Best Practices, Part I

Christopher R. Genovese

Department of Statistics & Data Science

Thu 18 Sep 2025
Session #8

Plan

State Machines and the Builder Pattern Revisited

Plan

State Machines and the Builder Pattern Revisited

Building State Machines

Plan

State Machines and the Builder Pattern Revisited

Building State Machines

Style Practices and Principles

Announcements

- **Homework**:
  - **migit** assignment Tasks #1–#10 due Tue 23 Sep. Available on github problem bank.
  - Push mini-assignment **state-machines** when done

Goals for Today

Last time, we did an activity in object-oriented design. Today, we will spend some time wrapping up that activity.

Then, we will turn to principles of software design and style.

Plan

State Machines and the Builder Pattern Revisited

Building State Machines

Style Practices and Principles

Making a Package

Python:

```
uv init --lib state-machines
```

Look at the result and update the configs.

R:

```
install.packages("devtools")
install.packages("usethis")

usethis::create_package("state_machines")
```

Look at the result and update the configs.

Paying Attention to the Types

Notice how paying attention to the types of the components in your State Machine helps you keep the logic straight

Let's consider the types for the basic `StateMachine`, excluding selectors and guards for the moment.

Paying Attention to the Types

Let's start with a top-level view of the `StateMachine`. Here, the type `Dict k v` is a dictionary mapping keys of type `k` to values of type `v`.

```
data StateMachine =
  record StateMachine where
    currentState : State
    states       : List State
    transitions  : Dict TransitionId Transition
    events       : Dict EventType (Set TransitionId)
    stateActions : Dict (StateActionTiming, State) StateAction
    transActions : Dict (TransActionTiming, TransitionId) TransitionAction
```

Paying Attention to the Types

Let's start with a top-level view of the `StateMachine`. Here, the type `Dict k v` is a dictionary mapping keys of type `k` to values of type `v`.

```
data StateMachine =
  record StateMachine where
    currentState : State
    states       : List State
    transitions  : Dict TransitionId Transition
    events       : Dict EventType (Set TransitionId)
    stateActions : Dict (StateActionTiming, State) StateAction
    transActions : Dict (TransActionTiming, TransitionId) TransitionAction
```

We can give concrete definitions of all the component types.

Paying Attention to the Types

We can give concrete definitions of all the component types.

```
State : Type = String                -- These are type aliases
TransitionId : Type = String
EventType : Type = String

data Transition = record Transition where
                    source : State
                    target : State
                    name   : TransitionId    -- discretionary

data Event = record Event where
               type : EventType
               payload : payloadType type    -- payload type varies with event type

data StateActionTiming = Exit | Enter
data TransActionTiming = Before | During | After

StateAction : Type = Event -> State -> StateMachine -> ()
TransitionAction : Type = Event -> Transition -> StateMachine -> ()
```

Paying Attention to the Types

We can give concrete definitions of all the component types.

```
State : Type = String                -- These are type aliases
TransitionId : Type = String
EventType : Type = String

data Transition = record Transition where
                    source : State
                    target : State
                    name   : TransitionId     -- discretionary

data Event = record Event where
               type : EventType
               payload : payloadType type     -- payload type varies with event type

data StateActionTiming = Exit | Enter
data TransActionTiming = Before | During | After

StateAction : Type = Event -> State -> StateMachine -> ()
TransitionAction : Type = Event -> Transition -> StateMachine -> ()
```

Code Demo

Constructing the State Machine

Debrief and Demo

Handling Action Dispatch

Dispatch to all the actions follows the same pattern:

```
check if the particular action has been defined
if so, execute it; if not, do nothing
```

This might look like:

```
if (ENTER, state) in self.state_actions:
    action = self.state_actions[ENTER, state]
    action(event, state, self)
```

In R, using lists, it is similar

```
action_key = make_key(ENTER, state)
if ( action_key %in% names(self.state_actions) ) {
    action = self.state_actions$action_key
    action(event, state, self)
}
```

where

```
make_key <- function(...) {
    components <- list(...)
    return( paste(components, collapse=",") )
}
```

Handling Action Dispatch

We can streamline this pattern somewhat by using a **sentinel**, a default value that performs as desired in the default case.

Suppose we define two functions (*outside our class*):

```
def noop_state_action(_event, _state, _machine):
    pass

def noop_trans_action(_event, _transition, _machine):
    pass
```

Now, in our `dispatch` function, we look up in the table in a way that returns a default if a key is not present. For example:

```
action = self.trans_actions.get((BEFORE, transition['name']), noop_trans_action)
action(event, transition, self)

action = self.state_actions.get((EXIT, state), noop_state_action)
action(event, state, self)
```

We simply do this multiple times for the appropriate keys!

State Machine Dispatch

Debrief and Demo

State Machine Applications

We will build several examples of state machine use that are interesting: matching patterns, UI actions (e.g., todo list), bounded queue, ...

Plan

State Machines and the Builder Pattern Revisited

Building State Machines

Style Practices and Principles

The Builder Pattern

We separated the construction of the `StateMachine` from the logic needed to build and validate it for three reasons:

1. It allows the state machine be simple and lean with the logic of the machine primary, separating it from the complexity of specification and validation and testing.
2. We might build a state machine in several ways (e.g., in code, by parsing a specification, deserialization), and we don't necessarily want all of those supported in the State Machine itself, or even in the same place.
3. This allows us to implement and test the State Machine much more quickly.

The Builder Pattern (cont'd)

We specify a state machine by defining states, transitions, events, actions, along with guards and selectors if appropriate.

We will do this using the `StateMachineBuilder` class. So for instance, this might look like

```
machine = (
    StateMachineBuilder()
    .add_states('a', 'b', 'c', 'd')
    .add_transition('a', 'b')       # Name defaults to 'a -> b'
    .add_transition('c', 'd', name='foo')
    .add_action('enter', 'c', on_c_func)
        #...
    .build()
)
```

or in R

```
machine = StateMachineBuilder() |>
            add_states('a', 'b', 'c', 'd') |>
            add_transition('a', 'b', name='ok') |>       # Name defaults to 'a -> b
            add_transition('c', 'd', name='foo') |>
            add_action('enter', 'c', on_c_func)  |>
                #... |>
            build()
```

The `build` method validates the specification and creates the `StateMachine` object. You should design the Builder interface to be clear and easy to use but flexible enough for reasonable

Plan

State Machines and the Builder Pattern Revisited

Building State Machines

Style Practices and Principles

Best Practices

Programming is a form of communication to *two* audiences: the computer and human readers (including future you).

As long as your code is syntactically correct, the computer will run it, for better or worse. But your code will be checked, studied, tested, documented, debugged, used, modified, generalized, and reused by humans. To get the most value from your time spent programming, you need to pay attention to how *humans* process your code.

Indeed, the features that characterize good code are very similar in spirit to the features that characterize good writing.

There are many details to manage in a complex piece of code, and consequently there are many detailed choices to consider in practice. These include matters of

- style
- *naming*
- documentation
- organization
- design
- dependence
- error handling
- tooling

See our rubric or e.g., the book *Code Complete* by Steve McConnell. These are worth reading and studying.

But for our purposes today, we can cover a lot of ground with only **a few basic principles**.

Write Code to be Read

Code *communicates ideas* and *describes abstractions*, often complicated ones. Its execution is like an unfolding story, with characters traveling along their own narrative arcs.

Try to maximize the **ease** and **clarity** with which the reader can process the code. Help them to

- understand the ideas/abstractions behind the code
- identify the characters/entities involved, and
- follow the story.

A good principle for writing/presentation: **prepare your reader for the information you are about to give**.

Here are a few implications of this principle:

- Format your code to make it easy to read
- Use meaningful, concrete, and descriptive *names*
- Arrange your code to bring out the central idea in each chunk
- Make critical relationships salient
- Structure your interfaces to present a clean and consistent abstraction
- Avoid hidden side effects and obscure features
- Use documentation to supplement code not mimic it

Write Code to be Read: Documentation

Give readers an entry point for understanding how the code is used, how data flows, et cetera. Examples and description can help, even if brief. Section labels and pointers to entry points can be helpful. Tests are a form of documentation too.

Use docstrings/structured comments for nontrivial functions.

The code can impart meaning on details – on the how – but not as easily on the why or when. Comments help there.

If you write a *clever* piece of code, first ask if you *need to be clever* and if so, consider documenting the goals of the code, constraints, reasons, etc.

Task for sharpening: Go to an open-source repository that interests you and **read the code**. What works for you? What doesn't? What *makes you work*?

Be consistent

A *foolish* consistency may be the hobgoblin of little minds, but for programming, a practical consistency is helpful to in many ways.

Here are a few implications of this principle.

- Use consistent *formatting, spacing, and style*
- Use consistent *naming schemes* for variables, functions, classes, and files (`CamelCase`, `kebab-case`, `snake_case`, `ALL_CAPS`)
- Use consistent documentation formatting, style, and scope
- Use consistent *interfaces* to functions and classes
- Use consistent error handling

Many conventions for naming, formatting, spacing, etc. are included in *style guides* used by projects or programming languages.

For example, PEP 8 describes naming and formatting conventions for Python code, and *your code will be expected to follow it*. (PEP 8 is unusual because nearly every major Python project uses it.) R has a lot of historical cruft that means nobody uses the exact same style, but the tidyverse style guide is a good reference. Read these guides!

Don't Repeat Yourself

Seriously, don't repeat yourself. It's inefficient to repeat yourself. So don't do it. Really.

Keep your code DRY! (Not WET – wasting everyone's time!)

Each piece of knowledge embodied in the code should have one unambiguous and authoritative representation.

Here are a few implications of this principle.

- If you find yourself repeating a piece of code, put it in a function.
- If you find yourself using a number or other literal, make it a *named constant*. (Besides a few basic cases such as 0, 1.)
- Documentation should not merely repeat what the code does but should add value. For instance: why, who, when?

It's easier to chew small pieces

Any stretch of code focuses on a few key ideas. Organizing your code to bring out one idea at a time, rearranging as needed.

- Organize your code modularly (paragraphs, functions, files)
- Prefer functions that do *one* thing well
- Prefer orthogonality (decoupling)
- Prefer functions/classes/modules with a distinct purpose and identity

Coupling: Consider how a change in your code/design/interface/... will cascade through your whole codebase.

Keep the contract clear

Each function or class has an explicit contract behind it. *"I give you this, you give me that."*

Make that contract salient in your code, your *names*, your tests, and your documentation.

An implication: **separate calculations, actions, and data** (Referential transparency is a good goal in any paradigm.)

An idea we will discuss: using language features to enforce this contract (from types to assertions to explicit pre/post conditions).

Keep information on a need to know basis

Each function, class, and module in your code needs some information to do its job.

Give it the information it needs but no more.

Giving too much information couples parts of the code that should be independent, making them harder to test, debug, and reason about.

Objects in particular should "**encapsulate**" the information they contain quite jealously.

Make it run, make it right, make it fast – in that order

Only optimize the bottlenecks! This is a *data-driven* process.

> *The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.*
>
> *– Donald Knuth, The Art of Computer Programming*

The End