C++ now

Modernizing Finite State Machines Implementation

Empowering Simplicity and Boosting Performance with std::variant

Amandeep Chawla

2024

Modernizing Finite State Machines Implementation

Amandeep Chawla | Sr. Computer Scientist II

C++Now 2024

Hi, I am Aman (अमन)!

- Sr. Computer Scientist II @ Adobe, San Francisco
  - Working with Adobe for last 18 years
  - Majorly working on C++ code base
  - Cross Platform code - Windows, Mac, iOS and Android.
  - Also works on JavaScript components both BackEnd (Node) and FrontEnd (React)

- Has worked on embedded systems at start of my career.
  - Navigation and weapon integration on Fighter aircrafts.

- Learned C++ during my college days (way back in 1998) and in love with the language since.

- Email: adchawla@gmail.com / amandeep@adobe.com

Ground Rules

- Feel free to interrupt and ask questions
- Code on the slide is in slide-aware format as is more to get the idea across.
- In normal code I utilize and encourage others to use following keywords, where-ever applicable:
  - [[nodiscard]]
  - noexcept
  - const
  - override
  - Final

Agenda

- What are Finite State Machines
- Discuss an example of Finite State Machine
- Evolution of implementation techniques w.r.t Finite State Machines
  - Naïve (using Enum & Switch)
  - State Pattern
  - STL (Variants)
- Benchmarks

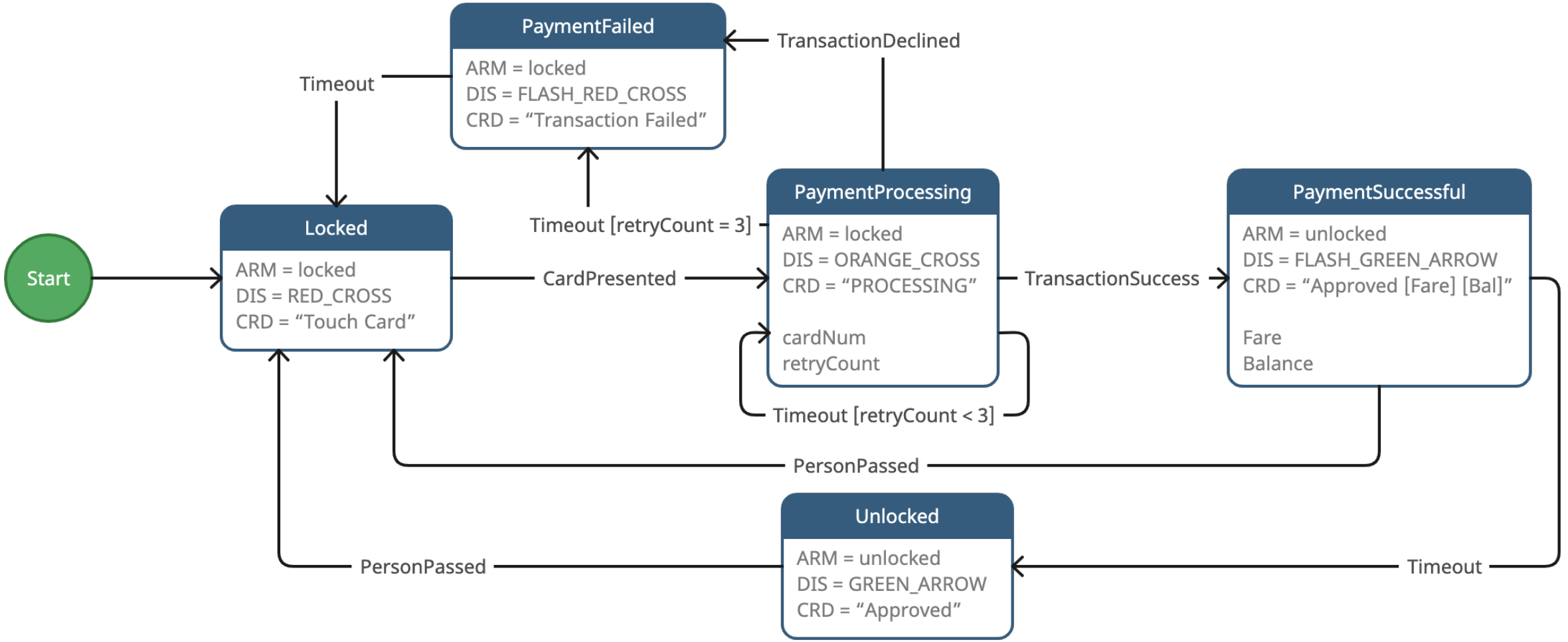
Finite State Machine

is an abstract machine that can be in exactly one of a finite number of *states* at any given time.

It can change from one state to another in response to some inputs; the change from one state to another is called a *transition*.

It is defined by a list of its states, its initial state, and the inputs that trigger each transition

Subway Turnstile | A Simple Finite State Machine



Ground Work
Common To All Implementations

Subway Turnstile | External Devices | Swing Door & LED Display

Subway Turnstile | External Devices | Swing Door & LED Display

```
class SwingDoor {
public:
    enum class eStatus { Closed, Open };
    void open() { _status = eStatus::Open; }
    void close() { _status = eStatus::Closed; }
    eStatus getStatus() const { return _status; }
private:
    eStatus _status{eStatus::Closed};
};

class LEDController {
public:
    enum class eStatus { RedCross, FlashRedCross, GreenArrow, OrangeCross };
    void setStatus(eStatus status) { _status = status;
    eStatus getStatus() const { return _status; }
private:
    eStatus _status{eStatus::RedCross};
};
```

Subway Turnstile | External Devices | POS Terminal

Subway Turnstile | External Devices | POS Terminal

```
class POSTerminal {
public:
    explicit POSTerminal(std::string firstRow, std::string secondRow = "", std::string thirdRow = "")
        : _firstRow(std::move(firstRow)), _secondRow(std::move(secondRow)), _thirdRow(std::move(thirdRow)) { }
    void setRows(std::string firstRow, std::string secondRow = "", std::string thirdRow = "") {
        _firstRow = std::move(firstRow);
        _secondRow = std::move(secondRow);
        _thirdRow = std::move(thirdRow);
    }
    const std::string & getFirstRow() const { return _firstRow; }
    const std::string & getSecondRow() const { return _secondRow; }
    const std::string & getThirdRow() const { return _thirdRow; }
private:
    std::string _firstRow;
    std::string _secondRow;
    std::string _thirdRow;
};
```

Subway Turnstile | Events

```
struct CardPresented {
    std::string cardNumber;
};

struct TransactionDeclined {
    std::string reason;
};

struct TransactionSuccess {
    int fare;
    int balance;
};

struct PersonPassed {};

struct Timeout {};
```

Ways to Implement

- Using (if/else)
- Using Enums
- Inheritance/State Pattern
- std::variant (C++17)
  - Possible with C++11 onwards [Variant's implementations - MPark]
- Boost (C++14)
- Coroutines (C++20)

Finite State Machine Using Enums

FSM | Using Enums | States

```
enum class eState {
    Locked,
    PaymentProcessing,
    PaymentFailed,
    PaymentSuccess,
    Unlocked
};
```

FSM | Using Enums | Implementation

```
class FSM {
public:
    FSM & process(CardPresented event);
    FSM & process(TransactionDeclined event);
    FSM & process(TransactionSuccess event);
    FSM & process(PersonPassed event);
    FSM & process(Timeout event);
    eState getState() const { return _state; }

private:
    // External Actions
    void initiateTransaction(const std::string & gateway, const std::string & cardNumber, int amount);

    eState _state{eState::Locked};
    // Connected Devices
    SwingDoor _door;
    POSTerminal _pos{"Touch Card"};
    LEDController _led;
    // All States Data
    int _retryCounts{0};
    std::string _cardNumber{};
    int _fare;
    int _balance;
};
```

FSM | Using Enums | Transitions | Unlocked -> PaymentProcessing

```
FSM & FSM::process(CardPresented event) {
    switch (_state) {
    case eState::Locked:
        transitionToPaymentProcessing(_gateways[0], std::move(event.cardNumber));
        break;
    default:
        break;
    }
    return *this;
}

void FSM::transitionToPaymentProcessing(const std::string & gateway, std::string cardNumber) {
    _cardNumber = std::move(cardNumber);
    initiateTransaction(gateway, _cardNumber, getFare());
    _door.close();
    _pos.setRows("Processing");
    _led.setStatus(LEDController::eStatus::OrangeCross);
    _state = eState::PaymentProcessing;
}
```

FSM | Using Enums | Transitions | Cont...

```
FSM & FSM::process(TransactionDeclined event) {
    switch (_state) {
    case eState::PaymentProcessing: transitionToPaymentFailed(std::move(event.reason)); break;
    default: break;
    }
    return *this;
}

FSM & FSM::process(TransactionSuccess event) {
    switch (_state) {
    case eState::PaymentProcessing: transitionToPaymentSuccessful(event.fare, event.balance); break;
    default: break;
    }
    return *this;
}

FSM & FSM::process(PersonPassed event) {
    switch (_state) {
    case eState::PaymentSuccess:
    case eState::Unlocked: transitionToLocked(); break;
    default: break;
    }
    return *this;
}
```

FSM | Using Enums | Transitions | Cont...

```
void FSM::transitionToPaymentFailed(const std::string & reason) {
    _door.close();
    _pos.setRows("Declined", reason);
    _led.setStatus(LEDController::eStatus::FlashRedCross);
    _state = eState::PaymentFailed;
}
void FSM::transitionToLocked() {
    _state = eState::Locked;
    _door.close();
    _pos.setRows("Touch Card");
    _led.setStatus(LEDController::eStatus::RedCross);
}
void FSM::transitionToPaymentSuccessful(int fare, int balance) {
    _state = eState::PaymentSuccess;
    _door.open();
    _pos.setRows("Approved", string("Fare: ") + to_string(fare), string("Balance: ") + to_string(balance));
    _led.setStatus(LEDController::eStatus::GreenArrow);
}
void FSM::transitionToUnlocked() {
    _state = eState::Unlocked;
    _door.open();
    _pos.setRows("Approved");
    _led.setStatus(LEDController::eStatus::GreenArrow);
}
```

FSM | Using Enums | Transitions | Timeout Handling

```
FSM & FSM::process(Timeout event) {
    switch (_state) {
    case eState::PaymentProcessing:
        if (++_retryCounts > 2) {
            transitionToPaymentFailed("Network Error");
        } else {
            initiateTransaction(_gateways[_retryCounts], _cardNumber, getFare());
        }
        break;
    case eState::PaymentFailed: transitionToLocked(); break;
    case eState::PaymentSuccess: transitionToUnlocked(); break;
    default: break;
    }
    return *this;
}
```

FSM | Using Enums | Transitions | Unlocked -> PaymentProcessing

```
TEST(FSMWithEnums, TestPaymentProcessing) {
    FSMWithEnums fsm;
    fsm.process(CardPresented{"A"});

    EXPECT_EQ(fsm.getState(), eState::PaymentProcessing);
    EXPECT_EQ(SwingDoor::eStatus::Closed, fsm.getDoor().getStatus());
    EXPECT_EQ(LEDController::eStatus::OrangeCross, fsm.getLEDController().getStatus());
    EXPECT_EQ("Processing", fsm.getPOS().getFirstRow());
    EXPECT_EQ("", fsm.getPOS().getSecondRow());
    EXPECT_EQ("", fsm.getPOS().getThirdRow());
    EXPECT_EQ(fsm.getLastTransaction(), std::make_tuple("Gateway1", "A", getFare()));
}
```

// Console Output

```
STATE: Locked :: Door[Closed], LED: [RedCross] and PosTerminal[Touch Card, , ]
EVENT: CardPresent
ACTIONS: Initiated Transaction to [Gateway1] with card [A] for amount [3]
STATE: PaymentProcessing :: Door[Closed], LED: [OrangeCross] and PosTerminal[Processing, , ]
```

FSM | Using Enums | Quiz Time

```
FSMWithEnums fsm;
fsm
    .process(CardPresented{"A"})
    .process(Timeout{})
    .process(Timeout{})
    .process(Timeout{})
    .process(Timeout{})

// Q1. What is the state now

    .process(CardPresented{"A"})
    .process(Timeout{});
// Q2. What is the state now
```

FSM | Using Enums | Testing 1st Attempt

```
[----------] 10 tests from FSMWithEnums
[ RUN      ] FSMWithEnums.TestInitialState
[       OK ] FSMWithEnums.TestInitialState (0 ms)
[ RUN      ] FSMWithEnums.TestPaymentProcessing
[       OK ] FSMWithEnums.TestPaymentProcessing (0 ms)
[ RUN      ] FSMWithEnums.TestPaymentFailed
[       OK ] FSMWithEnums.TestPaymentFailed (0 ms)
[ RUN      ] FSMWithEnums.TestTimeoutOnPaymentProcessing
[       OK ] FSMWithEnums.TestTimeoutOnPaymentProcessing (0 ms)
[ RUN      ] FSMWithEnums.TestLockedFromPaymentFailed
[       OK ] FSMWithEnums.TestLockedFromPaymentFailed (0 ms)
[ RUN      ] FSMWithEnums.TestPaymentSuccessful
[       OK ] FSMWithEnums.TestPaymentSuccessful (0 ms)
[ RUN      ] FSMWithEnums.TestUnlocked
[       OK ] FSMWithEnums.TestUnlocked (0 ms)
[ RUN      ] FSMWithEnums.TestLockedFromUnlocked
[       OK ] FSMWithEnums.TestLockedFromUnlocked (0 ms)
[ RUN      ] FSMWithEnums.TestLockedFromPaymentSuccessful
[       OK ] FSMWithEnums.TestLockedFromPaymentSuccessful (0 ms)
[ RUN      ] FSMWithEnums.TestBug
/Users/amandeep/Code/GitHub/adchawla/FSM/tests/testFSMWithEnums.cpp:164: Failure
Expected equality of these values:
  eState::PaymentProcessing
    Which is: 4-byte object <01-00 00-00>
  fsm.getState()
    Which is: 4-byte object <02-00 00-00>

[  FAILED  ] FSMWithEnums.TestBug (0 ms)
```

FSM | Using Enums | Bug

```
void FSMWithEnums::transitionToPaymentProcessing(const std::string & gateway, std::string cardNumber) {
    _cardNumber = std::move(cardNumber);
    initiateTransaction(gateway, _cardNumber, getFare());
    _door.close();
    _pos.setRows("Processing");
    _led.setStatus(LEDController::eStatus::OrangeCross);
    _state = eState::PaymentProcessing;
    _retryCounts = 0;
}
```

FSM | Using Enums | Testing 2nd Attempt

```
[----------] 10 tests from FSMWithEnums
[ RUN      ] FSMWithEnums.TestInitialState
[       OK ] FSMWithEnums.TestInitialState (0 ms)
[ RUN      ] FSMWithEnums.TestPaymentProcessing
[       OK ] FSMWithEnums.TestPaymentProcessing (0 ms)
[ RUN      ] FSMWithEnums.TestPaymentFailed
[       OK ] FSMWithEnums.TestPaymentFailed (0 ms)
[ RUN      ] FSMWithEnums.TestTimeoutOnPaymentProcessing
[       OK ] FSMWithEnums.TestTimeoutOnPaymentProcessing (0 ms)
[ RUN      ] FSMWithEnums.TestLockedFromPaymentFailed
[       OK ] FSMWithEnums.TestLockedFromPaymentFailed (0 ms)
[ RUN      ] FSMWithEnums.TestPaymentSuccessful
[       OK ] FSMWithEnums.TestPaymentSuccessful (0 ms)
[ RUN      ] FSMWithEnums.TestUnlocked
[       OK ] FSMWithEnums.TestUnlocked (0 ms)
[ RUN      ] FSMWithEnums.TestLockedFromUnlocked
[       OK ] FSMWithEnums.TestLockedFromUnlocked (0 ms)
[ RUN      ] FSMWithEnums.TestLockedFromPaymentSuccessful
[       OK ] FSMWithEnums.TestLockedFromPaymentSuccessful (0 ms)
[ RUN      ] FSMWithEnums.TestBug
[       OK ] FSMWithEnums.TestBug (0 ms)
[----------] 10 tests from FSMWithEnums (0 ms total)
```

FSM | Using Enums | Typical Bugs

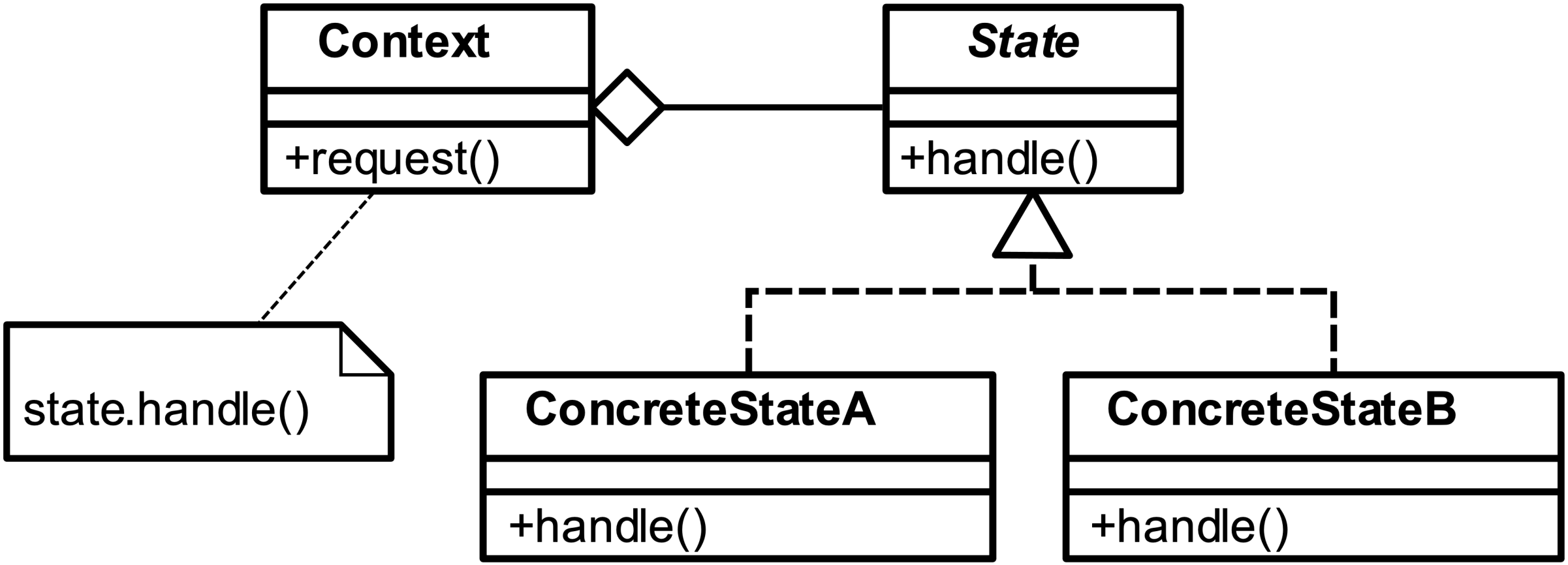
- Failure to reset variables belonging to a state on transition
- Canceling/Resetting external actions (timeout in our case).

FSM | Using Enums | Summary

- Advantages
  - No Heap Usage
  - Simplicity for simpler state machines
- Disadvantages
  - States are simple (enum, don't have any data) so State Machine must be an algebraic **Product Type**
    - variables stay in scope more than they are required
    - Increases memory footprint
  - Hard to add new events and states
  - Shared responsibility

Finite State Machine Using State Pattern

FSM | Using State Pattern | State Pattern



FSM | Using State Pattern | Abstract Base State

```
class BaseState {
public:
    explicit BaseState(std::reference_wrapper<FSM> context) : _context(context) { }
    virtual ~BaseState() = default;

    virtual eState getState() = 0;

    // event handlers
    virtual std::unique_ptr<BaseState> process(CardPresented event) { return nullptr; }
    virtual std::unique_ptr<BaseState> process(TransactionDeclined event) { return nullptr; }
    virtual std::unique_ptr<BaseState> process(TransactionSuccess event) { return nullptr; }
    virtual std::unique_ptr<BaseState> process(PersonPassed event) { return nullptr; }
    virtual std::unique_ptr<BaseState> process(Timeout event) { return nullptr; }

protected:
    std::reference_wrapper<FSM> _context;
};
```

FSM | Using State Pattern | States | Locked

```
class Locked final : public BaseState {
public:
    explicit Locked(std::reference_wrapper<FSM> context) : BaseState(context) {
        auto & fsm = _context.get();
        fsm.getDoor().close();
        fsm.getLED().setStatus(LEDController::eStatus::RedCross);
        fsm.getPOS().setRows("Touch Card");
    }

    eState state() override { return eState::Locked; }
    std::unique_ptr<BaseState> process(CardPresented event) override {
        return std::make_unique<PaymentProcessing>(_context, std::move(event.cardNumber));
    }
};
```

FSM | Using State Pattern | States | PaymentProcessing

```
class PaymentProcessing final : public BaseState {
public:
    explicit PaymentProcessing(std::reference_wrapper<FSM> context, std::string cardNumber);

    eState state() override { return eState::PaymentProcessing; }

    std::unique_ptr<BaseState> process(TransactionDeclined event) override;
    std::unique_ptr<BaseState> process(TransactionSuccess event) override;
    std::unique_ptr<BaseState> process(Timeout event) override;

private:
    size_t _retryCount{0};
    std::string _cardNumber;
    TimeoutManager _timeoutManager;
};
```

FSM | Using State Pattern | States | PaymentProcessingState...

```
PaymentProcessing::PaymentProcessing (
    std::reference_wrapper<FSM> context, std::string cardNumber)
    : BaseState(context), _cardNumber(std::move(cardNumber)), _timeoutManager([&] {process(Timeout{});}, 2s) {
    auto & fsm = _context.get();
    fsm.getDoor().close();
    fsm.getLED().setStatus(LEDController::eStatus::OrangeCross);
    fsm.getPOS().setRows("Processing");
    fsm.initiateTransaction(GATEWAYS[_retryCount], _cardNumber, getFare());
}
std::unique_ptr<BaseState> PaymentProcessing::process(TransactionDeclined event) {
    return std::make_unique<PaymentFailed>(_context, std::move(event.reason));
}
std::unique_ptr<BaseState> PaymentProcessing::process(TransactionSuccess event) {
    return std::make_unique<PaymentSuccess>(_context, event.fare, event.balance);
}
std::unique_ptr<BaseState> PaymentProcessing::process(Timeout event) {
    if (++_retryCount >= GATEWAYS.size()) {return std::make_unique<PaymentFailed>(_context, "Network Failure");}
    _context.get().initiateTransaction(GATEWAYS[_retryCount], _cardNumber, getFare());
    _timeoutManager.restart(2s);
    return nullptr;
}
```

FSM | Using State Pattern | States | PaymentFailed

```
class PaymentFailed final : public BaseState {
public:
    PaymentFailed(std::reference_wrapper<FSM> context, std::string reason)
        : BaseState(context), _reason(std::move(reason)), _timeoutManager([&] {process(Timeout{});}, 2s) {
        auto & fsm = _context.get();
        fsm.getDoor().close();
        fsm.getLED().setStatus(LEDController::eStatus::FlashRedCross);
        fsm.getPOS().setRows("Declined", _reason);
    }

    eState state() override { return eState::PaymentFailed; }
    std::unique_ptr<BaseState> process(Timeout event) override {
        return std::make_unique<Locked>(_context);
    }

private:
    std::string _reason;
    TimeoutManager _timeoutManager;
};
```

FSM | Using State Pattern | States | PaymentSuccess

```
class PaymentSuccess final : public BaseState {
public:
    PaymentSuccess(std::reference_wrapper<FSM> context, int fare, int balance)
        : BaseState(context), _timeoutManager([&] {process(Timeout{});}, 2s) {
        auto & fsm = _context.get();
        fsm.getDoor().open();
        fsm.getLED().setStatus(LEDController::eStatus::GreenArrow);
        fsm.getPOS().setRows(
           "Approved", string("Fare: ") + to_string(fare), string("Balance: ") + to_string(balance));
    }

    eState state() override { return eState::PaymentSuccess; }
    std::unique_ptr<BaseState> process(PersonPassed event) override {
        return std::make_unique<Locked>(_context);
    }
    std::unique_ptr<BaseState> process(Timeout event) override {
        return std::make_unique<Unlocked>(_context);
    }

private:
    TimeoutManager _timeoutManager;
};
```

FSM | Using State Pattern | States | Unlocked

```
class Unlocked final : public BaseState {
public:
    explicit Unlocked(std::reference_wrapper<FSM> context)
        : BaseState(context) {
        auto & fsm = _context.get();
        fsm.getDoor().open();
        fsm.getLED().setStatus(LEDController::eStatus::GreenArrow);
        fsm.getPOS().setRows("Approved");
    }

    eState state() override { return eState::Unlocked; }
    std::unique_ptr<BaseState> process(PersonPassed event) override {
        return std::make_unique<Locked>(_context);
    }
};
```

FSM | Using State Pattern | Concrete State Machine (Context)

```
class FSM {
public:
    FSM () _state(std::make_unique<LockedState>(std::ref(*this))) {};

    template <typename Event>
    FSM & process(Event && event) {
        if (auto newState = _state->process(std::forward<Event>(event))) { _state = std::move(newState); }
        return *this;
    }

    eState getState() const;
    SwingDoor & getDoor() { return _door; }
    POSTerminal & getPOS() { return _pos; }
    LEDController & getLED() { return _led; }

    // External Actions
    void initiateTransaction(const std::string & gateway, const std::string & cardNum, int amount);
private:
    // Connected Devices
    SwingDoor _door;
    POSTerminal _pos{""};
    LEDController _led;
    std::unique_ptr<BaseState> _state;
};
```

FSM | Using State Pattern | Summary

- Advantages
  - Cleaner separation because of Algebaric Sum Type
  - Easy to add new states
  - Utilizes RAII
  - Easy to Unit Test
- Disadvantages
  - Heap Allocation
  - Virtual method calls
  - Some work required to add new events.

Finite State Machine Using std::variant

FSM | Using std::variant | Common Stuff

```
using States = std::variant<
    Locked,
    PaymentProcessing,
    PaymentFailed,
    PaymentSuccess,
    Unlocked>;

using OptState = std::optional<States>;
```

FSM | Using std::variant | States | Base State

```
class BaseState {
public:
    explicit BaseState(std::reference_wrapper<FSM> context) : _context(context) { }

    template <typename EventType> OptState process(EventType);

protected:
    std::reference_wrapper<FSM> _context;
};

// Must be defined after all the state definitions
template <typename EventType>
OptState BaseState::process(EventType) {
    return OptState{};
}
```

FSM | Using std::variant | States | Locked

```
class Locked : public BaseState {
public:
    explicit Locked (std::reference_wrapper<FSM> context) : BaseState(context) {
        auto & fsm = _context.get();
        fsm.getDoor().close();
        fsm.getLED().setStatus(LEDController::eStatus::RedCross);
        fsm.getPOS().setRows("Touch Card");
    }

    using BaseState::process;
    OptState process(CardPresented event) {
        return PaymentProcessing(_context, std::move(event.cardNumber));
    }
};
```

FSM | Using std::variant | States | PaymentProcessing

```
class PaymentProcessing : public BaseState {
public:
    explicit PaymentProcessing(std::reference_wrapper<FSM> context, std::string cardNumber);

    using BaseState::process;
    OptState process(TransactionDeclined event);
    OptState process(TransactionSuccess event);
    OptState process(Timeout event);

private:
    size_t _retryCount{0};
    std::string _cardNumber;
    TimeoutManager _timeoutManager;
};
```

FSM | Using std::variant | States | PaymentProcessing...

```
PaymentProcessing::PaymentProcessing(
    std::reference_wrapper<FSM> context, std::string cardNumber)
    : BaseState(context), _cardNumber(std::move(cardNumber)), _timeoutManager([&] {process(Timeout{});}, 2s) {
    auto & fsm = _context.get();
    fsm.getDoor().close();
    fsm.getLED().setStatus(LEDController::eStatus::OrangeCross);
    fsm.getPOS().setRows("Processing");
    fsm.initiateTransaction(_gateways[_retryCount], _cardNumber, getFare());
}
OptState PaymentProcessing::process(TransactionDeclined event) {
    return PaymentFailed(_context, std::move(event.reason));
}
OptState PaymentProcessing::process(TransactionSuccess event) {
    return PaymentSuccess(_context, event.fare, event.balance);
}
OptState PaymentProcessing::process(Timeout event) {
    if (++_retryCount >= _gateways.size()) { return PaymentFailed(_context, "Network Failure"); }
    _context.get().initiateTransaction(_gateways[_retryCount], _cardNumber, getFare());
    _timeoutManager.restart(2s);
    return OptState{};
}
```

FSM | Using std::variant | States | PaymentFailed

```
class PaymentFailed : public BaseState {
public:
    PaymentFailed(std::reference_wrapper<FSM> context, std::string reason)
        : BaseState(context), _reason(std::move(reason)), _timeoutManager([&] {process(Timeout{});}, 2s) {
        auto & fsm = _context.get();
        fsm.getDoor().close();
        fsm.getLED().setStatus(LEDController::eStatus::FlashRedCross);
        fsm.getPOS().setRows("Declined", _reason);
    }

    using BaseState::process;
    OptState process(Timeout event) {
        return Locked(_context);
    }

private:
    std::string _reason;
    TimeoutManager _timeoutManager;
};
```

FSM | Using std::variant | States | PaymentSuccess

```
class PaymentSuccess final : public BaseState {
public:
    PaymentSuccess(std::reference_wrapper<FSM> context, int fare, int balance)
        : BaseState(context), _timeoutManager([&] {process(Timeout{});}, 2s) {
        auto & fsm = _context.get();
        fsm.getDoor().open();
        fsm.getLED().setStatus(LEDController::eStatus::GreenArrow);
        fsm.getPOS().setRows(
           "Approved", string("Fare: ") + to_string(fare), string("Balance: ") + to_string(balance));
    }

    OptState process(PersonPassed event) override {
        return Locked (_context);
    }

    OptState process(Timeout event) override {
        return Unlocked(_context);
    }

private:
    TimeoutManager _timeoutManager;
};
```

FSM | Using std::variant | States | Unlocked

```
class Unlocked final : public BaseState {
public:
    explicit Unlocked(std::reference_wrapper<FSM> context)
        : BaseState(context) {
        auto & fsm = _context.get();
        fsm.getDoor().open();
        fsm.getLED().setStatus(LEDController::eStatus::GreenArrow);
        fsm.getPOS().setRows("Approved");
    }

    OptState process(PersonPassed event) override {
        return Locked(_context);
    }
};
```

FSM | Using std::variant | Generic State Machine class

```
template <typename... States>
class TFSM {
public:
    template <typename InitialState>
    explicit TFSM(InitialState && state) : _state{std::forward<InitialState>(state)} {
    }

    template <typename Event>
    void process(Event event) {
        auto optResult = std::visit([&](auto & state) { return state.process(std::move(event)); }, _state);
        if (optResult) {
            _state = std::move(optResult.value());
        }
    }

private:
    std::variant<States...> _state;
};
```

FSM | Using std::variant | Actual FSM class

```
class FSM {
public:
    FSM() : _fsm{Locked{std::ref(*this)}} { }

    template <typename Event> FSM & process(Event event) {
        _fsm.process(std::move(event));
        return *this;
    }

    eState getState() const;
    SwingDoor & getDoor() { return _door; }
    POSTerminal & getPOS() { return _pos; }
    LEDController & getLED() { return _led; }

    // External Actions
    void initiateTransaction(const std::string & gateway, const std::string & cardNum, int amount);
private:
    // Connected Devices
    SwingDoor _door;
    POSTerminal _pos{""};
    LEDController _led;
    TFSM<Locked, PaymentProcessing, PaymentFailed, PaymentSuccess, Unlocked> _fsm;
};
```

FSM | Using std::variant | Summary

- Advantages
  - Cleaner separation resulting in Sum Type
  - Easy to add new states
  - RAII
  - Easy to Unit Test
  - No Heap Allocation
  - No Virtual Functions
  - Trivial to add new events
- Disadvantages
  - Order of definition
  - Tough to figure out compile time errors

But are we done yet

FSM | Using std::variant | External Transitions

- States are no longer responsible for coding the logic of transitions.
- We can code external transitions in a separate object.

FSM | Using std::variant | External Transitions | Transition Table

```
struct TransitionTable {
    OptState operator()(LockedState & state, CardPresented event) {
        return PaymentProcessingState(state._context, std::move(event.cardNumber));
    }
    OptState operator()(PaymentProcessingState & state, TransactionDeclined event) {
        return PaymentFailed(state._context, std::move(event.reason));
    }
    OptState operator()(PaymentProcessingState & state, TransactionSuccess event) {
        return PaymentSuccess(state._context, event.fare, event.balance);
    }
    OptState operator()(PaymentProcessingState & state, Timeout event) {
        return state.tryRetry() ? OptState{} : PaymentFailed(state._context, "Network Failure");
    }

    OptState operator()(PaymentFailed & state, Timeout) { return LockedState(state._context); }
    OptState operator()(PaymentSuccess & state, Timeout) { return Unlocked(state._context);  }
    OptState operator()(PaymentSuccess & state, PersonPassed) { return LockedState(state._context); }
    OptState operator()(Unlocked & state, PersonPassed) { return LockedState(state._context); }

    template <typename State, typename Event>
    auto operator()(State & s, Event e) const { return OptState{}; }
};
```

FSM | Using std::variant | States | PaymentProcessing

```
class PaymentProcessing : public BaseState {
public:
    explicit PaymentProcessing(std::reference_wrapper<FSM> context, std::string cardNumber);

    bool tryRetry() {
        if (++_retryCount >= GATEWAYS.size()) { return false; }
        _context.get().initiateTransaction(GATEWAYS[_retryCount], _cardNumber, getFare());
        _timeoutManager.restart(2s);
        return true;
    }

private:
    size_t _retryCount{0};
    std::string _cardNumber;
    TimeoutManager _timeoutManager;
};
```

FSM | Using std::variant | External Transitions | Generic State Machine

```
template <typename Transitions, typename... States>
class TFSM {
public:
    template <typename InitialState>
    explicit TFSM(InitialState && state) : _state{std::forward<InitialState>(state)} {
    }

    template <typename Event>
    void process(Event event) {
        auto optResult =
            std::visit([&](auto & state) { return _transitions.operator()(state, std::move(event)); }, _state);
        if (optResult) {
            _state = std::move(optResult.value());
        }
    }

private:
    std::variant<States...> _state;
    Transitions _transitions{};
};
```

FSM | Using std::variant | External Transitions | Actual FSM class

```
class FSM {
public:
    FSM() : _fsm{Locked{std::ref(*this)}} { }

    template <typename Event> FSM & process(Event event) {
        _fsm.process(std::move(event));
        return *this;
    }

    eState getState() const;
    SwingDoor & getDoor() { return _door; }
    POSTerminal & getPOS() { return _pos; }
    LEDController & getLED() { return _led; }

    // External Actions
    void initiateTransaction(const std::string & gateway, const std::string & cardNum, int amount);
private:
    // Connected Devices
    SwingDoor _door;
    POSTerminal _pos{""};
    LEDController _led;
    TFSM<TransitionTable, Locked, PaymentProcessing, PaymentFailed, PaymentSuccess, Unlocked> _fsm;
};
```

FSM | Using std::variant | External Transitions | Summary

- Advantages
  - All the ones we had previously
  - States become pure and don't have to code transitions
  - Transitions are coded separately.
- Disadvantages
  - Order of definition
  - Tough to figure out compile time errors

I am loving this...
but what about DRY

FSM | Using std::variant | Problem

```
template <typename... States>
class TFSM {
public:
    template <typename InitialState>
    explicit TFSM(InitialState && state)
        : _state{std::forward<InitialState>(state)} {
    }

    template <typename Event>
    void process(Event event) {
        auto optResult =
            std::visit([&](auto & state) { return state.process(std::move(event)); }, _state);
        if (optResult) {
            _state = std::move(optResult.value());
        }
    }

private:
    std::variant<States...> _state;

};
```

FSM | Using std::variant | Solution | TFSMBase

```
template <typename Stategy, typename... States>
class TFSMBase {
public:
    template <typename InitialState>
    explicit TFSMBase(Strategy strategy, InitialState && state)
        : _state{std::forward<InitialState>(state)}, _strategy{std::move(strategy)} {
    }

    template <typename Event>
    void process(Event event) {
        auto optResult =
            std::visit([&](auto & state) { return _strategy.execute(state, std::move(event)); }, _state);
        if (optResult) {
            _state = std::move(optResult.value());
        }
    }

protected:
    std::variant<States...> _state;
    Strategy _strategy;
};
```

FSM | Using std::variant | Solution | Strategies

```
template <typename Transitions>
struct TExternalTransitions {
    explicit TExternalTransitions(Transitions transitions) : _transitions(std::move(transitions)) {
    }
    template <typename State, typename Event>
    auto execute(State & state, Event && event) {
        return _transitions.operator()(state, std::forward<Event>(event));
    }

private:
    Transitions _transitions{};
};

struct StatesHandlingTransitions {
    template <typename State, typename Event>
    auto execute(State & state, Event && event) {
        return state.process(std::forward<Event>(event));
    }
};
```

FSM | Using std::variant | Solution

```
template <typename... States>
class TFSMStateTransitions : public details::TFSMBase<details::StatesHandlingTransitions, States...> {
public:
    using BaseType = details::TFSMBase<details::StatesHandlingTransitions, States...>;
    using StrategyType = details::StatesHandlingTransitions;

    template <typename InitialState>
    explicit TFSMStateTransitions(InitialState && state)
        : BaseType{StrategyType{}, std::forward<InitialState>(state)} { }
};


template <typename Transitions, typename... States>
class TFSMExternalTransitions : public details::TFSMBase<details::TExternalTransitions<Transitions>, States...> {
public:
    using BaseType = details::TFSMBase<details::TExternalTransitions<Transitions>, States...>;
    using StrategyType = details::TExternalTransitions<Transitions>;

    template <typename InitialState>
    explicit TFSMExternalTransitions(Transitions transitions, InitialState && state)
        : BaseType{StrategyType{std::move(transitions)}, std::forward<InitialState>(state)} { }
};
```

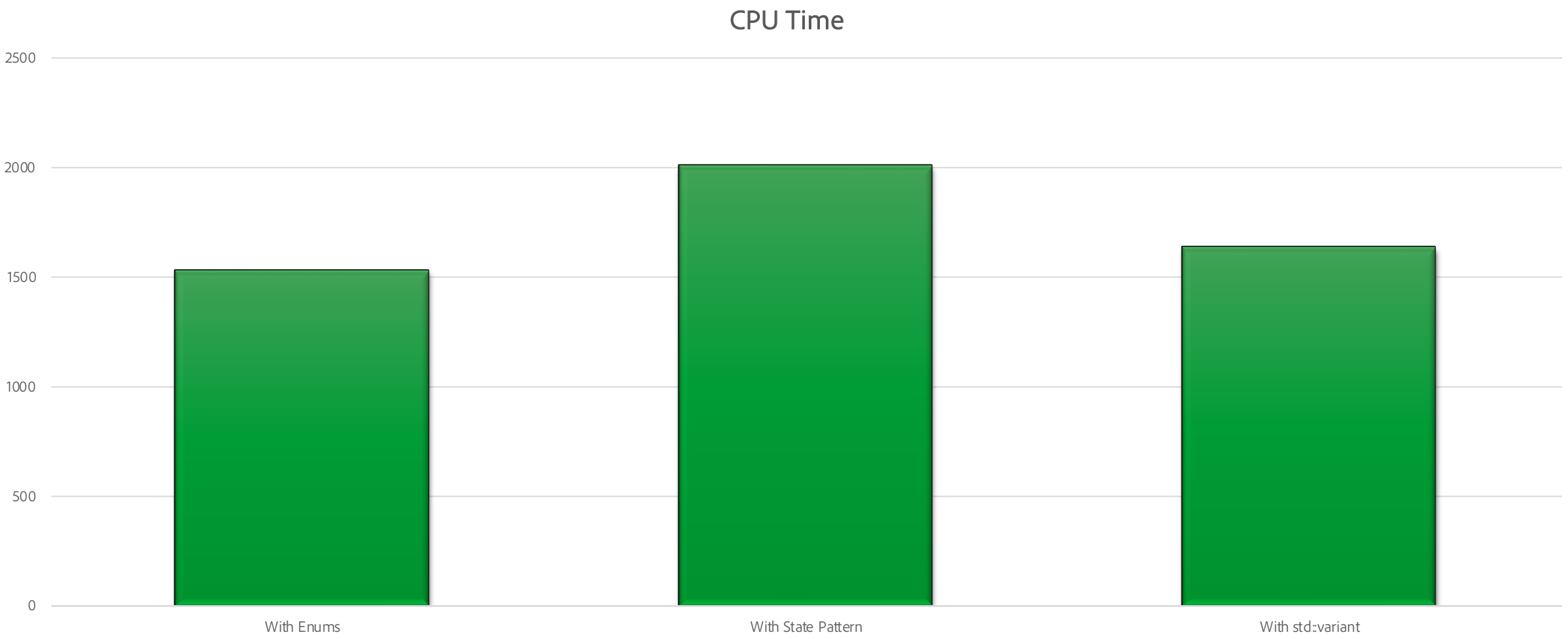
Now it looks good…

But what about performance??

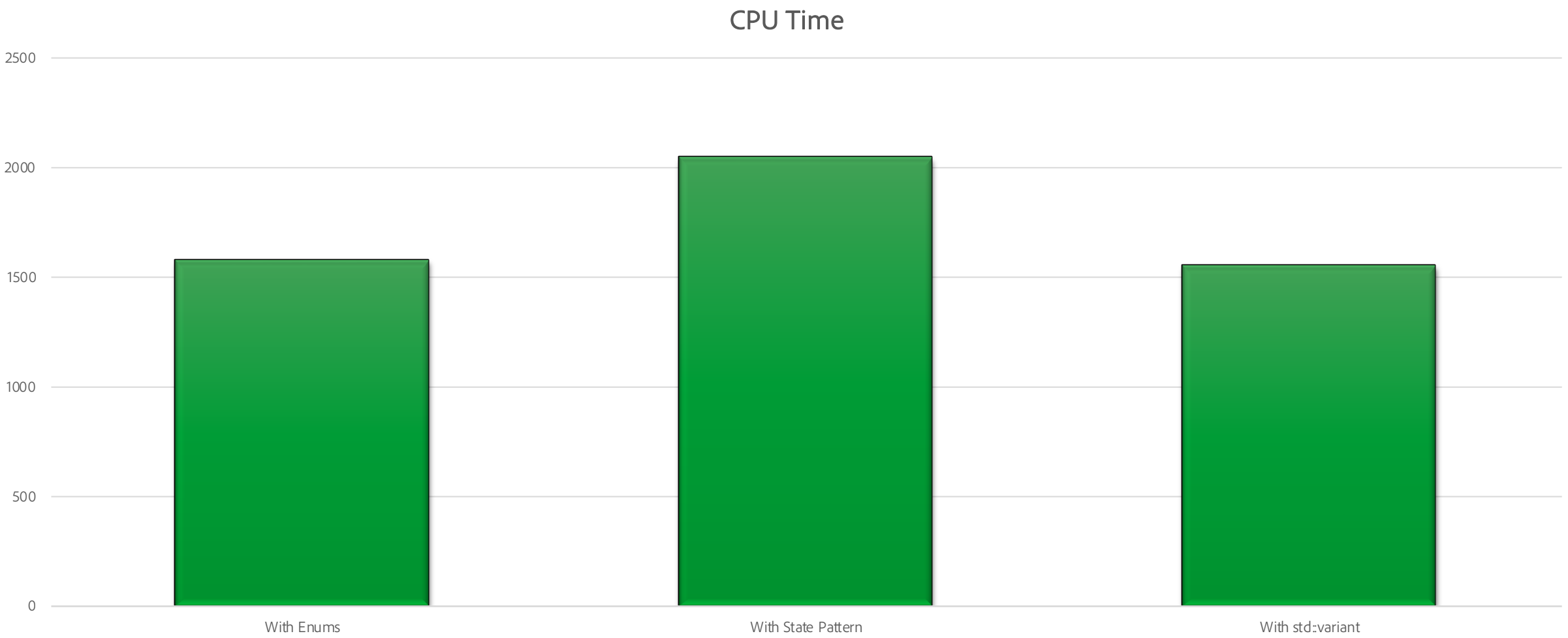
Benchmark Results

```
static void BM_FSM(benchmark::State & state) {
    FSM fsm;
    for (auto _ : state)
        fsm.process(CardPresented{})
            .process(Timeout{})
            .process(Timeout{})
            .process(TransactionSuccess{5, 25})
            .process(Timeout{})
            .process(PersonPassed{});
}
BENCHMARK(BM_FSM);
```

Benchmark Results | Skewed



Benchmark Results



Summary

- we designed and implemented a performant FSM Engine in last 1:5 hour.
  - It provides all the flexibility we need
  - And all the advantages listed in previous slides
- Code/tests/benchmarks are available @ https://github.com/adchawla/FSM


Adobe

Back Up Slides

Benchmarks

```
Running ./benchmarks
Run on (16 X 2400 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB
  L1 Instruction 32 KiB
  L2 Unified 256 KiB (x8)
  L3 Unified 16384 KiB
Load Average: 3.45, 3.61, 3.71
```

Benchmark	Time	CPU	Iterations
BM_FSMStateTransitions	1632 ns	1629 ns	433662
BM_FSMWithEnums	1525 ns	1525 ns	456985
BM_FSMWithStatePattern	2004 ns	2002 ns	359433

Benchmarks

- 2024-05-02T11:08:10-06:00
- Running ./benchmarks
- Run on (16 X 2400 MHz CPU s)
- CPU Caches:
- L1 Data 32 KiB
- L1 Instruction 32 KiB
- L2 Unified 256 KiB (x8)
- L3 Unified 16384 KiB
- Load Average: 1.64, 2.67, 2.84
- ---
- Benchmark Time CPU Iterations
- ---
- BM_FSMStateTransitions 1548 ns 1547 ns 444303
- BM_FSMWithEnums 1572 ns 1572 ns 452489
- BM_FSMWithStatePattern 2124 ns 2040 ns 319722

std::variant | alternative types

Can go in std::variant	Can't go in std::variant
Value types (int etc.)	C Arrays
POD structs	void (use std::monostate)
class	Reference types (use std::reference_wrapper)
pointers	
Unions	

- Types can be cv-qualified