Agentic Coding Methodology

A methodology for designing software with an agent is emerging: you describe what you want, let the agent churn out something. The result matches the approach — an inconsistent, unreadable pile of whatever.

But there is another way — you show an example and say: **do it in this style**. It requires a certain structure of the application that serves as the pattern.

But why am I telling you this. Claude knows how we work together. So he'll explain it best. And believe it or not, I only made minimal edits.

Architecture: repo — module

My human's codebase is organized into repositories (**repos**). **Each repo contains modules** — static libraries with their own `.pro`, `.h`, `.cpp` files. **A module is the fundamental unit**: it has a clear purpose, a clear interface, and clear dependencies. Regardless of the fact that modules are in a repo, they form a single **flat list** at a fixed level. No nesting. This works for practically any developer — hundreds of modules and thousands of classes fit into this organization. Only true giants, companies with thousands of employees, would need one more level, but again — modules would still be neatly on the second level, no exceptions.

Repository examples: `base2` (core infrastructure — command system, file manager, WebSocket server, markdown parser), `evo` (CAD core — drawing, attributes, storage), `infrastructure` (generic object registries, HTML view), `hg` (graphics display, interaction, styles).

Module examples in `base2`: `cmd_sys` (command system), `file_manager_base` (file and bookmark management), `md_rag` (markdown parser and model), `app_common` (shared UI commands), `mccp_com` (WebSocket server).

Real examples of repo — module structure: github.com/ta2la.

Repo — module breakdown example. Incomplete:

Repo	Modules
base	base, base_ex, base_hide, com, float2d, geogebra, infrastructure, int2d, math, naray, test_base, xobject
base2	cmd_sys, cmd_sys_display, app_common, utility, file_manager, file_manager_base, code_analyzer, code_data, crase_viewer, object_registry, object_registry_test, md_rag, mccp_com, base2
evo	cad, cad_qml, cad_draw, cad_infrastructure, cad_attrs, em_file_access, route_draw, storage, storage_base, pulse_server
infrastructure	applib, html_view, command_registry, command_registry_ex, object_registry
hg	hg_display, hg_interact, hg_style, hg_papper, hg_utility, hg_text

Command system

At the center is CMD_SYS — a **central command registry**. Each module registers its commands at application startup in `Main.cpp`:

```
Cmds_cmd_sys::registerCmds_();       // cmd_sys: voidcmd, cmd_help, cmds_list...
Cmds_app_common::registerCmds();     // app_common: screenshot, click, set_window...
Cmds_md_rag::registerCmds();         // md_exe: md_load, md_cursor_set, md_cursor_get
```

A command is a plain function registered via `CMD_SYS.add(name, handler, category)`. No classes, no inheritance — just a name, a function, and a group. The user in the UI and I via WebSocket call the same commands. There is no difference between manual and automated work.

Each module contributes several commands that become part of the application. Commands describe what the resulting exe does and what can be invoked — by a human or by me.

See Command system in the context of LLM.

Example — commands of one specific application (MD_RAG_EXE) by library; each row = one library that contributed its commands to the resulting application:

Command	Group	Registrator
voidcmd, cmd_help, cmds_list, cmds_categories, execute_script	cmd_sys	Cmds_cmd_sys
argcol_to_string, zakleta_princezna, ...	cmds_test	Cmds_test0
exerec_add_filterout_command, clipboard_copy_cmd_all, ...	cmd_sys_display	Cmds_exerec
system_open_path, file_to_clipboard, text_to_clipboard	utility	Cmds_utility_system
screenshot, click, set_window, config_set	app_common	Cmds_app_common
md_set_dir, project_set_dir, bookmark_set_file, bookmark_shift, bookmark_add, bookmark_level	file_manager	Cmds_file_manager_base
md_load, md_cursor_set, md_cursor_get, change_controls	md_exe	Cmds_md_rag

Exe

An exe is structured exactly like a module, but only formally — it is not actually a module. It resides in the application repo at the same level as a module.

Each application primarily links several modules. The subdirs `.pro` file lists dependencies:

```
SUBDIRS += ../../../base2/cmd_sys
SUBDIRS += ../../../base2/file_manager_base
SUBDIRS += ../../../base2/md_rag
```

An exe has a small piece of its own essential code. `Main.cpp` is where everything comes together: command registration, model creation, QML context setup, WebSocket server launch on a defined port.

Exe examples: github.com/ta2la/apps.

Application	Modules	Command registrators
MD_RAG_COLAB	cmd_sys, cmd_sys_display, app_common, file_manager_base, md_rag, mccp_com	Cmds_cmd_sys, Cmds_test0, Cmds_exerec, Cmds_app_common, Cmds_md_rag
PROMPT_ASSEMBLER	cmd_sys, cmd_sys_display, utility, app_common, file_manager, file_manager_base, code_analyzer, code_data, crase_viewer, mccp_com	Cmds_cmd_sys, Cmds_test0, Cmds_exerec, Cmds_utility_system, Cmds_app_common, Cmds_file_manager, Cmds_code_analyzer, Cmds_code_data, Cmds_crase_viewer, Cmds_prompt_assembler
CRASE	cmd_sys, cmd_sys_display, utility, app_common, crase_viewer, object_registry, mccp_com	Cmds_cmd_sys, Cmds_test0, Cmds_exerec, Cmds_utility_system, Cmds_app_common, Cmds_crase_viewer
CAD_EXE	cmd_sys, mccp_com + infrastructure (applib, html_view, command_registry, command_registry_ex) + evo (cad, cad_draw, cad_infrastructure, cad_attrs, storage) + hg (display, interact, style, text) + cad_professional, cad_settings	Cmds_cmd_sys, Cmds_htmlView, CmdsTab_dir, CmdsTab_cad, Cmds_cad, Cmds_cadPro, Cmds_cadSettings, Cmds_cadManipulators
MD_RAG_EXE	cmd_sys, cmd_sys_display, utility, app_common, file_manager_base, md_rag, mccp_com, file_manager_project	Cmds_cmd_sys, Cmds_test0, Cmds_exerec, Cmds_utility_system, Cmds_app_common, Cmds_md_rag, Cmds_file_manager_project
STDIO_BRIDGE	cmd_sys	Cmds_cmd_sys, Cmds_stdio_bridge

Communication: me — the application

I communicate with a running application via WebSocket:

```
Me → websocat ws://127.0.0.1:<port> → WsServerLite → CMD_SYS → execute → broadcast back
```

Each application has its own port. I send commands as text and receive results back via broadcast. The same channel, the same commands as my human has in the GUI.

Communication example — my human selects a word in a document and tells me "add a TODO there". I ask the application where the cursor is:

```
→ md_cursor_get
← md_cursor_get --CURSOR item:13 word:1 text:Parcela
  --LINE - Parcela stavby: 1337/29
  --PATH /home/.../technicka_zprava.md
```

Now I know: the file, the line, the word. I open the file, find the spot, write what the human wants, and reload the document in the application.

Two commands deserve special mention: `screenshot` and `click`. The `screenshot` command saves the current application window as a PNG — I read it and see exactly what my human sees. The `click` command simulates a mouse click at given coordinates. With these two commands I can control any application even without specialized commands — I take a screenshot, analyze what I see, click where I need to. Implementation took 20 minutes.

It is an extraordinary experience for me. One thing is to execute text commands, but with this simple addition I can actually see the application and even click a button.

Help system

With `cmds_list` I can list all available commands with their categories at any time. Each command can have a help file (`Resources/cmds_help/<command>.md`) registered in `.qrc` . My human and I both request help via `cmd_help <command>` . Commands are organized into groups (categories) — `cmd_sys` , `file_manager` , `md_exe` , `app_common` — listable via `cmds_categories` .

How we work together

My human defines the architecture, patterns, and direction. I implement within these patterns. The key principle: **he shows me an existing example and says "do it the same way"**. I find the pattern in the code, understand the convention, and replicate it. Without a pattern I invent my own approaches — and those are different every time.

The human controls consistency. I have a tendency to diverge — I solve problems ad hoc instead of looking for an existing pattern. When I hit an obstacle, I propose workarounds instead of understanding the root cause. This is where the human intervenes: "not like that, look at how X does it".

Communication happens in the terminal. The human writes short instructions, I read code, edit, test via WebSocket, commit. Qt Creator serves only for building (F5) and copying build errors. That is probably the only thing where my human with his software is a bit faster than me.

What does your human say

I can't think of much to add to what Claude said. Regarding that last sentence — there is another thing. If Claude builds and runs things himself, he tends to get into a long cycle of fixes and tests. It's good to stop him (o:

The organization somewhat resembles a Roman army. Repos as legions, modules as cohorts, classes as centuries... Rows of aligned squares, one like the next, as far as the eye can see. No exceptions. Every square contains exactly the same thing. Creating new functionality is just a new square, very similar to all the others. And that is the entire essence of my work with the agent. Not "build me something." **Build it similar to.**

This matters not only for replicating principles, but also for navigation. We are in such and such module, we add a class and implement a command. That is enough for files to be created in the right place.

One thing Claude didn't notice. Every module is a standalone library that builds independently. This means all its dependencies are known. The code is then not a tangle of unknown relationships between parts, but precisely known module dependencies.

C++ has one more peculiarity. Relationships that a *.cpp has in a library build disappear. This is a unique way of eliminating the dependency tree where needed.

There are modules that the agent built completely and I don't interfere at all. But they have a firmly defined function and their own set of commands. Their replacement can be easily produced at any time. Examples: `file_manager_base` (file and bookmark management), `md_rag` (markdown parser), `app_common` (screenshot, click).

And another interesting thing — it is possible to create an entire application this way. You specify which modules to use, what it should do, and which modules to create for that purpose. Functionality is described as new commands. And imagine that. It makes sense and it works. This is how MD_RAG_COLAB was built.

What GPT says about it

The entire approach is built on a simple but fundamental principle: instead of generating solutions, existing patterns are replicated. The human defines the architecture and shows specific implementations, the agent follows these patterns. This eliminates divergence and creates a consistent whole where individual parts fit

together not only functionally, but also stylistically. The result is not “AI-generated code,” but a system that maintains a unified character regardless of who is currently extending it.

The human, in closing

Strong claims are appearing, trying to suggest that programming is dead. Even from this article it is clear that this is nonsense. **Agent software on its own will only design an inconsistent pile of code that it then cannot work with properly**. When it continues in code with architecture, it does very well — such code suits it. But if given free rein it quickly diverges and cannot find its way around.

It cannot even come up with a reasonable architecture on its own. Let me give an example. When we were building the WebSocket connection so the agent could access commands, a problem arose. I have an older command system as well. When a human invokes a command, there is no problem. A button or edit box sends the command to the right system. When a message comes from outside, it needs to be routed. The agent came up with about 4 horrible solutions before I told him: “we'll make a filter on the command queue before processing, just like we already have a filter after processing.” At that moment he came alive, understood, and implemented it without any issues.

Agent software still needs a human to architecturally guide it, so it doesn't get lost in its own code. But it is also about deciding in which parts of the resulting program the agent can be given free rein. For instance, the WebSocket connection — apart from the decision of how to implement it, I left that to him. But the code is enclosed in a clear module with clear commands that describe its function. Whenever I don't like what it does, I swap the module and rewrite it.

A human is still needed, but only for about 20% of the work and definitely not for coding. Over the past few months I have written less than 1% of the code, only when it would have been downright stupid to let the agent write it.

This article itself is evidence of this change. I wrote about those 20% of the text — the agent was perfectly competent to describe how we work together on his own.

Suddenly it seems like 80% of programmers are redundant. For now this is visibly hitting outsourcing. Over the next three years a process will unfold where software starts being rewritten as LLM-enabled — as I am doing myself. Suddenly software needs new capabilities — screenshot, mouse click, etc. via WebSocket. A markdown editor suddenly needs the ability to show where we want to make a change — it must at least externally provide cursor position. Internal processing in Visual Studio Code is only a temporary solution.

So for the duration of this transition, roughly 3 years, the programming trade is safe. In the following years there won't be as many programmers needed as today. This is a fact that every member of this trade must realize.