When to (not) use type state

When to (not) use type state*

*pattern (1)

About me

- Name: Matthias Farnbauer-Schmidt (he/him)
- Started using Rust in summer 2018
- Software Developer @ Paessler AG since 07.2021
- Lead maintainer of internal crates
- Host of our internal Paessler Rust Meetup

Online:

- GitHub: @MattesWhite
- URLO: farnbams

PAESSLER
THE MONITORING EXPERTS

Quick facts – ABOUT PAESSLER

- Over 400 employees from over 40 countries
- Customers in >190 countries all over the world
- 82% of fortune 200 companies worldwide use PRTG
- More than 500,000 users rely on PRTG every day
- US is the largest market – followed by DACH, UK and Benelux
- APAC is a growth region with high investments ongoing

THE MONITORING EXPERTS

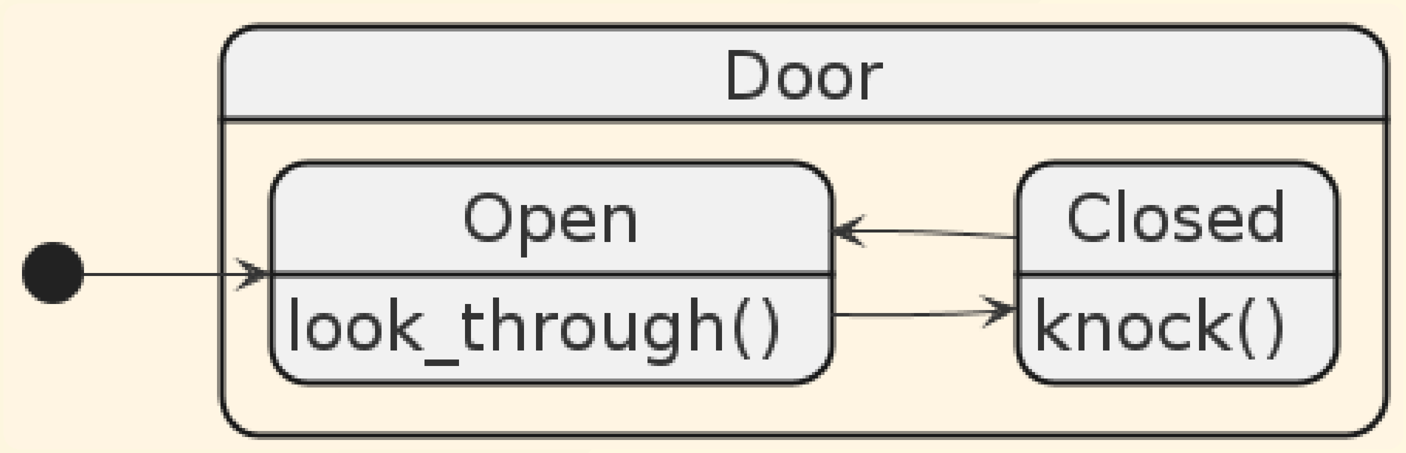
Founded in 1997 in Germany

What is typestate?

1. Encode *run-time* state in *compile-time* types
2. Operations are only available in associated states
3. Using operations that are not available in a state cause *compile-time* errors
4. State transitions make the previous state unaccessible

(2)

Implementation



demo 🎉

When to use typestate?

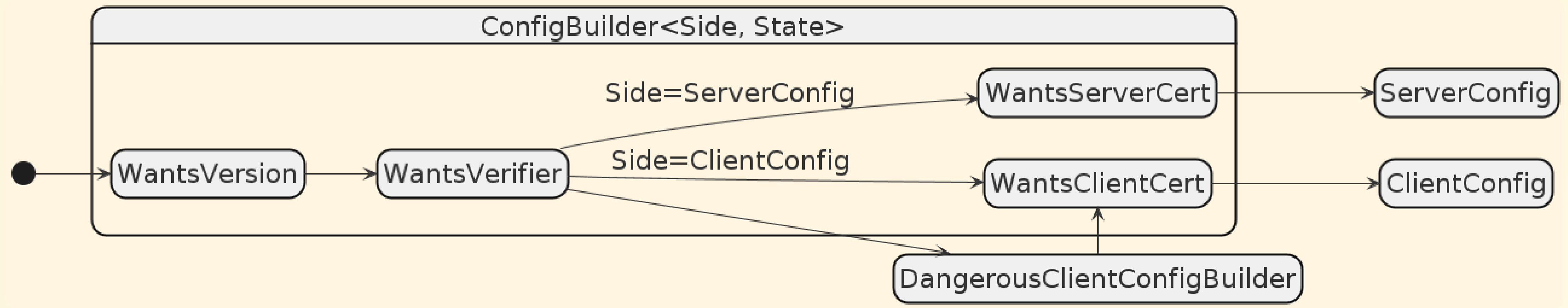
Benefits

- ✚ Encode application logic in the type system
- ✚ Move errors from *run-time* to *compile-time*
- ✚ Remove *run-time* checks
- ✚ Good IDE integration
- ✚ Self documentation
- ✚ Helps with Compiler-Driven-Development (3)
- ✚ Enforce order of operation

When to use typestate?

+ Builder pattern, e.g. `rustls::ConfigBuilder`

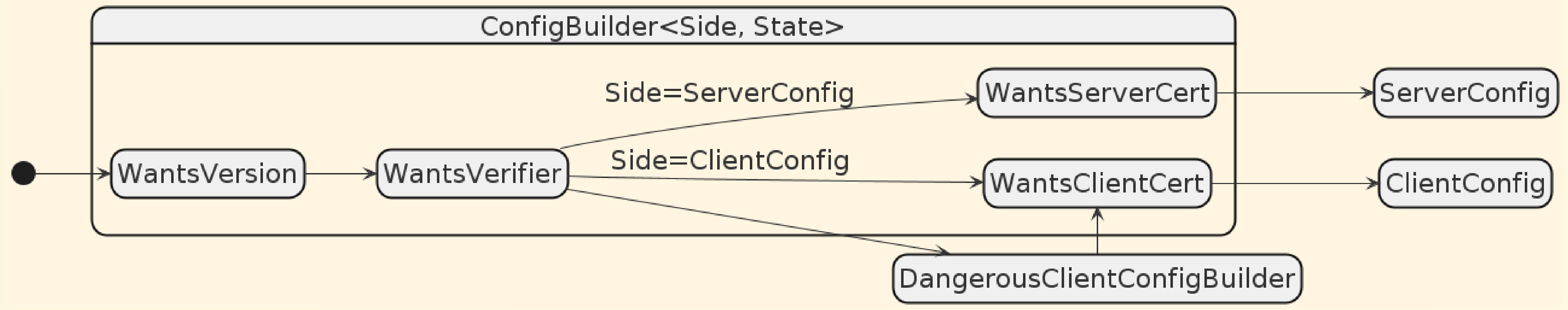
```
struct ConfigBuilder<Side: ConfigSide, State> { ... }
```



When to use typestate?

+ Builder pattern, e.g. `rustls::ConfigBuilder`

```
struct ConfigBuilder<Side: ConfigSide, State> { ... }
```



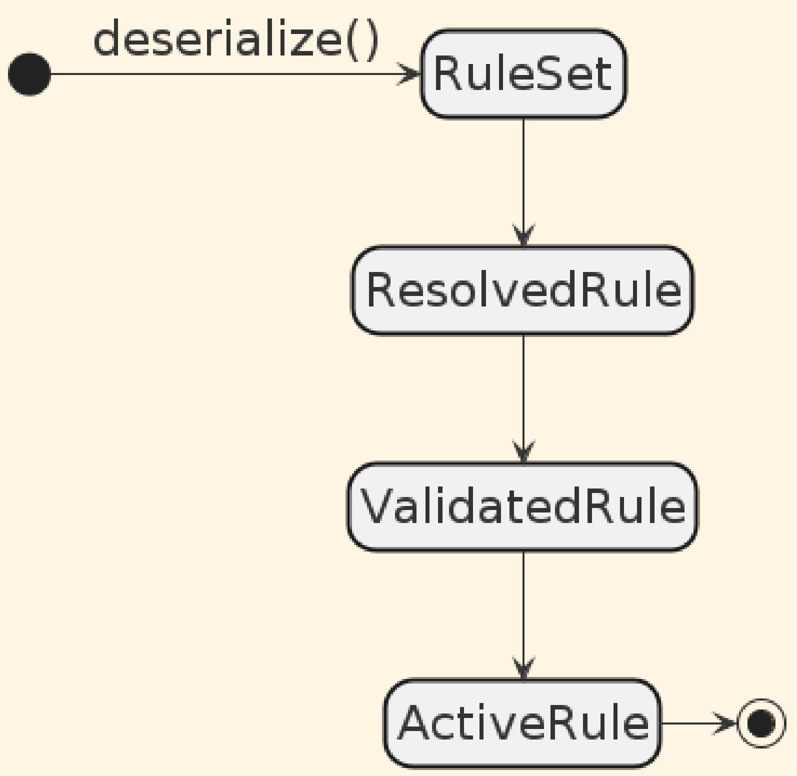
```
config.dangerous().with_custom_certificate_verifier(my_cert_verifier)
```

When to use typestate?

+ Transformation and validation, e.g. user input

```
receivers:
  syslog_port:
    type: udp
    listener: "0.0.0.0:514"

rules:
  forward-syslog:
    receiver: syslog_port
    destination:
      type: udp
      host: syslog.store.org:1514
```


$$\begin{tikzcd}
\bullet \arrow[r, "\text{deserialize()}"] & \boxed{\text{RuleSet}} \arrow[d] \\
& \boxed{\text{ResolvedRule}} \arrow[d] \\
& \boxed{\text{ValidatedRule}} \arrow[d] \\
& \boxed{\text{ActiveRule}} \arrow[r] & \odot
\end{tikzcd}$$

When to not use typestate?

Disadvantages

- Rare pattern in programming
- High entry level
- Combinatorial explosion
- High coupling between states
- Treacherous confidence
- Dynamic transitions are hard

When to not use typestate?

- Externally triggered transitions, e.g. network connections

```
impl Connection<Connected> {
    fn send(
        self, payload: &[u8],
    ) -> Result<Self, (Connection<Disconnected>, Error)> { ... }
}
let conn = match conn.send(b"Hello, RustFest!") {
    Ok(conn) => conn,
    Err((disconn, err)) => {
        warn!(%err, "disconnected, try reconnect");
        disconn.connect()?
    },
};
```

Conclusion

The benefits of typestate mainly apply to users

The disadvantages of typestate mainly affect crate authors

➡️ Do proper integration tests when using typestate

Resources

- (1) Mechanisms for compile-time enforcement of security, Robert E. Storm, 1983, doi: 10.1145/567067.567093
- (2) The Typestate Pattern in Rust, Cliff L. Biffle, 2019
- (3) Don't just test your code: MODEL IT, No Boilerplate, 2024
- (4) Pretty State Machine in Rust, Ana Hoverbear, 2016