A contributing factor to legacy Vim's huge codebase is its explicit support for dozens of widget toolkits for GUI interfaces. Neovim avoids that by delegating GUI implementation to external clients. The client(s) control the Neovim `nvim` process via a msgpack-rpc API, allowing them to:

- Execute Vim commands
- Evaluate Vimscript expressions
- Manipulate buffers, windows and tabs
- Receive/handle editor events

On top of that, the remote API has been designed for easy extensibility, so there will always be new possibilities.

A Neovim *remote plugin* (`rplugin`) is any program that talks to `nvim` through the remote API (which can be reached via any arbitrary transport mechanism: TCP address, named pipe, stdin/stdout, ...).

It's possible to test the current API interactively using the python REPL and the client library, but that isn't very useful for extending the editor. A typical Neovim plugin will have the following pseudo code running:

```
while true
    handle(next_vim_event())
```

If you think about it, that's pretty much how legacy Vim plugins currently work: They register autocommands to be executed whenever something interesting happens. The difference is that Neovim can also propagate events to other processes and receive commands asynchronously, and this opens some interesting possibilities for plugins:

- They can be written in any programming language.
  - Modern GUIs written in high-level languages that integrate better with the operating system (e.g., C#/WPF on Windows or Ruby/Cocoa on OS X).
- They are sandboxed so there's less chance of the editor (server) crashing due to bugs.
- They can perform background work without blocking the editor
- They can emit **custom events** that may be handled directly by GUIs (enables implementation of advanced features such as Sublime's minimap).
- Multiple remote GUIs can attach/detach to share editing sessions.
- Simplified headless testing.
- Embedding the editor into other programs. In fact, a GUI can be seen as a program that embeds neovim.

UI programs UIs are plugins that work as a bridge between the user and the editor. Here's pseudo-code of a UI program:

```
while true
    handle(next_vim_or_user_event())
```

It's almost the same as a non-UI plugin, except they must also listen for user events (keypresses, mouse clicks, etc) and translate these to the connected Neovim instance, which then emits *redraw* events back to the user.

Here's a sample process tree:

```
Neovim ------> GUI 1 (attach/detach to running instance)
  |  |
  |   `------> GUI 2 (communicating on a different socket or transport
  |                   mechanism, but sharing the same session with GUI 1)
   `--> Plugin 1
  |
   `--> Plugin 2
  |
   `--> Plugin 3
```

Here's an outline of the `nvim` startup process:

1. start listening on a socket or TCP address (random unless overridden)
    - UI may be specified as a command-line argument to `nvim`
2. read `~/.config/nvim/init.vim`
    - discover plugins (which may include UI)
3. start the UI program, passing the listen address.
4. open two-way communication channels with all plugins

Here's a hypothetical GUI session:

```
gui -> vim: {"id": 1, "method": "initClient",
             "params": {"size": {"rows": 20, "columns": 25}}}
vim -> gui: {"id": 1, "result": {"clientId": 1}}
vim -> gui: {"method": "redraw",
             "params": {"clientId": 1, "lines": {"5": "Welcome to neovim!"}}}
gui -> vim: {"id": 2, "method": "keyPress",
             "params": {"keys": ["H", "e", "l", "l", "o"]}}
vim -> gui: {"method": "redraw",
             "params": {"clientId": 1, "lines": {"1": "Hello", "5": ""}}}
```