

LLM, MDD and Micro-Modularity

Dirk Engel, info@engel-internet.de, November 26, 2023

<https://github.com/dirkengel/>

Introduction

In "[Neither Human nor Machine](#)", I highlighted that in software development, ChatGPT can serve as an effective sparring partner and a valuable tool, providing the desired results as long as trust is maintained. The principle "Trust, but verify" comes into play. Verification can be achieved by applying the divide-and-conquer principle, breaking down tasks into smaller, manageable pieces, see "[Divide \(along interfaces\) and Conquer](#)". This allows for a detailed review, understanding, improvement, or approval of the responses generated by the Large Language Model (LLM) system.

Particularly when dealing with recurrent software problems and aiming for a generic solution by describing the task's mechanics at a meta-level, micro-modularity can be employed. This involves breaking down tasks into smaller components to create high-quality code templates. These templates, multiplied by a generator, influence code quality in various places ("[A Software Dependency Flip Book](#)"). Therefore, special attention to these code parts is crucial.

In terms of quality attributes, we can address this by requesting idiomatic implementations from the LLM in various programming languages to enhance portability. Moreover, we can target performance efficiency by requesting different code variants for the same snippet. When discussing "different", we are not limited to comparing a few alternative implementations; the LLM should empower us to perform a kind of brute force analysis of as many variants as possible. In essence, developers and their sparring partner can collaboratively explore and analyze a broad spectrum of code variations.

Example

Simple State Machine

The example is a small state machine written in C++. Rather than starting with a blank piece of paper, I chose to specify in prose how I envision the states and transitions. Additionally, I aim to allow the injection of guard conditions as predicate functions, transition actions, and state entry and exit action code. To complete the picture, some test code is also desirable. After a few iterations, this provides enough input for ChatGPT to generate code that can serve as a good starting point. Here is the [Source Code](#) after several manual *editorial* changes:

```
1 // file: fsm.h -> framework code
2
3 #include <functional>
4
5 class State {
6 public:
7     State() = default;
8     ~State() = default;
9
10    std::function<void()> entryAction {nullptr};
11    std::function<void()> exitAction {nullptr};
12 };
13
14 class Transition {
15 public:
16     Transition(State* from, State* to) : fromState(from), toState(to) {}
17     ~Transition() = default;
18
19     State* getFromState() const {return fromState;}
20     State* getToState() const {return toState;}
```

```

21
22     std::function<bool()> condition [[] {return true;}); // valid by default
23     std::function<void()> action {nullptr};
24
25 private:
26     State* fromState {nullptr};
27     State* toState {nullptr};
28 };
29
30 class StateMachine {
31 public:
32     StateMachine(State* initialState) : currentState {initialState} {}
33     ~StateMachine() = default;
34
35     bool doTransitionCallActions(const Transition& transition) {
36         bool success = false;
37         if (currentState == transition.getFromState() && transition.condition()) {
38             if (currentState->exitAction != nullptr) {
39                 currentState->exitAction();
40             }
41             if (transition.action != nullptr) {
42                 transition.action();
43             }
44             currentState = transition.getToState();
45             if (currentState->entryAction != nullptr) {
46                 currentState->entryAction();
47             }
48             success = true;
49         }
50         return success;
51     }
52
53 private:
54     State* currentState;
55 };
56
57 // file: light_fsm.h -> generated code
58
59 #include "fsm.h"
60
61 class LightFsm {
62 public:
63     LightFsm() = default;
64     ~LightFsm() = default;
65
66     bool turnOff() {
67         bool result = false;
68         if (not result) {
69             result = fsm.doTransitionCallActions(turnOff_on_off);
70         }
71         return result;
72     }
73
74     bool turnOn() {
75         bool result = false;
76         if (not result) {
77             result = fsm.doTransitionCallActions(turnOn_off_on);
78         }
79         return result;
80     }
81
82     State offState {};
83     State onState {};
84
85     Transition turnOff_on_off {Transition(&onState, &offState)};
86     Transition turnOn_off_on {Transition(&offState, &onState)};
87
88 private:
89     StateMachine fsm {StateMachine(&offState)};
90 };
91
92 // file light_fsm.cpp -> test or application code
93
94 #include <iostream>
95 #include "light_fsm.h"
96
97 int main() {
98     LightFsm lightFsm;
99
100     // Inject actions
101     lightFsm.offState.entryAction = [] {std::cout << "entering off state" << std::endl;};
102     //lightFsm.offState.exitAction = [] {std::cout << "exiting off state" << std::endl;};
103
104     //lightFsm.onState.entryAction = [] {std::cout << "entering on state" << std::endl;};
105     lightFsm.onState.exitAction = [] {std::cout << "exiting on state" << std::endl;};
106
107     lightFsm.turnOn_off_on.action = [] {std::cout << "turning on" << std::endl;};
108     lightFsm.turnOff_on_off.action = [] {std::cout << "turning off" << std::endl;};
109
110     // Simulate events
111     lightFsm.turnOn();
112     lightFsm.turnOff();

```

```

113     std::cout << "off is off" << std::endl;
114     lightFsm.turnOff();
115
116     return 0;
117 }

```

Model

Model-Driven Development (MDD) requires a model as input for the code generator. Therefore, ChatGPT's next task is to reverse engineer a PlantUML state diagram that matches the above code. Here is both the model [Source Code](#) and its graphical representation:

```

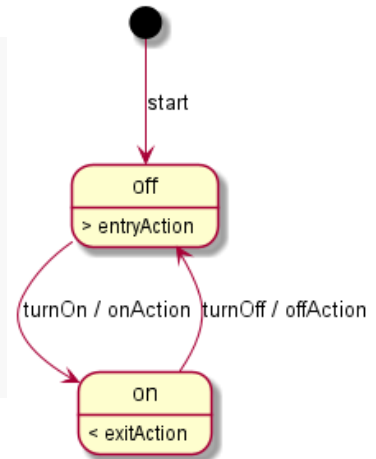
@startuml
[*] --> off : start
off --> on : turnOn / onAction
on --> off : turnOff / offAction

state off {
    off : > entryAction
}

state on {
    on : < exitAction
}

@enduml

```



Metamodel

The model is, of course, not enough; what is needed is a metamodel. This metamodel should contain the state and transition information found in the PlantUML models. While a Python script is a straight forward approach to achieve this, it is more conventional to ask the AI to provide that script rather than writing it manually. After a few adjustments we have the following script ([Source Code](#)):

```

1  #!/usr/bin/python3
2
3  import re
4  import argparse
5
6  def parse_plantuml(plantuml_code):
7      state_pattern = re.compile(r'state (\w+)\s*{')
8      entry_action_pattern = re.compile(r'\s*(\w+)\s*:\s*>\s*(.+)')
9      exit_action_pattern = re.compile(r'\s*(\w+)\s*:\s*<\s*(.+)')
10     transition_pattern = re.compile(r'(\w+)\s*-->\s*(\w+)\s*:\s*(\w+)\s*(\[.*\])?\s*/\s*(.+)')
11     init_pattern = re.compile(r'\[\*\]\s*-->\s*(\w+)')
12
13     states = []
14     transitions = []
15
16     lines = plantuml_code.split('\n')
17     current_state = None
18
19     for line in lines:
20         state_match = state_pattern.match(line)
21         if state_match:
22             current_state = state_match.group(1)
23             states.append({'name': current_state, 'entry_action': None, 'exit_action': None})
24
25         entry_action_match = entry_action_pattern.match(line)
26         if entry_action_match and current_state:
27             states[-1]['entry_action'] = entry_action_match.group(2)
28
29         exit_action_match = exit_action_pattern.match(line)
30         if exit_action_match and current_state:
31             states[-1]['exit_action'] = exit_action_match.group(2)
32
33         transition_match = transition_pattern.match(line)
34         if transition_match:
35             from_state = transition_match.group(1)
36             to_state = transition_match.group(2)
37             event = transition_match.group(3)
38             action = transition_match.group(4)
39             transitions.append({'from_state': from_state, 'to_state': to_state, 'event': event, 'action': action})
40
41         init_match = init_pattern.match(line)
42         if init_match:

```

```

43     init_state = init_match.group(1)
44
45     return states, transitions, init_state
46
47 def events(transitions):
48     events = set(transition['event'] for transition in transitions)
49     return list(events)
50
51 def transitions_for_event(transitions, event):
52     return [transition for transition in transitions if transition['event'] == event]
53
54 def indent(depth):
55     return " " * depth * 3
56
57 def generate_cpp(name, states, transitions, init_state):
58     pass # see Section 'Generator'
59
60 def generate_rust(name, states, transitions, init_state):
61     pass # see Section 'Generator'
62
63 def main():
64     parser = argparse.ArgumentParser(description="Parse PlantUML file and print collected information.")
65     parser.add_argument("plantuml_file", help="Path to the PlantUML file")
66     parser.add_argument("-l", "--language", choices=["c++", "rust"], help="Select between C++ and Rust")
67     args = parser.parse_args()
68
69     plantuml_file = args.plantuml_file
70
71     try:
72         with open(plantuml_file, 'r') as file:
73             plantuml_code = file.read()
74             name = plantuml_file.split('.')[0]
75             states, transitions, init_state = parse_plantuml(plantuml_code)
76
77             # Print collected information
78             print("Name:", name)
79             print("Initial State:", init_state)
80
81             print("\nStates:")
82             for state in states:
83                 print(state)
84
85             print("\nTransitions:")
86             for transition in transitions:
87                 print(transition)
88
89             if args.language is not None:
90                 language = args.language.upper()
91                 if language == "C++":
92                     print("\nC++:\n")
93                     generate_cpp(name, states, transitions, init_state)
94                 elif language == "RUST":
95                     print("\nRust:\n")
96                     generate_rust(name, states, transitions, init_state)
97
98     except FileNotFoundError:
99         print(f"Error: File '{plantuml_file}' not found.")
100     exit(1)
101
102 if __name__ == "__main__":
103     main()

```

Generator

The following two subsections list the source code of the language specific generator functions `generate_cpp()` and `generate_rust()`, respectively.

Variant C++

The `fsm.h` header represents the static part of the simple FSM framework, while `light_fsm.cpp` serves as test or application code. In our example, the actual business logic generated from a model resides in `light_fsm.h`. The following function `generate_cpp()` extends the metamodel script to produce the desired `light_fsm.h` code ([Source Code](#)):

```

1 def generate_cpp(name, states, transitions, init_state):
2     print("#include \"fsm.h\"\n")
3     print("class " + name + " {")
4     print("public:")
5     print(indent(1) + name + "() = default;")
6     print(indent(1) + "~" + name + "() = default;\n")
7     transition_members = set()
8     for e in events(transitions):
9         print(indent(1) + "bool " + e + "() {")
10        print(indent(2) + "bool result = false;")

```

```

11     ets = transitions_for_event(transitions, e)
12     for et in ets:
13         print(indent(2) + "if (not result) {"")
14         transition_name = et['event'] + "_" + et["from_state"] + "_" + et["to_state"]
15         transition_members.add((transition_name, et["from_state"], et["to_state"]))
16         print(indent(3) + "result = fsm.doTransitionCallActions(" + transition_name + ");")
17         print(indent(2) + "}")
18     print(indent(2) + "return result;")
19     print(indent(1) + "}")
20     for s in states:
21         print(indent(1) + "State " + s['name'] + "State {};"")
22     print()
23     for t in transition_members:
24         print(indent(1) + "Transition " + t[0] + " {Transition(&" + t[1] + "State, &" + t[2] + "State)};"")
25     print("\nprivate:")
26     print(indent(1) + "StateMachine fsm {StateMachine(&" + init_state + "State)};"")
27     print("};")

```

Variant Rust

Now, with the language-independent model and a generator based on the metamodel available, addressing the portability quality attribute, especially when leveraging an LLM, seems obvious. Creating the Rust code with ChatGPT help was significantly more time-consuming compared to C++; it took quite a while to even compile. Well, I am sure this will get much better with more training data. However, I am less concerned with the results of the LLM and more concerned with my own superficial knowledge of this language. It is hard or even not possible to evaluate the results, push the LLM output in the right direction, or modify the code according to my own concepts if I do not have a complete oversight of all the language capabilities and specifics. This is not the right condition to produce quality. The idea of micro-modularity is not fulfilled, as I cannot assert that I clearly understand all possible side effects and consequences of the Rust code. My personal lessons learned: a deep dive into Rust is next on my agenda. The example [Source Code](#):

```

1 // file: fsm.rs -> framework code
2
3 pub mod fsm {
4     use std::collections::HashMap;
5     use std::rc::Rc;
6
7     pub struct State {
8         pub entry_action: Option<Box<dyn Fn()>>,
9         pub exit_action: Option<Box<dyn Fn()>>
10     }
11
12     impl State {
13         fn new() -> Self {
14             Self {
15                 entry_action: None,
16                 exit_action: None
17             }
18         }
19     }
20
21     #[derive(Clone)]
22     pub struct Transition {
23         from_state: String,
24         to_state: String,
25         pub condition: Rc<dyn Fn() -> bool>,
26         pub action: Option<Rc<dyn Fn()>>
27     }
28
29     pub struct StateMachine {
30         current_state: String,
31         pub states: HashMap<String, State>,
32         pub transitions: HashMap<String, Transition>
33     }
34
35     impl StateMachine {
36         pub fn new(initial_state: &str) -> Self {
37             let states = HashMap::new();
38             let transitions = HashMap::new();
39             //states.insert(String::from(initial_state), State::new());
40
41             Self {
42                 current_state: String::from(initial_state),
43                 states,
44                 transitions
45             }
46         }
47
48         pub fn add_state(&mut self, name: &str) {

```

```

49     self.states.insert(String::from(name), State::new());
50 }
51
52 pub fn add_transition(&mut self, name: &str, from_state: &str, to_state: &str) {
53     self.transitions.insert(String::from(name), Transition {
54         from_state: String::from(from_state),
55         to_state: String::from(to_state),
56         condition: Rc::new(|| true),
57         action: None
58     });
59 }
60
61 pub fn do_transition_call_actions(&mut self, transition: Transition) -> bool {
62     if self.current_state == transition.from_state && (transition.condition)() {
63         if let Some(exit_action) = self.states.get(&self.current_state).unwrap().exit_action.as_ref() {
64             exit_action();
65         }
66         if let Some(transition_action) = transition.action.as_ref() {
67             transition_action();
68         }
69         self.current_state = transition.to_state.clone();
70         if let Some(entry_action) = self.states.get(&self.current_state).unwrap().entry_action.as_ref() {
71             entry_action();
72         }
73         return true;
74     }
75     false
76 }
77 }
78 }
79
80 // file light_fsm.rs -> generated code
81
82 pub mod light_fsm {
83     use crate::fsm::fsm::StateMachine;
84
85     pub struct LightFsm {
86         pub fsm: StateMachine
87     }
88
89     impl LightFsm {
90         pub fn new(initial_state: &str) -> Self {
91             let mut fsm = StateMachine::new(initial_state);
92
93             fsm.add_state("off");
94             fsm.add_state("on");
95             fsm.add_transition("turnOff_on_off", "on", "off");
96             fsm.add_transition("turnOn_off_on", "off", "on");
97             Self { fsm }
98         }
99
100         pub fn turn_off(&mut self) -> bool {
101             let mut result = false;
102             if !result {
103                 let transition = self.fsm.transitions["turnOff_on_off"].clone();
104                 result = self.fsm.do_transition_call_actions(transition)
105             }
106             result
107         }
108
109         pub fn turn_on(&mut self) -> bool {
110             let mut result = false;
111             if !result {
112                 let transition = self.fsm.transitions["turnOn_off_on"].clone();
113                 result = self.fsm.do_transition_call_actions(transition)
114             }
115             result
116         }
117     }
118 }
119 }
120
121 // file main.rs -> test or application code
122
123 mod fsm;
124 mod light_fsm;
125
126 use std::rc::Rc;
127 use crate::light_fsm::light_fsm::LightFsm;
128
129 fn main() {
130     let mut light_fsm = LightFsm::new("off");
131
132     // Inject actions
133     light_fsm.fsm.states.get_mut("off").unwrap().entry_action = Some(Box::new(|| println!("entering off state")));
134     //light_fsm.fsm.states.get_mut("off").unwrap().exit_action = Some(Box::new(|| println!("exiting off state")));
135
136     //light_fsm.fsm.states.get_mut("on").unwrap().entry_action = Some(Box::new(|| println!("entering on state")));
137     light_fsm.fsm.states.get_mut("on").unwrap().exit_action = Some(Box::new(|| println!("exiting on state")));

```

```

138
139 light_fsm.fsm.transitions.get_mut("turnOn_off_on").unwrap().action = Some(Rc::new(|| println!("turning
on")));
140 light_fsm.fsm.transitions.get_mut("turnOff_on_off").unwrap().action = Some(Rc::new(|| println!("turning
off")));
141
142 // Simulate events
143 light_fsm.turn_on();
144 light_fsm.turn_off();
145 println!("off is off");
146 light_fsm.turn_off();
147 }

```

The following function `generate_rust()` extends the metamodel script to produce the desired `light_fsm.rs` code ([Source Code](#)):

```

1 def change_case(str): # https://www.geeksforgeeks.org/python-program-to-convert-camel-case-string-to-snake-
case/
2 res = [str[0].lower()]
3 for c in str[1:]:
4     if c in ('ABCDEFGHIJKLMNOPQRSTUVWXYZ'):
5         res.append('_')
6         res.append(c.lower())
7     else:
8         res.append(c)
9 return ''.join(res)
10
11 def generate_rust(name, states, transitions, init_state):
12 print("pub mod " + change_case(name) + " {")
13 print(indent(1) + "use crate::fsm::fsm::StateMachine;\n")
14 print(indent(1) + "pub struct " + name + " {")
15 print(indent(2) + "pub fsm: StateMachine")
16 print(indent(1) + "};\n")
17 print(indent(1) + "impl " + name + " {")
18 print(indent(2) + "pub fn new(initial_state: &str) -> Self {")
19 print(indent(3) + "let mut fsm = StateMachine::new(initial_state);\n")
20 for s in states:
21     print(indent(3) + "fsm.add_state(\"" + s['name'] + "\");")
22 for e in events(transitions):
23     ets = transitions_for_event(transitions, e)
24     for et in ets:
25         transition_name = et['event'] + "_" + et["from_state"] + "_" + et["to_state"]
26         print(indent(3) + "fsm.add_transition(\"" + transition_name + "\", \"" + et["from_state"] + "\", \"" +
et["to_state"] + "\");")
27 print(indent(3) + "Self { fsm }")
28 print(indent(2) + "};\n")
29 for e in events(transitions):
30     ets = transitions_for_event(transitions, e)
31     print(indent(2) + "pub fn " + change_case(e) + "(&mut self) -> bool {")
32     print(indent(3) + "let mut result = false;")
33     for et in ets:
34         transition_name = et['event'] + "_" + et["from_state"] + "_" + et["to_state"]
35         print(indent(3) + "if !result {")
36         print(indent(4) + "let transition = self.fsm.transitions[\"" + transition_name + "\"].clone();")
37         print(indent(4) + "result =self.fsm.do_transition_call_actions(transition)")
38         print(indent(3) + "}")
39         print(indent(3) + "result")
40     print(indent(2) + "};\n")
41 print(indent(1) + "}")
42 print("}")

```

Conclusion

In this article, I propose an innovative ([Creative, More Creative, Most Creative](#)) approach to combine modern AI techniques with traditional concepts like MDD to advance the ongoing pursuit of high-quality code. Rather than tasking the LLM with generating extensive, cohesive pieces of code that may become less comprehensible to developers involved, the core idea involves breaking down larger modules into micro-modules – components so small that the code can be understood in minutes.

These smaller fragments can then be optimized with the assistance of an LLM and assessed in isolation. Model-Driven Development (MDD) could serve as a method to systematically reintegrate generalized and optimized micro-modules back into the original larger context.

References

- Source Code *Source code of this article*, Nov, 2023. Available on [GitHub](#).
- Engel23a Engel, Dirk, *Neither Human nor Machine*, May 2, 2023. Shared on [LinkedIn](#) and available on [GitHub](#).
- Engel23b Engel, Dirk, *Creative, More Creative, Most Creative*, July 20, 2023. Shared on [LinkedIn](#) and available on [GitHub](#).
- Engel23c Engel, Dirk, *Divide (along interfaces) and Conquer*, August 22, 2023. Shared on [LinkedIn](#) and available on [GitHub](#).
- Engel23d Engel, Dirk, *A Software Dependency Flip Book*, October 31, 2023. Shared on [LinkedIn](#) and available on [GitHub](#).