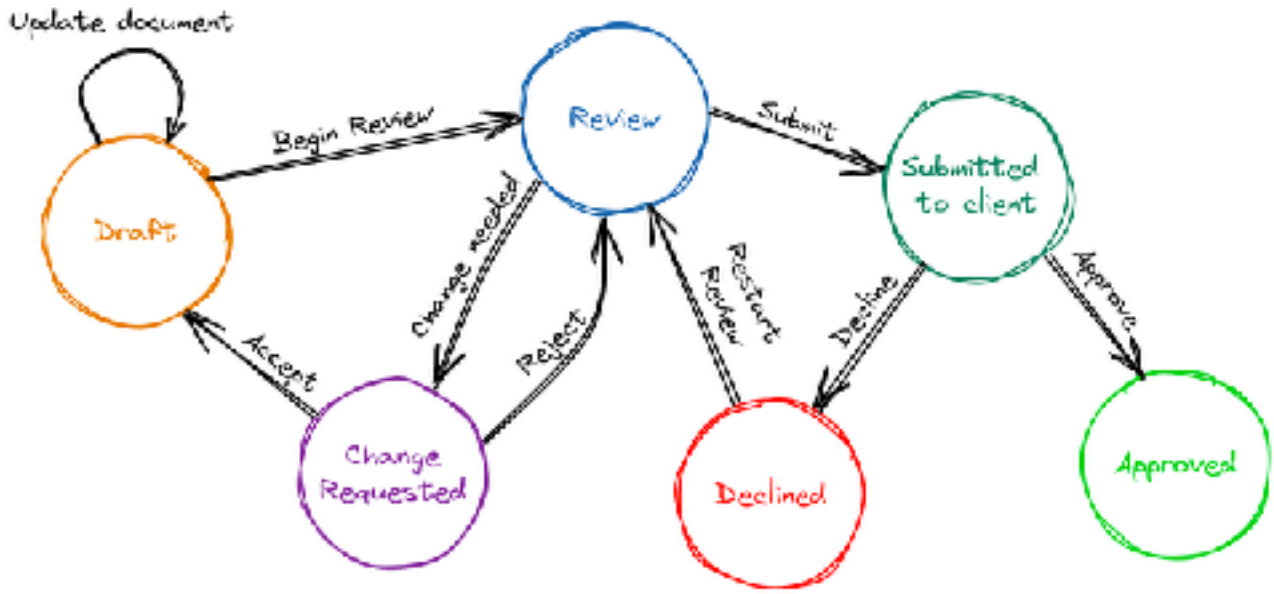
State Machine

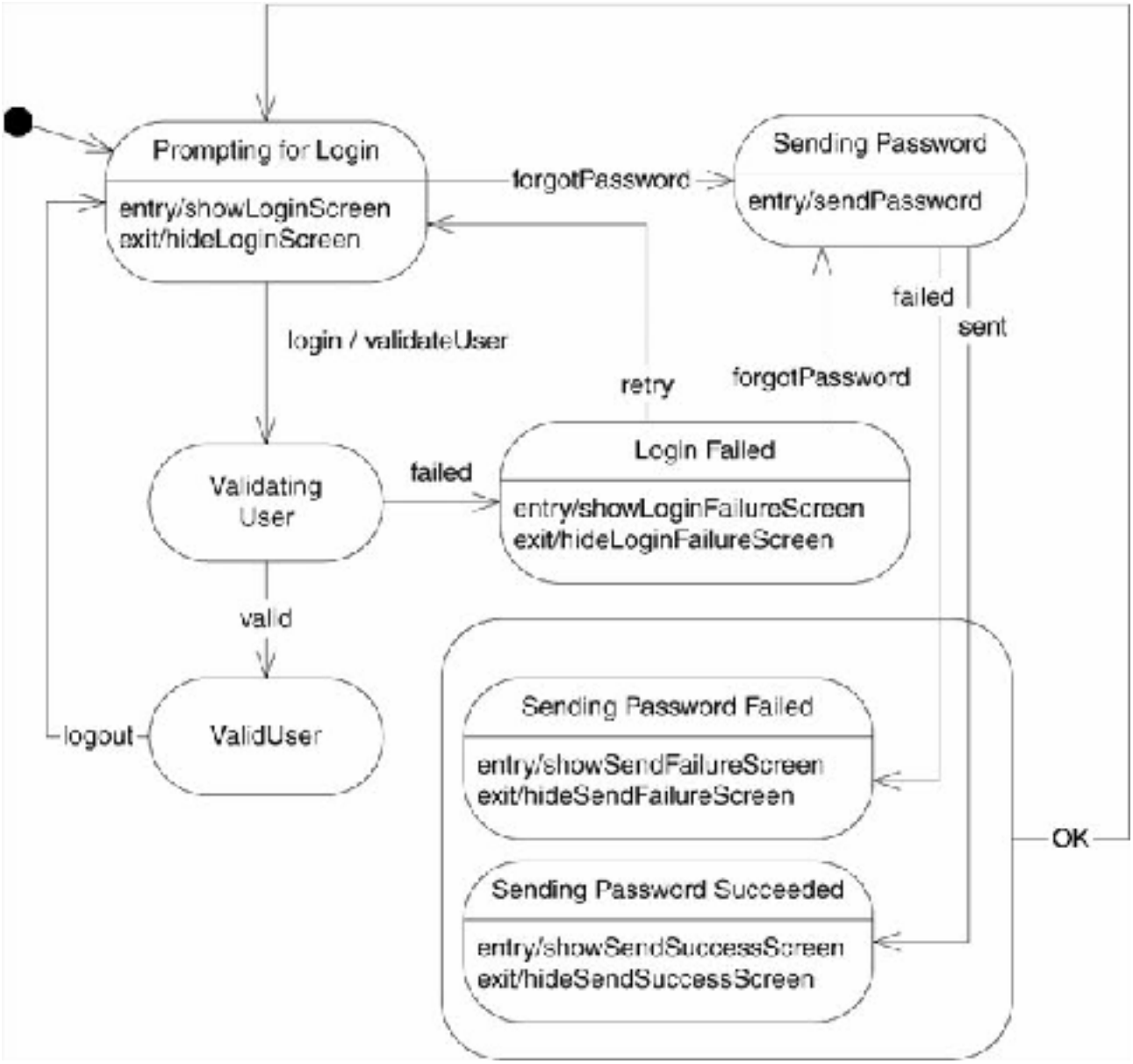
Case Study

Imagine a document which needs to be “approved” before it can be sent to a customer - to check for anything from grammar and spelling mistakes to agreed services and prices. Then, imagine that once it’s been approved it’s then sent to the customer. The customer has to approve, reject, or request changes. Whichever they choose, the document is then sent back to the approver for further review. This goes on until the document is complete and every stakeholder has approved it.



The problem

1. Before it’s either approved or sent to the customer it’s obviously in some kind of state. Call it “draft”.
2. It has to be approved or changes requested by the internal stakeholders.
3. If changes have been requested, then that branches into its own set of states.
4. The customer approves the document.
5. The customer rejects the document.



6. The legal team approves the document.
7. *and so on…*

The more states, the more complex the workflow. If we try model this workflow in code, we'll soon end up with dozens if not hundreds of flags to represent the "current" state.

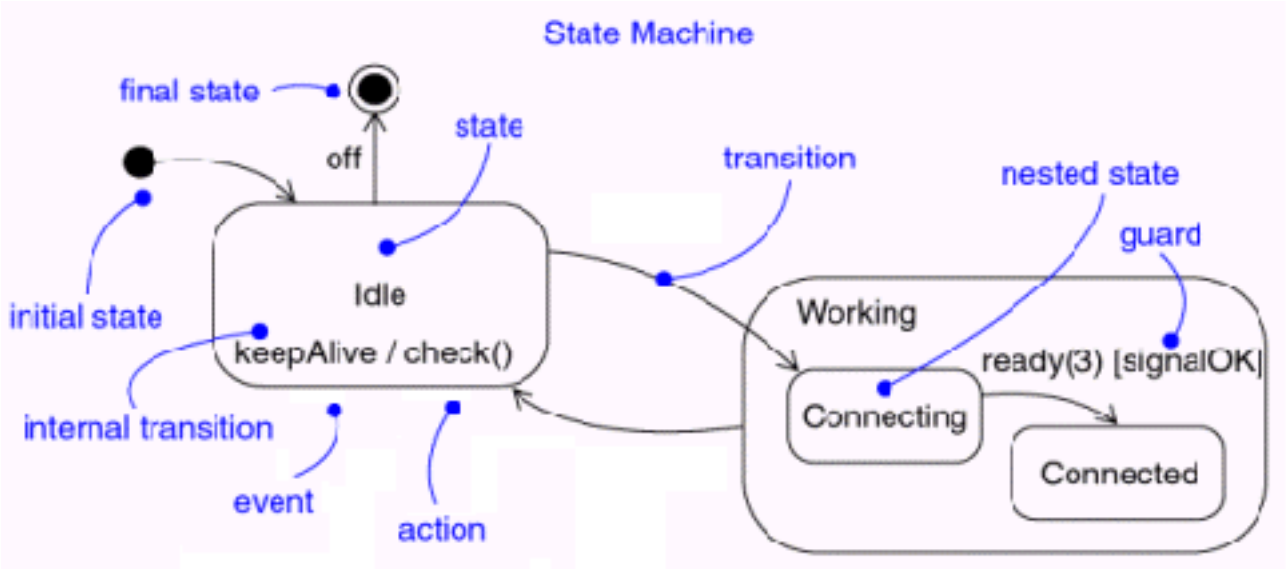
```
if (document.IsInDraft)
{
    if (!document.HasBeenReviewedByLegal)
    {
        ...
    }
    else if (document.IsSentToCustomer && document.Approved ||
document.Rejected)
    {
```

```
        ...
    }
    else
    {
        ...
    }
}
...
```

This type of code can often become *that* part of the codebase that spans hundreds or thousands of lines long and probably all in a single file too. *That* part of a codebase is often the part developers do not want to look at or work on. Developers might even try circumventing it by writing new logic in another part of the codebase thus compounding the issue even further.

A solution

State machines have been around in computer science for a long time. You'll find they are especially popular in reactive systems, like the software for video games and robotics. Designers use state machines to model a system using states, events, and transitions.



State
The basic unit that composes a state machine. A state machine can be in one state at any particular time.

Entry Action
An activity executed when entering the state

Exit Action
An activity executed when exiting the state

Transition
A directed relationship between two states that represents the complete response of a state machine to an occurrence of an event of a particular type.

Trigger
A triggering activity that causes a transition to occur.

Condition
A constraint that must evaluate to true after the trigger occurs in order for the transition to complete.

Transition Action
An activity that is executed when performing a certain transition.

Initial State
A state that represents the starting point of the state machine. A state machine workflow must have one and only one initial state.

Final State
A state that represents the completion of the state machine. A state machine workflow must have at least one final state.

A state can have an Entry and an Exit action. (A state configured as a final state may have only an entry action). When a workflow instance enters a state, any activities in the entry action execute. When the entry action is complete, the triggers for the state's transitions are scheduled. When a transition to another state is

confirmed, the activities in the exit action are executed, even if the state transitions back to the same state. After the exit action completes, the activities in the transition's action execute, and then the new state is transitioned to, and its entry actions are executed. All states must have at least one transition, except for a final state, which may not have any transitions.

When a state machine is idle, It is common to unload the state machine that have gone idle and to reload them to continue execution when a event is triggered. StateMachine run in an environment that provides facilities for exception handling, fpersistence of state data, loading and unloading of in-progress state machine from memory, tracking, and transaction flow.