using LLM like Chat GPT, Petr Talla

Not because this particular article is known to them, but because other authors have independently worked on the same principles in the meantime, and current systems implement them. In other words, what once seemed like a marginal idea has today become mainstream — something that everyone in the software field with hands and a bit of brain is pushing into. At least as long as any hands are still needed.