statemachines

Telfort.

GitHub

github.com/rolfvreijdenberger/izzum-statemachine

about me

linkedin.com/in/rolfvreijdenberger

github.com/rolfvreijdenberger

co-founder

sharing knowledge

software architect fixed delivery streets

so much to talk about ...

- and so little time

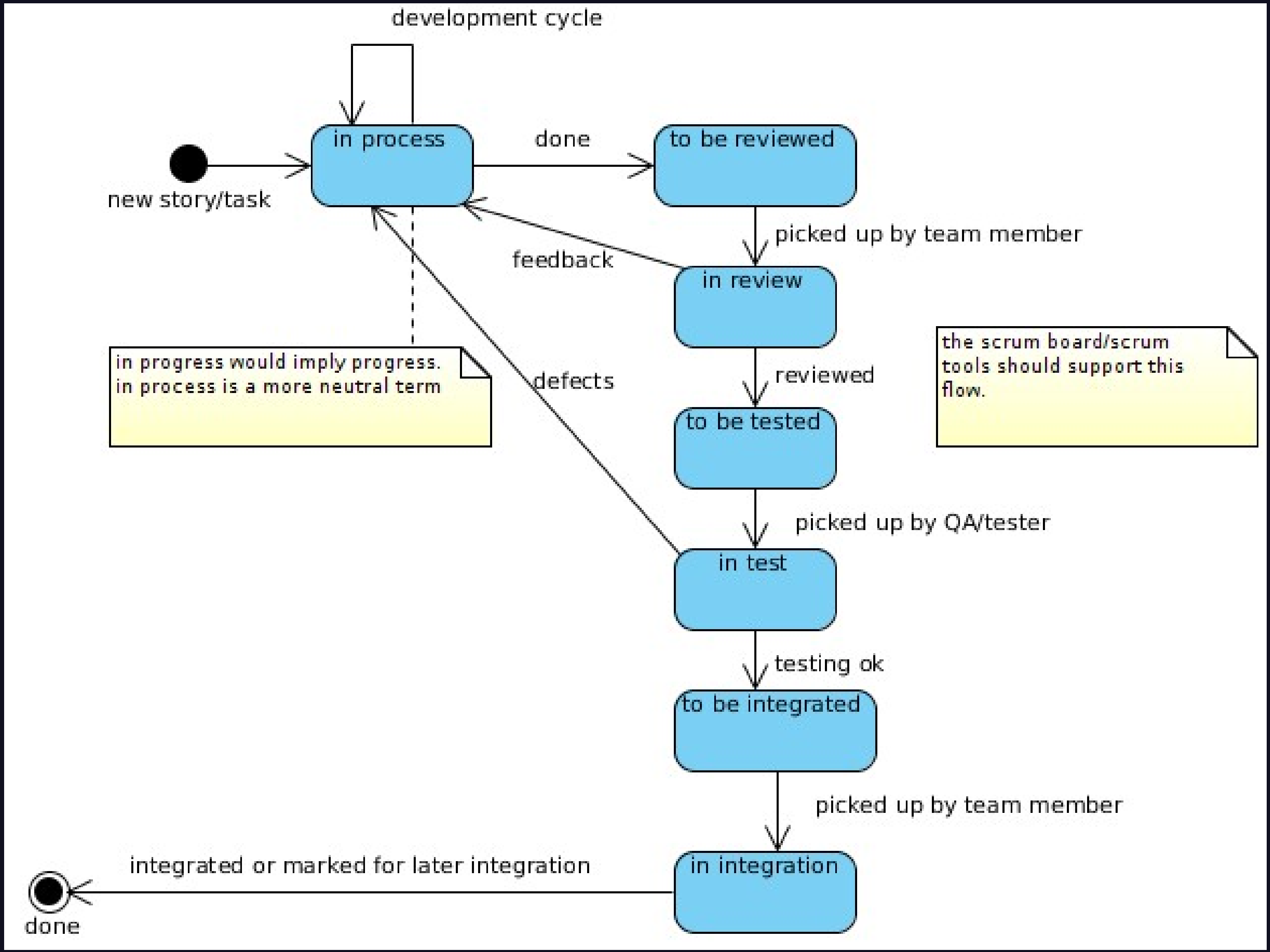
a little bit of theory

definition

- A finite statemachine is a model for the behaviour of a system that consists of a finite number of states.Transitions defined between those states can have guard logic and transition logic

- more:
  - https://en.wikipedia.org/wiki/Finite-state_machine
  - https://en.wikipedia.org/wiki/UML_state_machine

scrum workflow



in progress would imply progress.
in process is a more neutral term

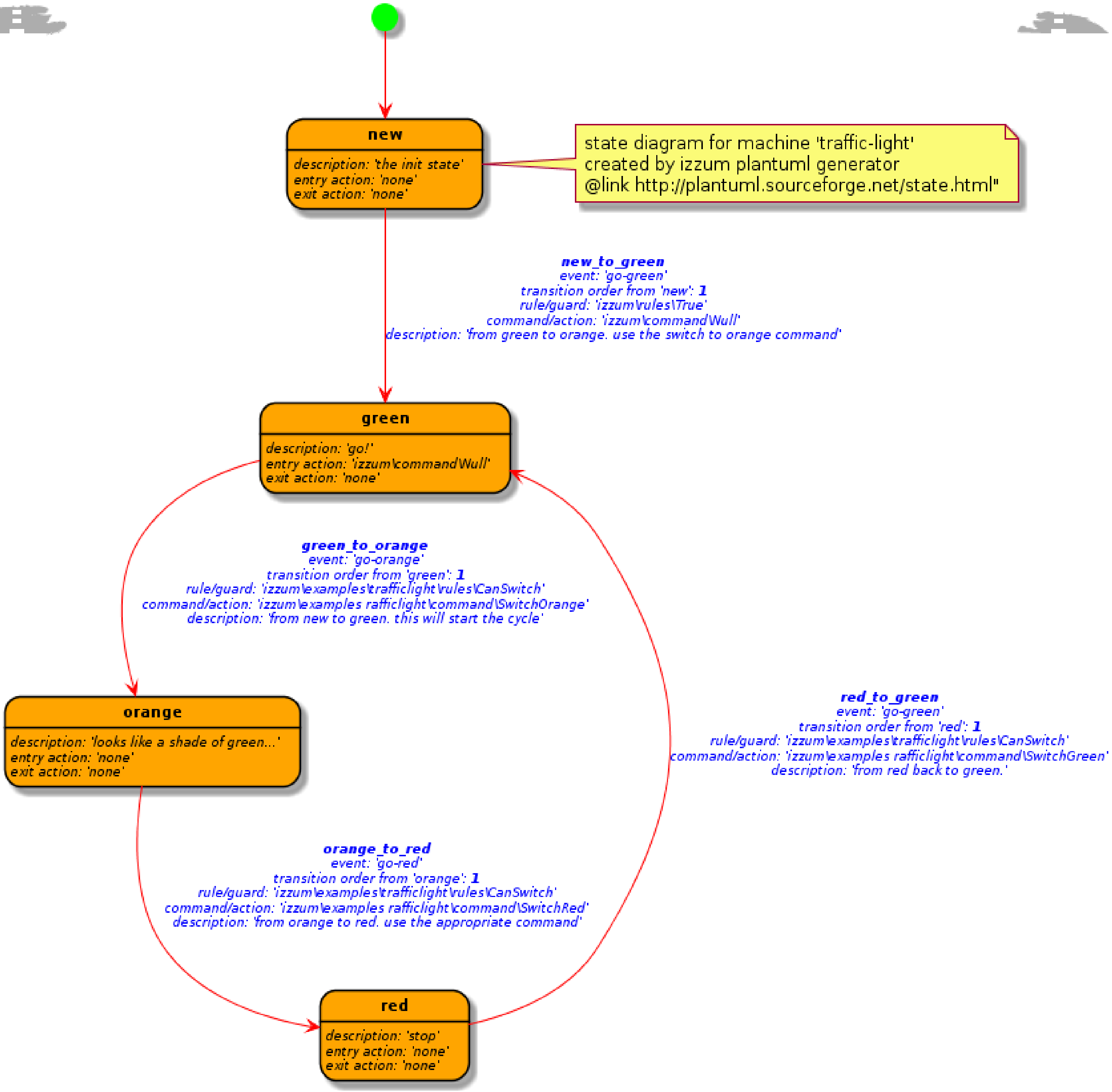
the scrum board/scrum tools should support this flow.

some concepts

- The machine is in only one state at a time: the current state.
- It can change from one state to another when initiated by a triggering event or condition; this is called a transition
- A transition can be (dis)allowed by guard logic
- Changing states can have logic executed as part of the transition

applications of a statemachine

- anything that has statefull behaviour
  - games
  - process flows
  - traffic lights
  - text parsing
  - protocol analysis
  - delivery streets
  - etc.



state diagram for machine 'traffic-light'
created by izzum plantuml generator
@link http://plantuml.sourceforge.net/state.html"

new
description: 'the init state'
entry action: 'none'
exit action: 'none'

new_to_green
event: 'go-green'
transition order from 'new': ***1***
rule/guard: 'izzum\rules\True'
command/action: 'izzum\command\Null'
description: 'from green to orange. use the switch to orange command'

green
description: 'go!'
entry action: 'izzum\command\Null'
exit action: 'none'

green_to_orange
event: 'go-orange'
transition order from 'green': ***1***
rule/guard: 'izzum\examples\trafficlight\rules\CanSwitch'
command/action: 'izzum\examples rafficlight\command\SwitchOrange'
description: 'from new to green. this will start the cycle'

orange
description: 'looks like a shade of green...'
entry action: 'none'
exit action: 'none'

orange_to_red
event: 'go-red'
transition order from 'orange': ***1***
rule/guard: 'izzum\examples\trafficlight\rules\CanSwitch'
command/action: 'izzum\examples rafficlight\command\SwitchRed'
description: 'from orange to red. use the appropriate command'

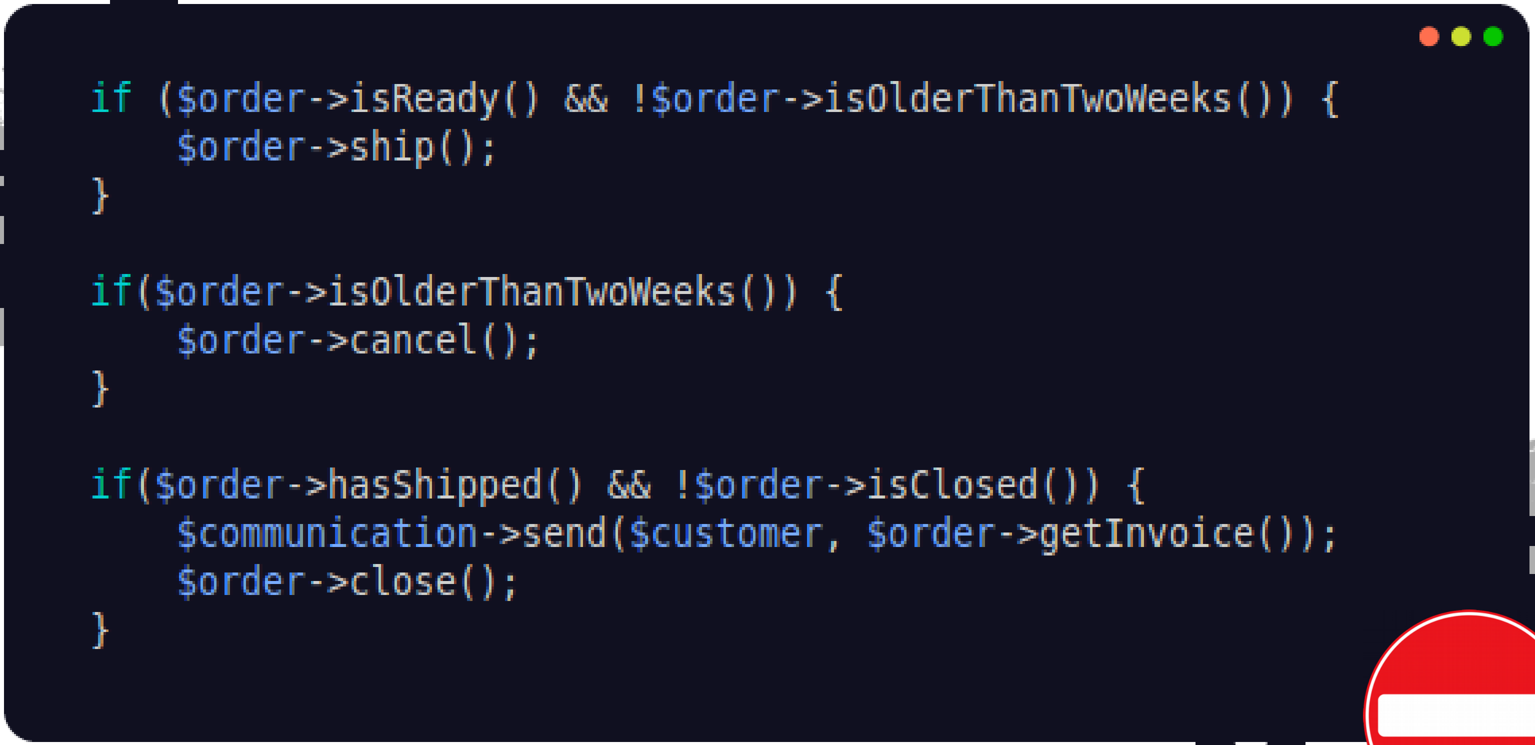
red
description: 'stop'
entry action: 'none'
exit action: 'none'

red_to_green
event: 'go-green'
transition order from 'red': ***1***
rule/guard: 'izzum\examples\trafficlight\rules\CanSwitch'
command/action: 'izzum\examples rafficlight\command\SwitchGreen'
description: 'from red back to green.'

when to use a statemachine?

- when state and status fields are all over your application: '*has_paid', 'is_shipped', 'date_sent' and of course 'state'*
- when business logic is closely coupled with these states: multiple status fields are checked to see if something should take place *(select * from order where .. and .. and .. and ..)*
- when a process lifecycle flow follows discrete steps with multiple paths through the lifecycle (graph)
- when you want to simplify following a sequence of actions through an application
- when mechanism (how) vs policy (what/when) is not clear: the policy of when should we do something (selection of states) is part of the mechanism of what you are doing (logic execution for those states)

no statemachine here



```php
if ($order->isReady() && !$order->isOlderThanTwoWeeks()) {
    $order->ship();
}

if($order->isOlderThanTwoWeeks()) {
    $order->cancel();
}

if($order->hasShipped() && !$order->isClosed()) {
    $communication->send($customer, $order->getInvoice());
    $order->close();
}
```

meanwhile, at Telfort

problems we encountered

- automating process flow in delivery streets with cron jobs does not scale well: performance suffers for batch jobs
- bugs were increasingly hard to solve
- tests for flows that are changing is hard
- business logic spread all over the place
- problems were solved inconsistently in the teams
- certain steps in the delivery streets did “too much”
- many status fields used in selection criteria for executing logic

enter the statemachine

- start of new delivery street for Telfort at end of 2013
- statemachine implementation early in 2014
- existing solutions were not good enough
  - they did not store state in a backend
  - implementations were not using encapsulated logic (business rules and business logic) for transitions
  - were not tailored to our needs
- requirements were made and implemented rapidly to make use of it asap
- reuse of already existing conceptual components

(some) requirements

- shall be non-invasive to domain models. they shall not know they are governed by a statemachine
- statemachine shall work with any domain model
- minimal information is needed to identify a machine {name, id}
- states shall be preserved between processes. data is stored in a backend of choice
- defining transitions, state and logic should be easy via configuration
- seperate policy and mechanism
- interfacing with the statemachines shall be consistent and simple
- guard and transition logic shall be implemented in rules and commands, for which we can store the fully qualified classnames in our configuration in a backend of choice
- etc.

defining a statemachine

- *name*: the type identifier for what the machine is used for
  - this is more about the function of the process than about the domain model
  - order, change-order, customer-debt-management etc.
- *entity_id*: the unique id of an entity (*domain model*) for the machine
  - most probably a primary key in your application
  - maps naturally to the id of a domain model
- the *{name, entity_id}* machine will act on a specific domain model
  - {change-order, 4274} will be the statemachine that handles the flow of a change order on the domain model 'Order' with id 4274

so what can we use for our statemachine needs?

introducing izzum

- github.com/rolfvreijdenberger/izzum-statemachine
- php opensource implementation

GitHub

license MIT

php

about izzum

- fully documented & quality code
- feature rich while easy to use
- advanced features for power users
- extensible for your problem domain
- high test coverage
- examples included
- formal and less formal usage possible
- build passing | stable 3.2.3 | coverage 92 % | Scrutinizer 8.78

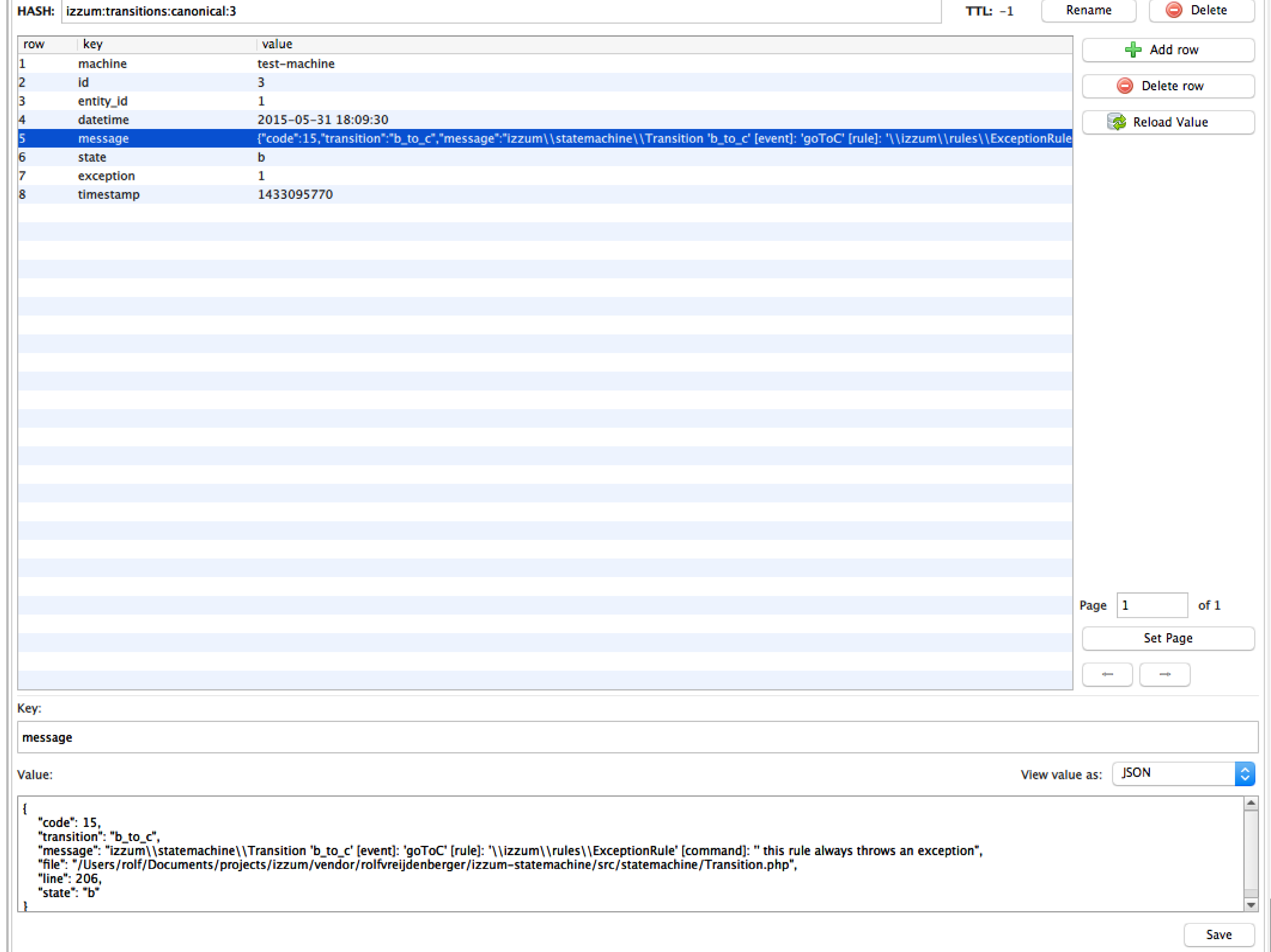
izzum storage & configuration

- works with different backends for storing state and transition history (+ write your own)

mongoDB PostgreSQL SQLite redis MySQL

- handles configuration of machines in different data description formats

SQL XML YAML {JSON} php



HASH: izzum:transitions:canonical:3

TTL: -1

row	key	value
1	machine	test-machine
2	id	3
3	entity_id	1
4	datetime	2015-05-31 18:09:30
5	message	{"code":15,"transition":"b_to_c","message":"izzum\\statemachine\\Transition 'b_to_c' [event]: 'goToC' [rule]: '\\izzum\\rules\\ExceptionRule
6	state	b
7	exception	1
8	timestamp	1433095770

Page 1 of 1

Key:

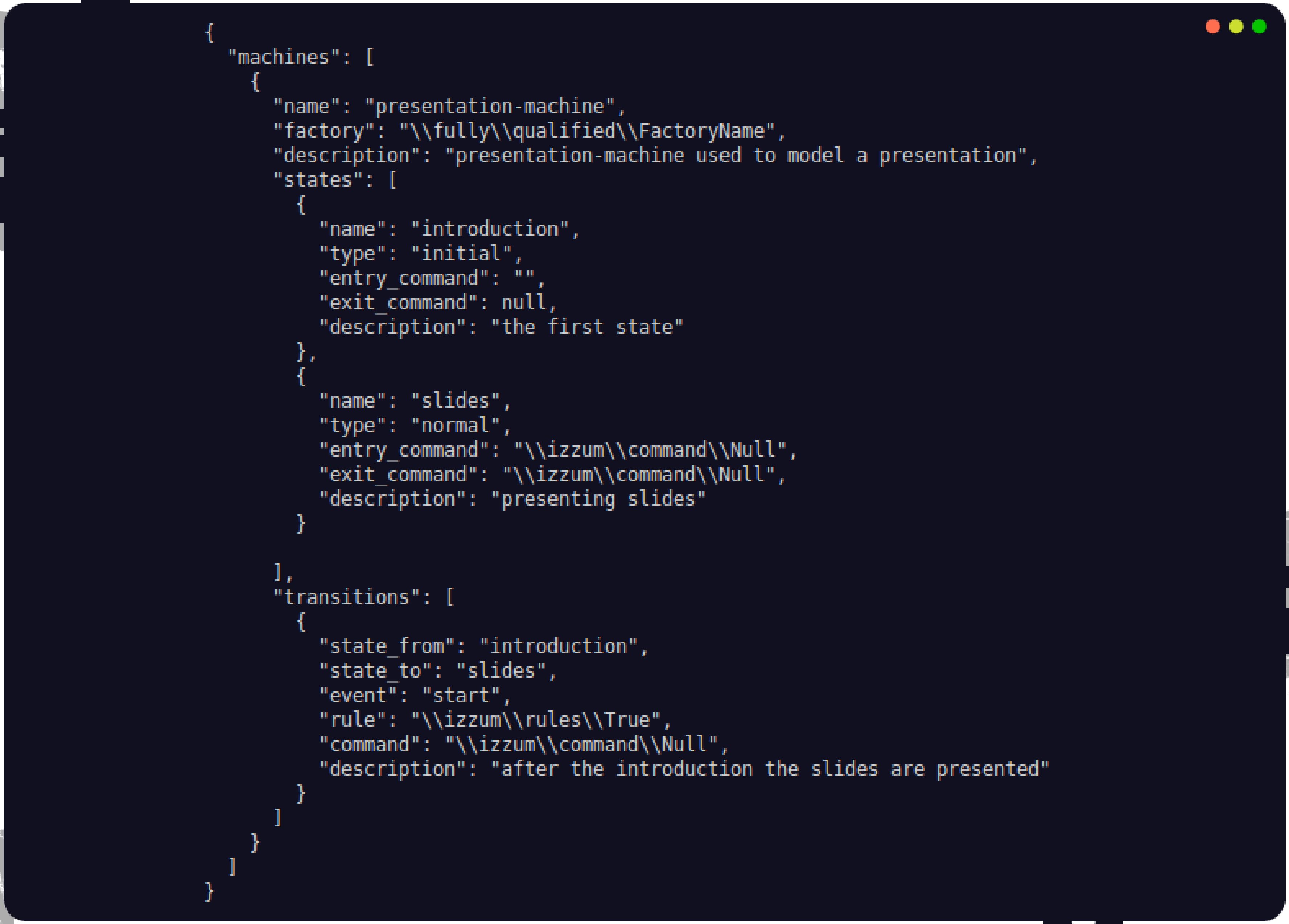
message

Value:

View value as: JSON

```
{
    "code": 15,
    "transition": "b_to_c",
    "message": "izzum\\statemachine\\Transition 'b_to_c' [event]: 'goToC' [rule]: '\\izzum\\rules\\ExceptionRule' [command]: '' this rule always throws an exception",
    "file": "/Users/rolf/Documents/projects/izzum/vendor/rolfvreijdenberger/izzum-statemachine/src/statemachine/Transition.php",
    "line": 206,
    "state": "b"
}
```

configuration in json



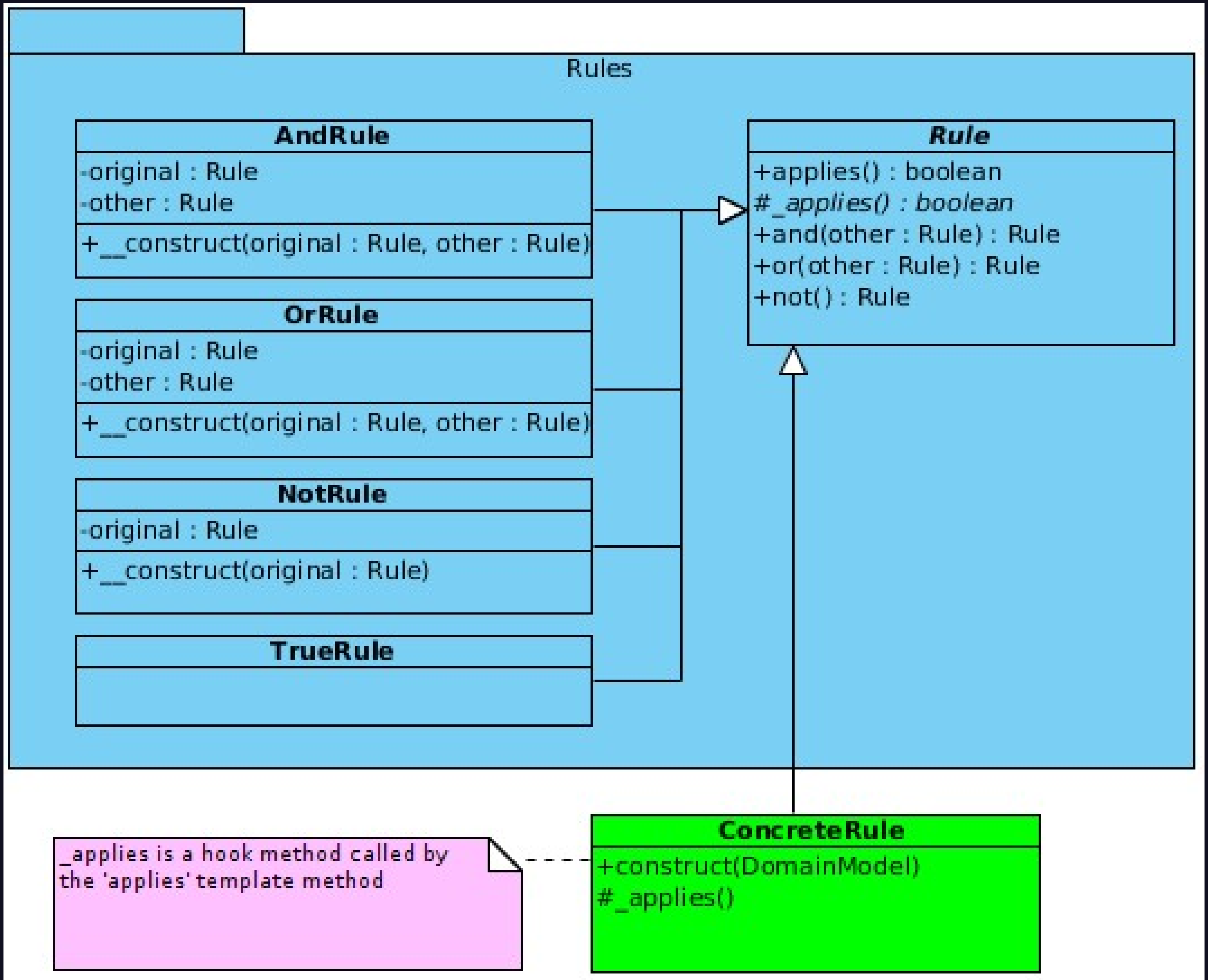
```
{
  "machines": [
    {
      "name": "presentation-machine",
      "factory": "\\fully\\qualified\\FactoryName",
      "description": "presentation-machine used to model a presentation",
      "states": [
        {
          "name": "introduction",
          "type": "initial",
          "entry_command": "",
          "exit_command": null,
          "description": "the first state"
        },
        {
          "name": "slides",
          "type": "normal",
          "entry_command": "\\izzum\\command\\Null",
          "exit_command": "\\izzum\\command\\Null",
          "description": "presenting slides"
        }

      ],
      "transitions": [
        {
          "state_from": "introduction",
          "state_to": "slides",
          "event": "start",
          "rule": "\\izzum\\rules\\True",
          "command": "\\izzum\\command\\Null",
          "description": "after the introduction the slides are presented"
        }
      ]
    }
  ]
}
```

core concepts of izzum

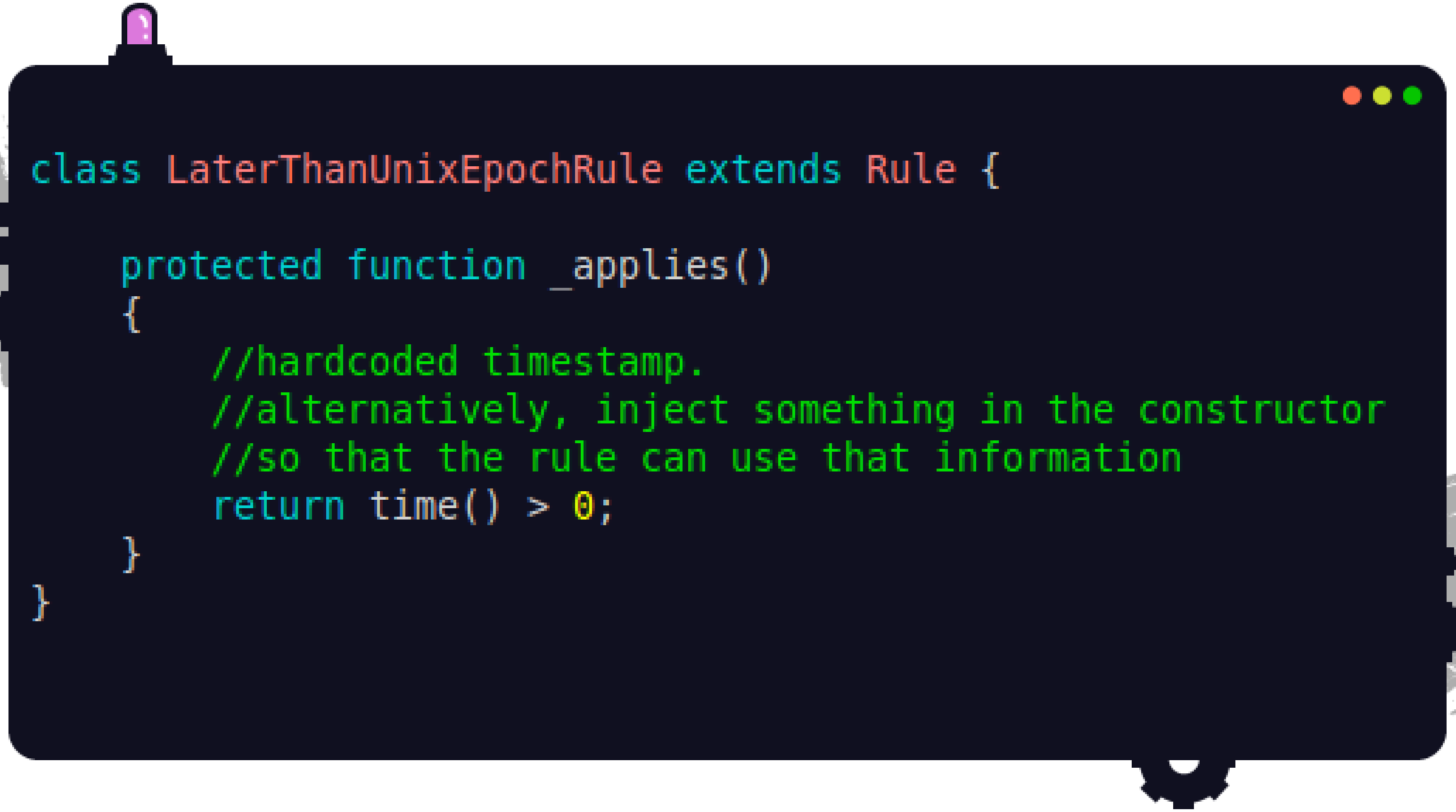
rules: guarding a transition

- function: guard logic. determine if a transition is (dis)allowed
- are encapsulating business rules that might allow a transition
- are all about 'policy' (as opposed to mechanism)
- return true or false for the 'applies()' method, have no side effects
- are subclasses of the \Rule class in the 'rules' package
- a 'True Rule' is used when a transition is allowed by default
- are instantiated at runtime from their fully qualified class name
- have a domain model (associated with a statemachine) injected via the constructor on which it can act
- rules are set on the definition of a transition (fully qualified classname)
- can be queried as to why it did not apply

rule class diagram

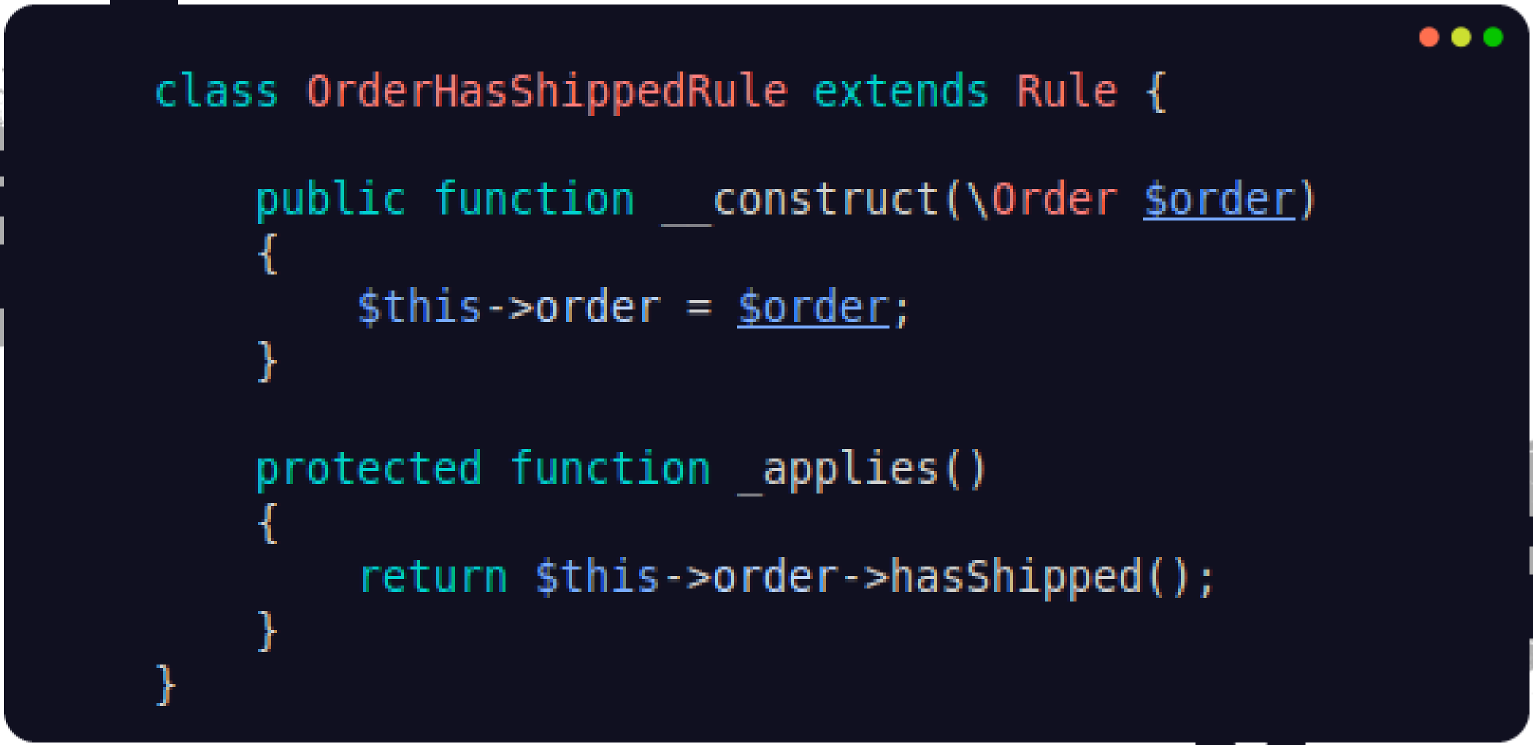


rule: simple



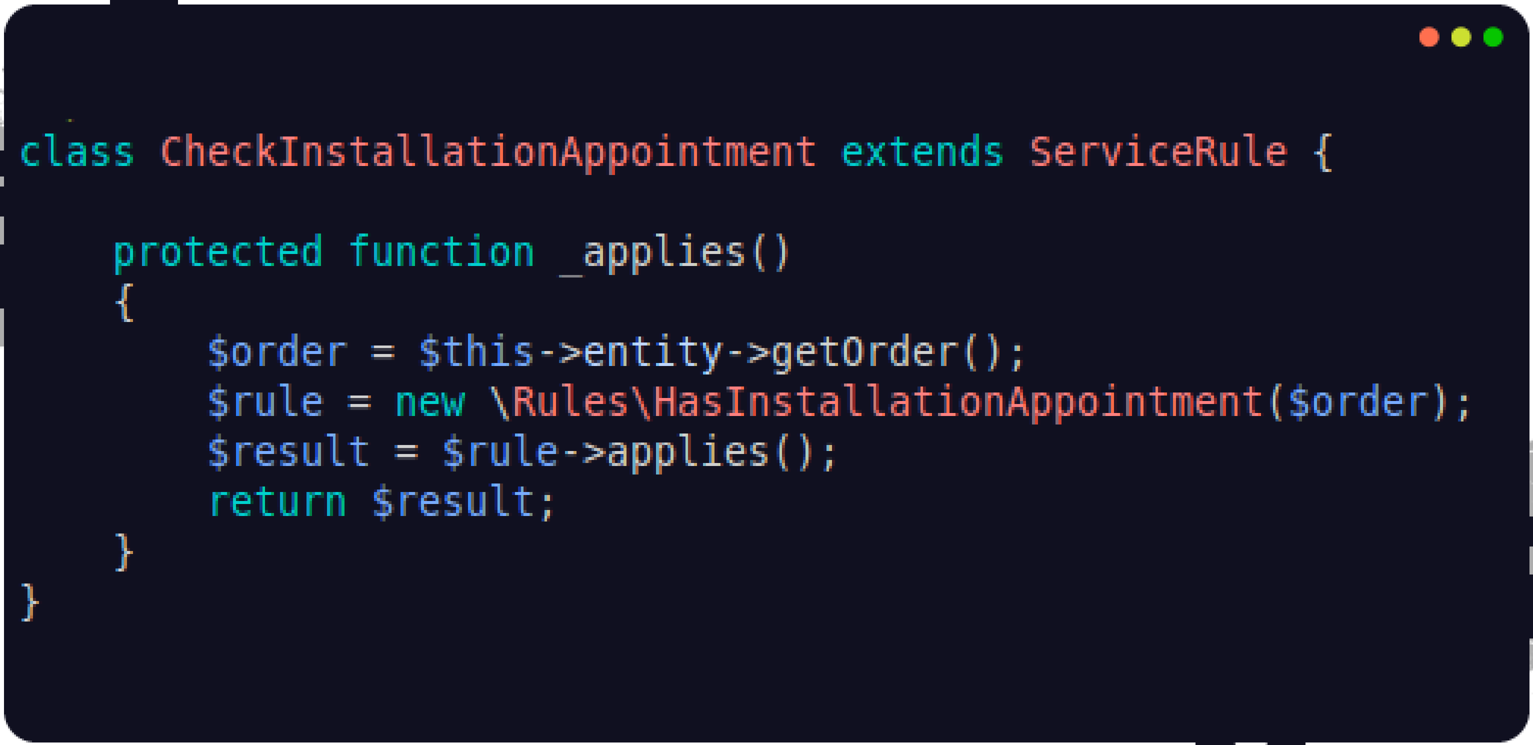
```
class LaterThanUnixEpochRule extends Rule {

    protected function _applies()
    {
        //hardcoded timestamp.
        //alternatively, inject something in the constructor
        //so that the rule can use that information
        return time() > 0;
    }
}
```

rule: using a dependency



```php
class OrderHasShippedRule extends Rule {

    public function __construct(\Order $order)
    {
        $this->order = $order;
    }

    protected function _applies()
    {
        return $this->order->hasShipped();
    }
}
```

rule: using entity and delegating to existing rule



```
class CheckInstallationAppointment extends ServiceRule {

    protected function _applies()
    {
        $order = $this->entity->getOrder();
        $rule = new \Rules\HasInstallationAppointment($order);
        $result = $rule->applies();
        return $result;
    }
}
```

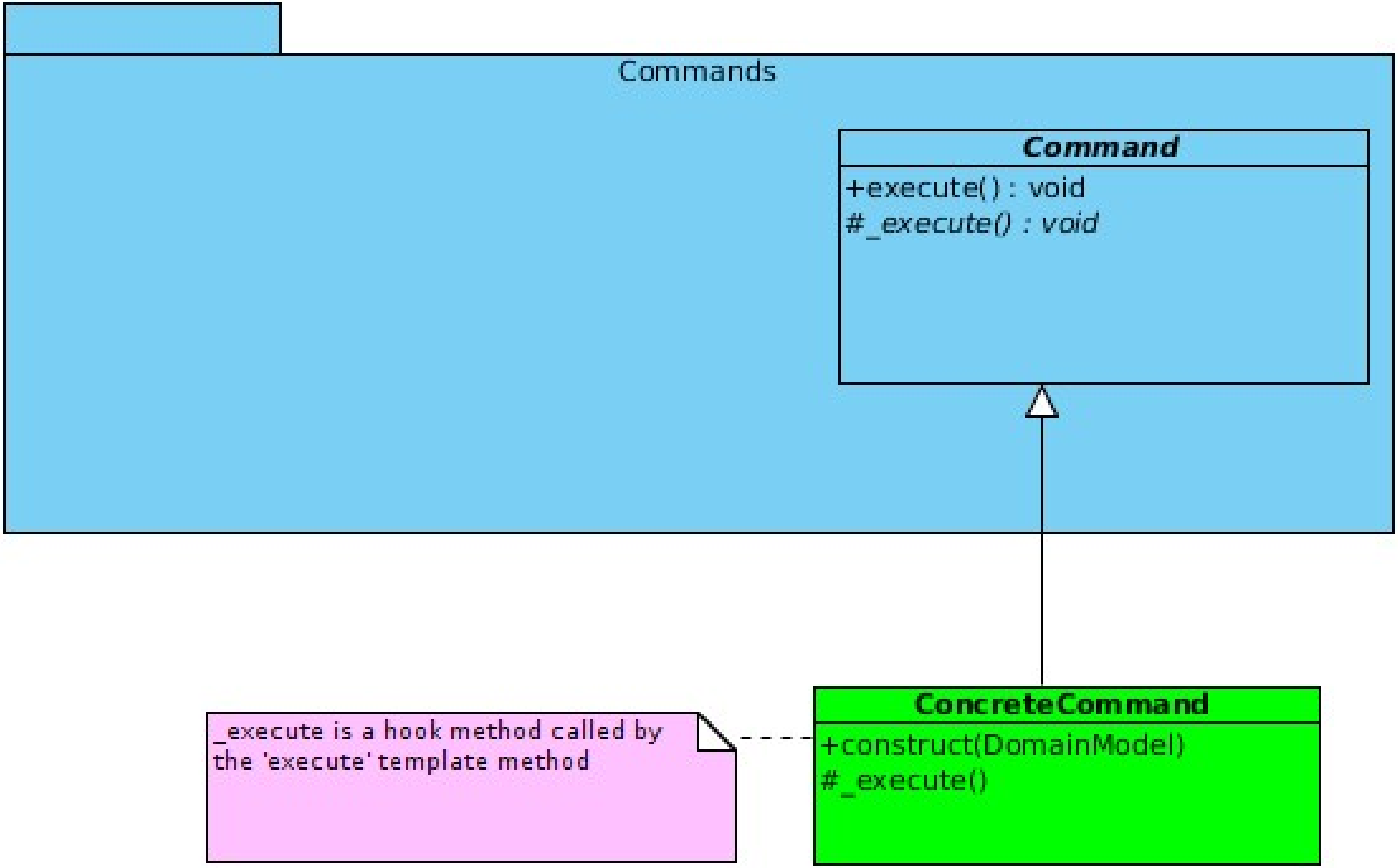
kinds of transition logic

- *exit logic*: associated with leaving a state, independent of the sink of the transition
- *entry logic*: associated with transitioning into a state, independent of the source of the transition
- *transition logic*: associated with a transition between 2 states

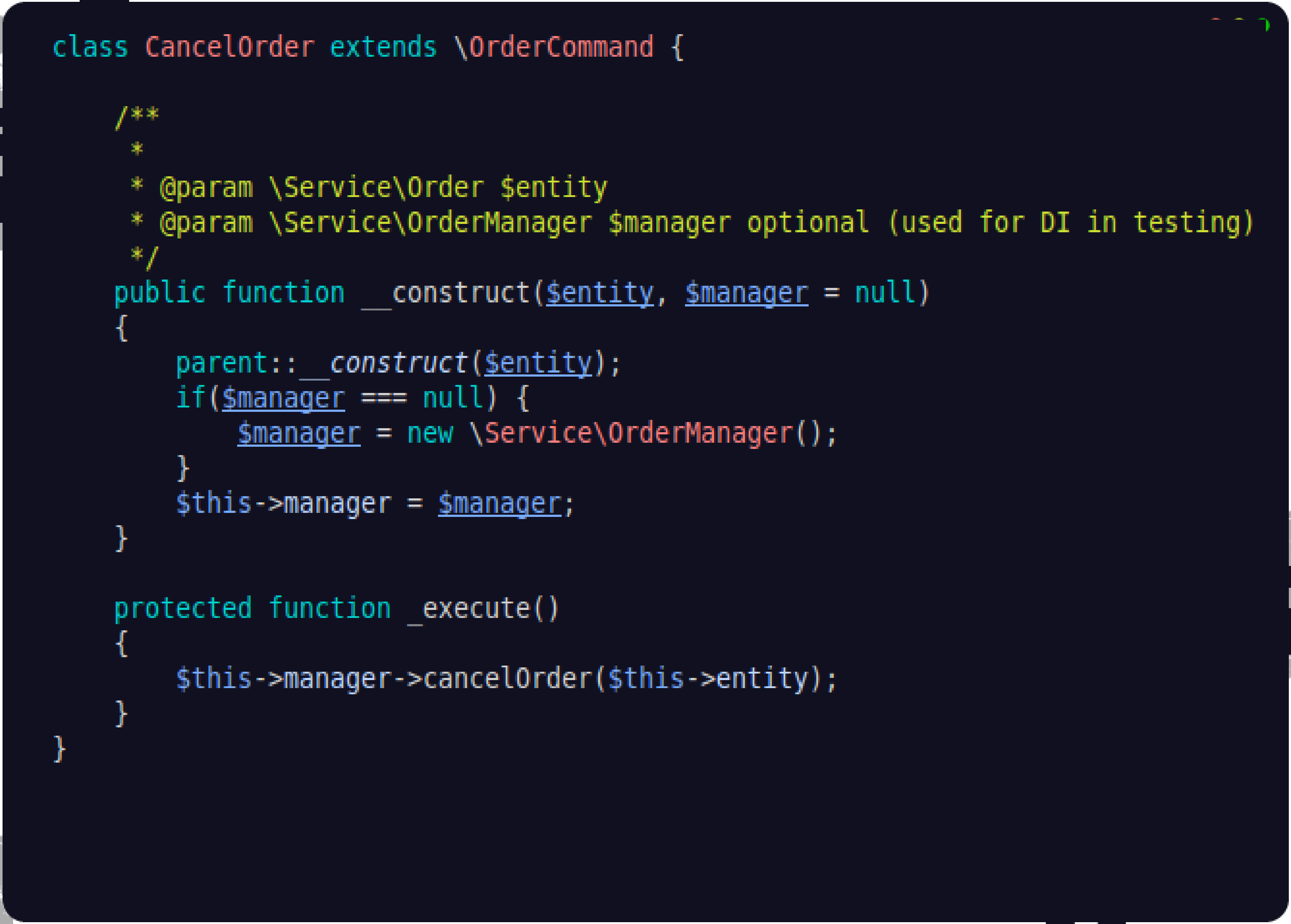
commands: transition logic

- function: *transition logic*. execute functionality associated with a transition and/or a state (entry/exit). These do the hard work
- are about 'mechanism' (as opposed to policy)
- are based on the 'Command' design pattern: *“a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time”*
- can have a side effect as part of the transition
- are subclasses of the \Command class in the 'commands' package
- implement the 'execute' method
- Use a 'Null Command' when no logic is needed
- are instantiated at runtime from their fully qualified class name
- have a domain model (associated with a statemachine) injected via the constructor
- can act on the domain model to alter data, use 3d party services etc
- commands are set on the definition of a transition or on those of a state (entry/exit) with a fully qualified classname.

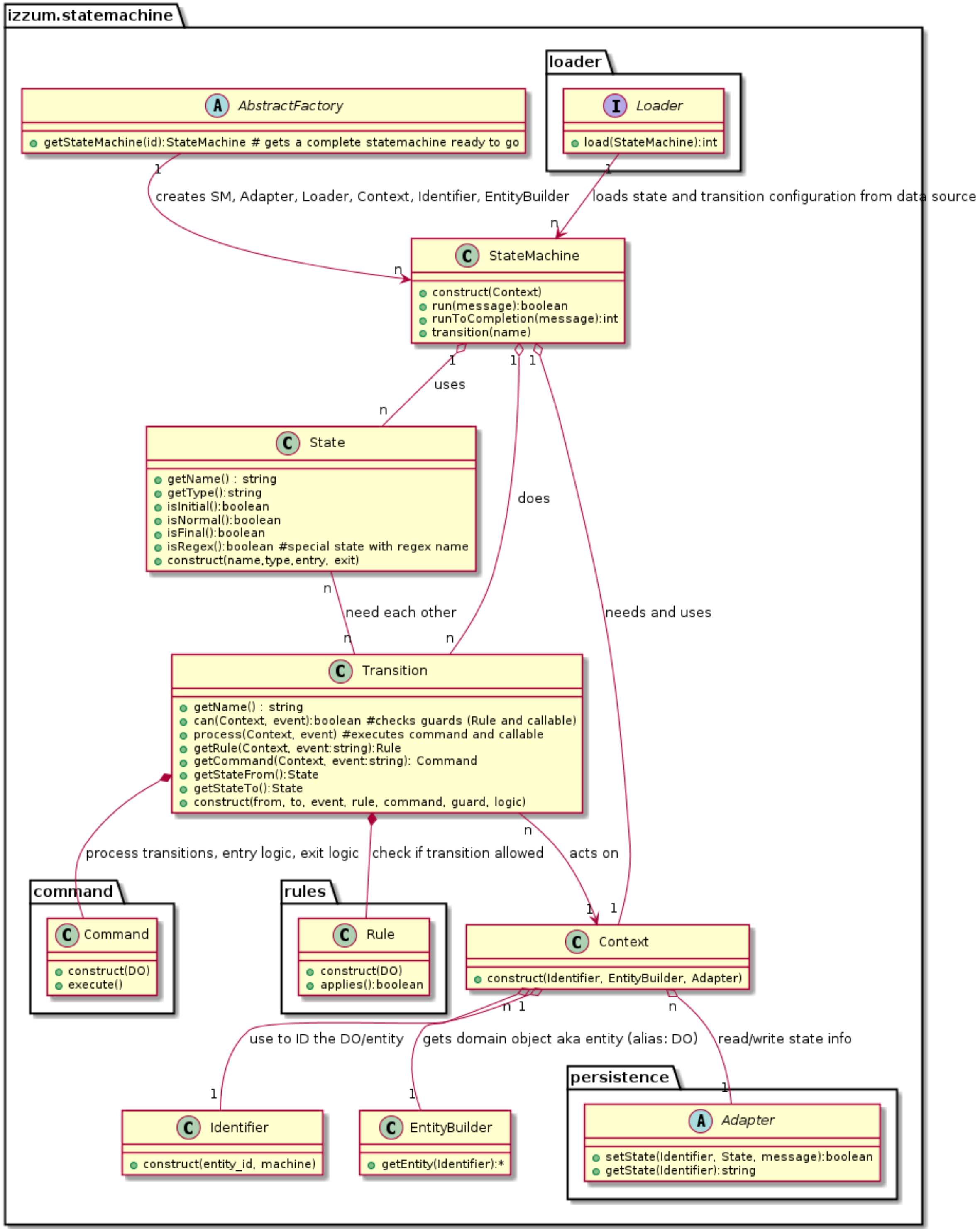
command class diagram



command



```
class CancelOrder extends \OrderCommand {

    /**
     *
     * @param \Service\Order $entity
     * @param \Service\OrderManager $manager optional (used for DI in testing)
     */
    public function __construct($entity, $manager = null)
    {
        parent::__construct($entity);
        if($manager === null) {
            $manager = new \Service\OrderManager();
        }
        $this->manager = $manager;
    }

    protected function _execute()
    {
        $this->manager->cancelOrder($this->entity);
    }
}
```



izzum.statemachine

loader

AbstractFactory
- getStateMachine(id):StateMachine # gets a complete statemachine ready to go

Loader
- load(StateMachine):int

1 creates SM, Adapter, Loader, Context, Identifier, EntityBuilder n

1 loads state and transition configuration from data source n

StateMachine
- construct(Context)
- run(message):boolean
- runToCompletion(message):int
- transition(name)

1 uses n

1 does n

1 needs and uses 1

State
- getName() : string
- getType():string
- isInitial():boolean
- isNormal():boolean
- isFinal():boolean
- isRegex():boolean #special state with regex name
- construct(name,type,entry, exit)

n need each other n

Transition
- getName() : string
- can(Context, event):boolean #checks guards (Rule and callable)
- process(Context, event) #executes command and callable
- getRule(Context, event:string):Rule
- getCommand(Context, event:string): Command
- getStateFrom():State
- getStateTo():State
- construct(from, to, event, rule, command, guard, logic)

process transitions, entry logic, exit logic

check if transition allowed

n acts on 1

command

Command
- construct(DO)
- execute()

rules

Rule
- construct(DO)
- applies():boolean

Context
- construct(Identifier, EntityBuilder, Adapter)

n use to ID the DO/entity 1

1 gets domain object aka entity (alias: DO) 1

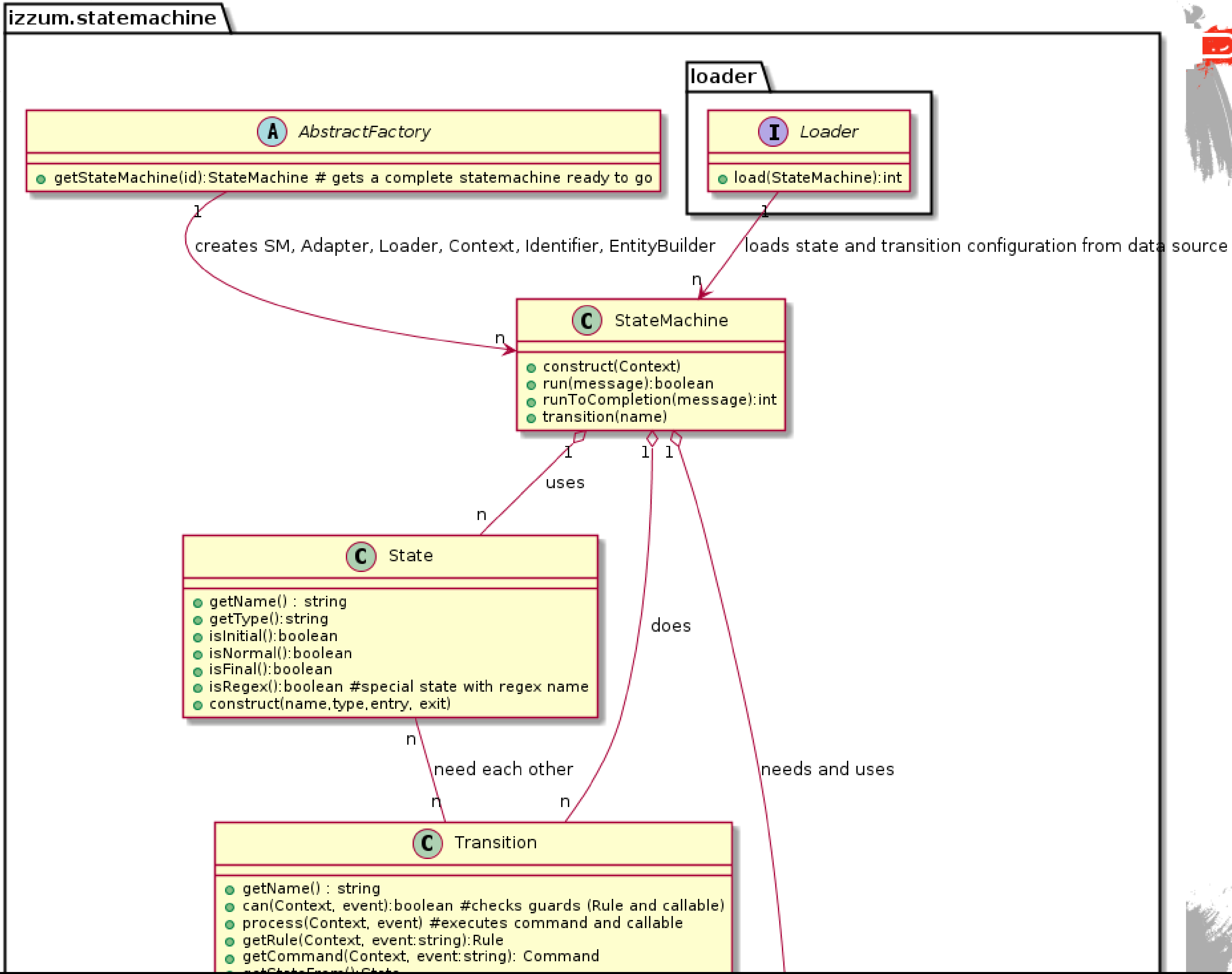
n read/write state info 1

Identifier
- construct(entity_id, machine)

EntityBuilder
- getEntity(Identifier):*

persistence

Adapter
- setState(Identifier, State, message):boolean
- getState(Identifier):string



izzum.statemachine

loader

Loader

+ load(StateMachine):int

AbstractFactory

+ getStateMachine(id):StateMachine # gets a complete statemachine ready to go

1 creates SM, Adapter, Loader, Context, Identifier, EntityBuilder n

1 loads state and transition configuration from data source n

StateMachine

+ construct(Context)
+ run(message):boolean
+ runToCompletion(message):int
+ transition(name)

1 uses n

1 does n

1 needs and uses

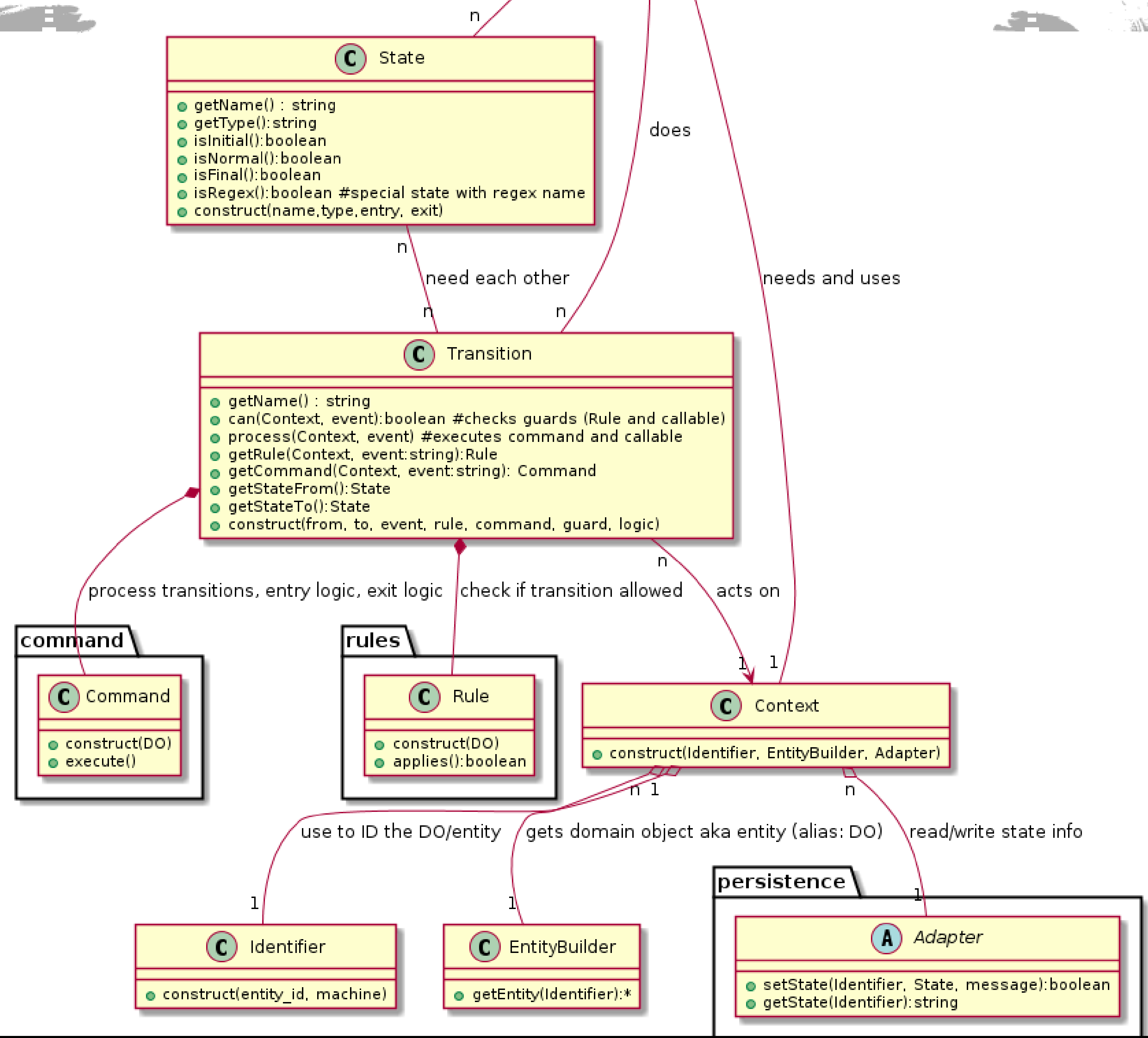
State

+ getName() : string
+ getType():string
+ isInitial():boolean
+ isNormal():boolean
+ isFinal():boolean
+ isRegex():boolean #special state with regex name
+ construct(name,type,entry, exit)

n need each other n

Transition

+ getName() : string
+ can(Context, event):boolean #checks guards (Rule and callable)
+ process(Context, event) #executes command and callable
+ getRule(Context, event:string):Rule
+ getCommand(Context, event:string): Command
+ getStateFrom():State



n

does

needs and uses

State

- getName() : string
- getType():string
- isInitial():boolean
- isNormal():boolean
- isFinal():boolean
- isRegex():boolean #special state with regex name
- construct(name,type,entry, exit)

n need each other n n

Transition

- getName() : string
- can(Context, event):boolean #checks guards (Rule and callable)
- process(Context, event) #executes command and callable
- getRule(Context, event:string):Rule
- getCommand(Context, event:string): Command
- getStateFrom():State
- getStateTo():State
- construct(from, to, event, rule, command, guard, logic)

process transitions, entry logic, exit logic

check if transition allowed

n acts on 1 1

command

Command

- construct(DO)
- execute()

rules

Rule

- construct(DO)
- applies():boolean

Context

- construct(Identifier, EntityBuilder, Adapter)

n 1 n

use to ID the DO/entity

gets domain object aka entity (alias: DO)

read/write state info

1 1 1

Identifier

- construct(entity_id, machine)

EntityBuilder

- getEntity(Identifier):*

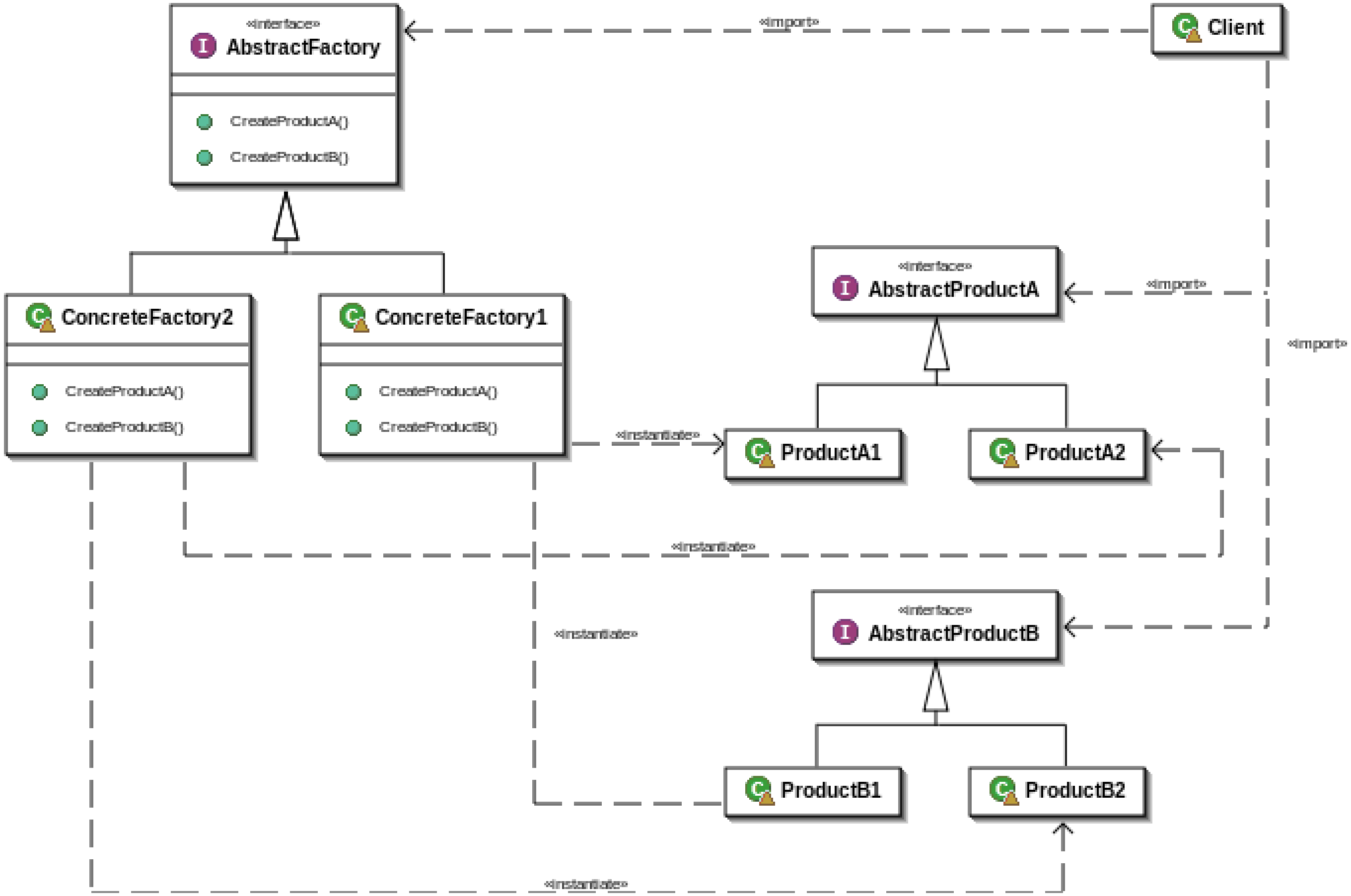
persistence

Adapter

- setState(Identifier, State, message):boolean
- getState(Identifier):string

so how can we use this to create tooling?

Abstract Factory Pattern



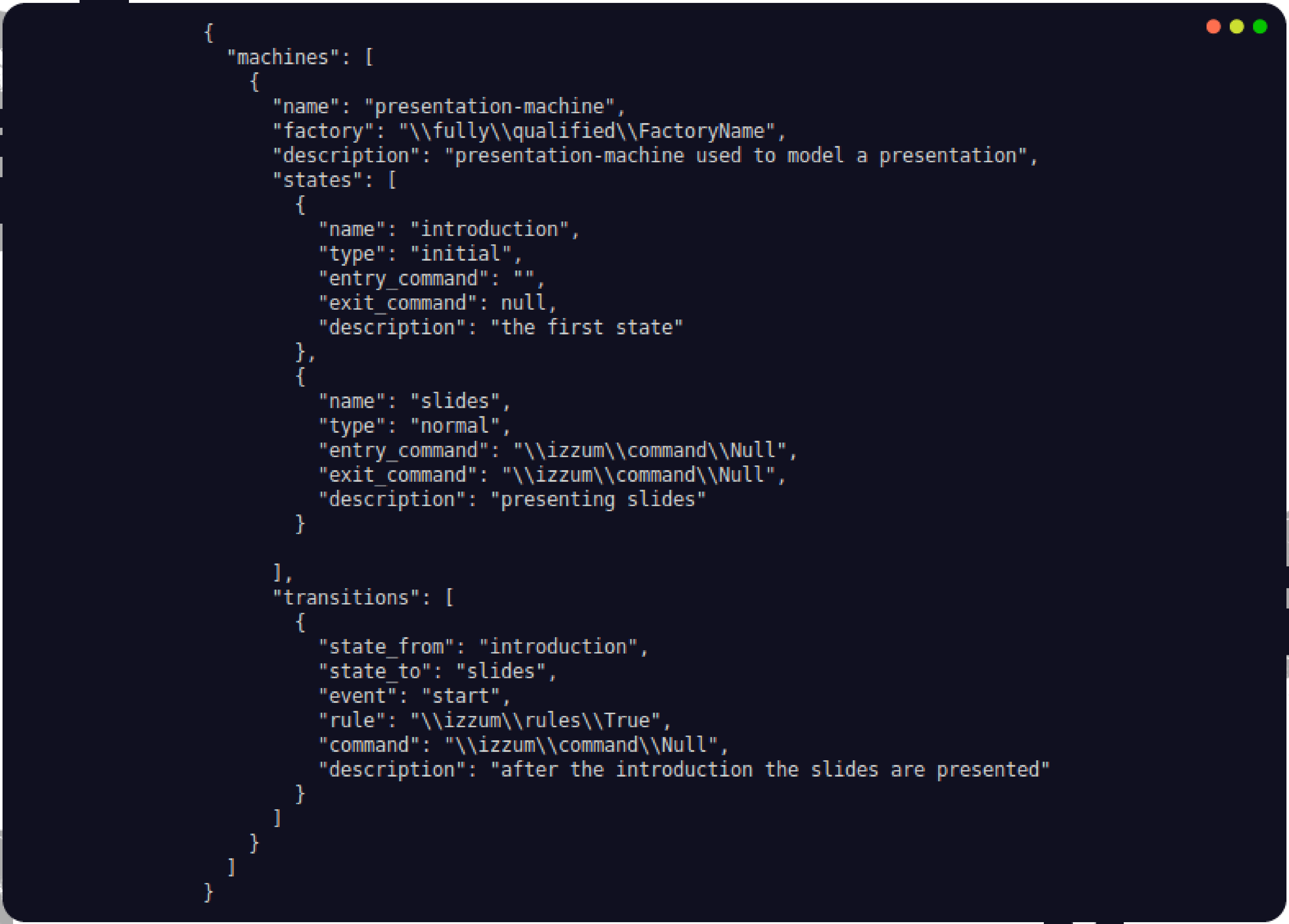
Abstract Factory Pattern

- *“provide an interface for creating families of related or dependent objects without specifying their concrete classes”*
  - **statemachine**: the class that handles all our transitions
  - **loader**: retrieve the definition of the statemachine: json, xml, sql, nosql, php etc.
  - **persistence adapter**: persist to memory, sql, session, mongo, redis etc.
  - **entity builder**: creates a domain object with the help of the id specified in the machine definition

Abstract Factory Pattern

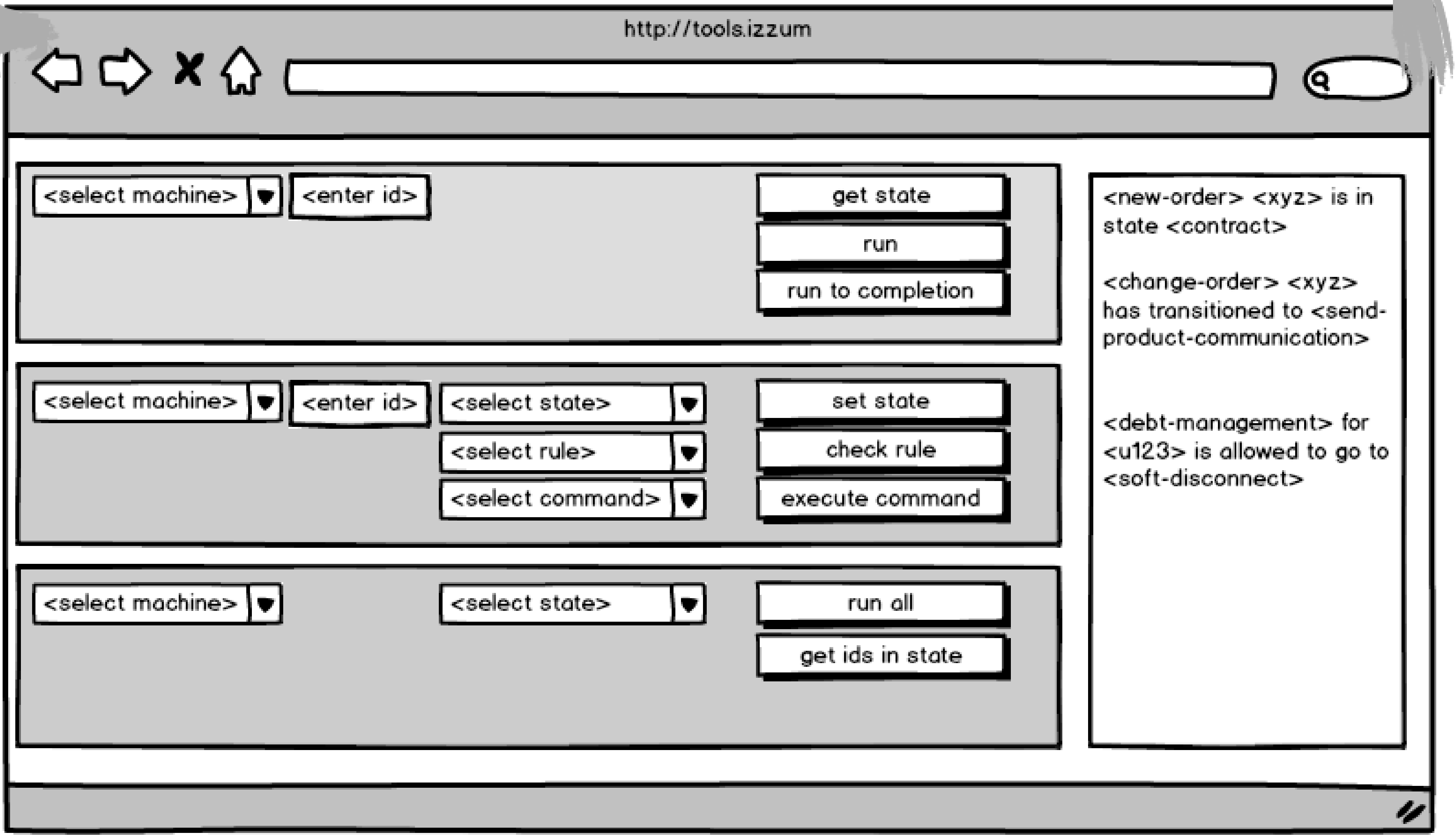
- each machine has it's own factory
- each machine can be instantiated via the factory
- the fully qualified factory classname is used to create statemachines
- statemachines can be handled polymorphically
- this allows us to design a GUI that handles all statemachines

configuration in json



```
{
  "machines": [
    {
      "name": "presentation-machine",
      "factory": "\\fully\\qualified\\FactoryName",
      "description": "presentation-machine used to model a presentation",
      "states": [
        {
          "name": "introduction",
          "type": "initial",
          "entry_command": "",
          "exit_command": null,
          "description": "the first state"
        },
        {
          "name": "slides",
          "type": "normal",
          "entry_command": "\\izzum\\command\\Null",
          "exit_command": "\\izzum\\command\\Null",
          "description": "presenting slides"
        }

      ],
      "transitions": [
        {
          "state_from": "introduction",
          "state_to": "slides",
          "event": "start",
          "rule": "\\izzum\\rules\\True",
          "command": "\\izzum\\command\\Null",
          "description": "after the introduction the slides are presented"
        }
      ]
    }
  ]
}
```

tools

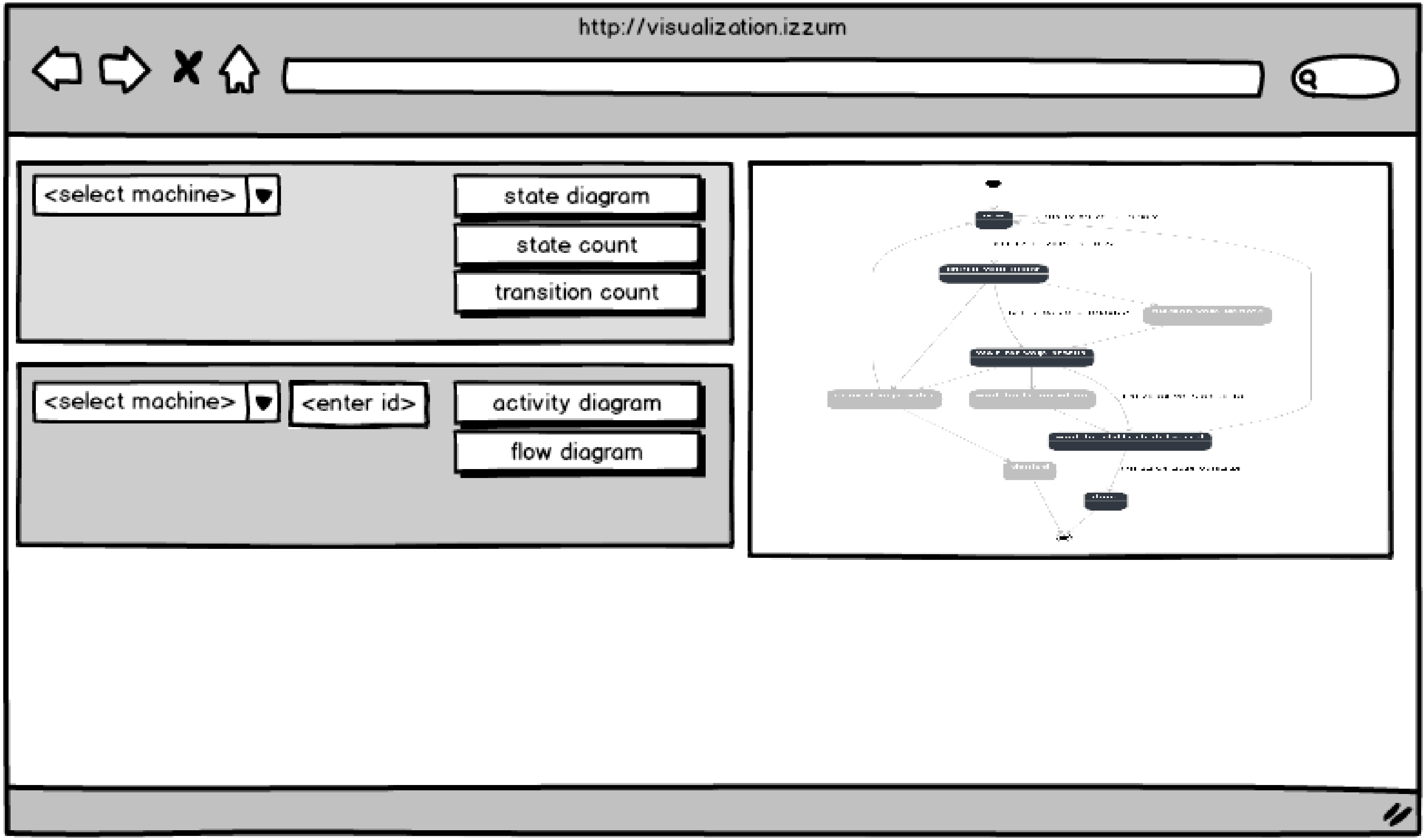


<new-order> <xyz> is in state <contract>

<change-order> <xyz> has transitioned to <send-product-communication>

<debt-management> for <u123> is allowed to go to <soft-disconnect>

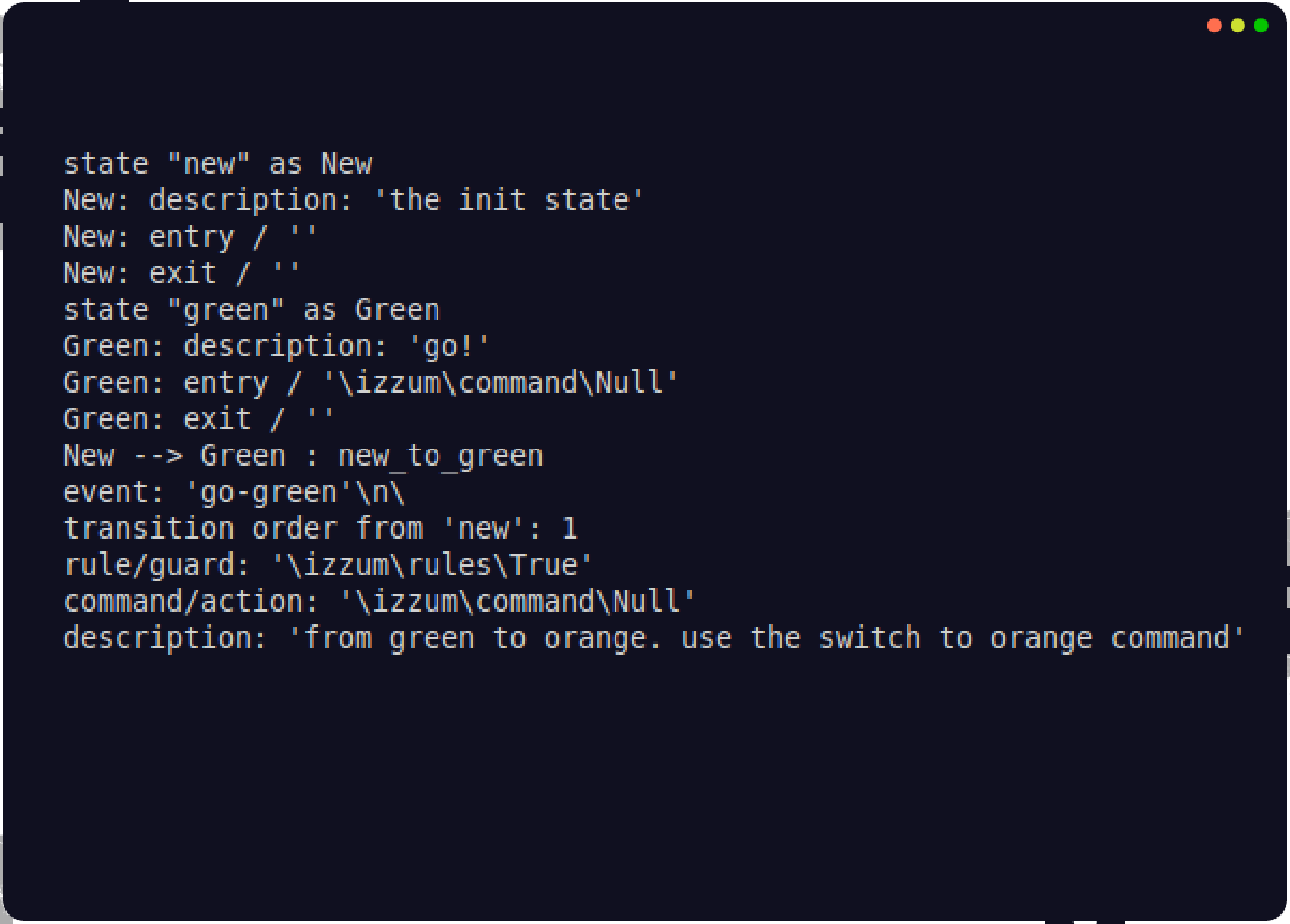
uml generation



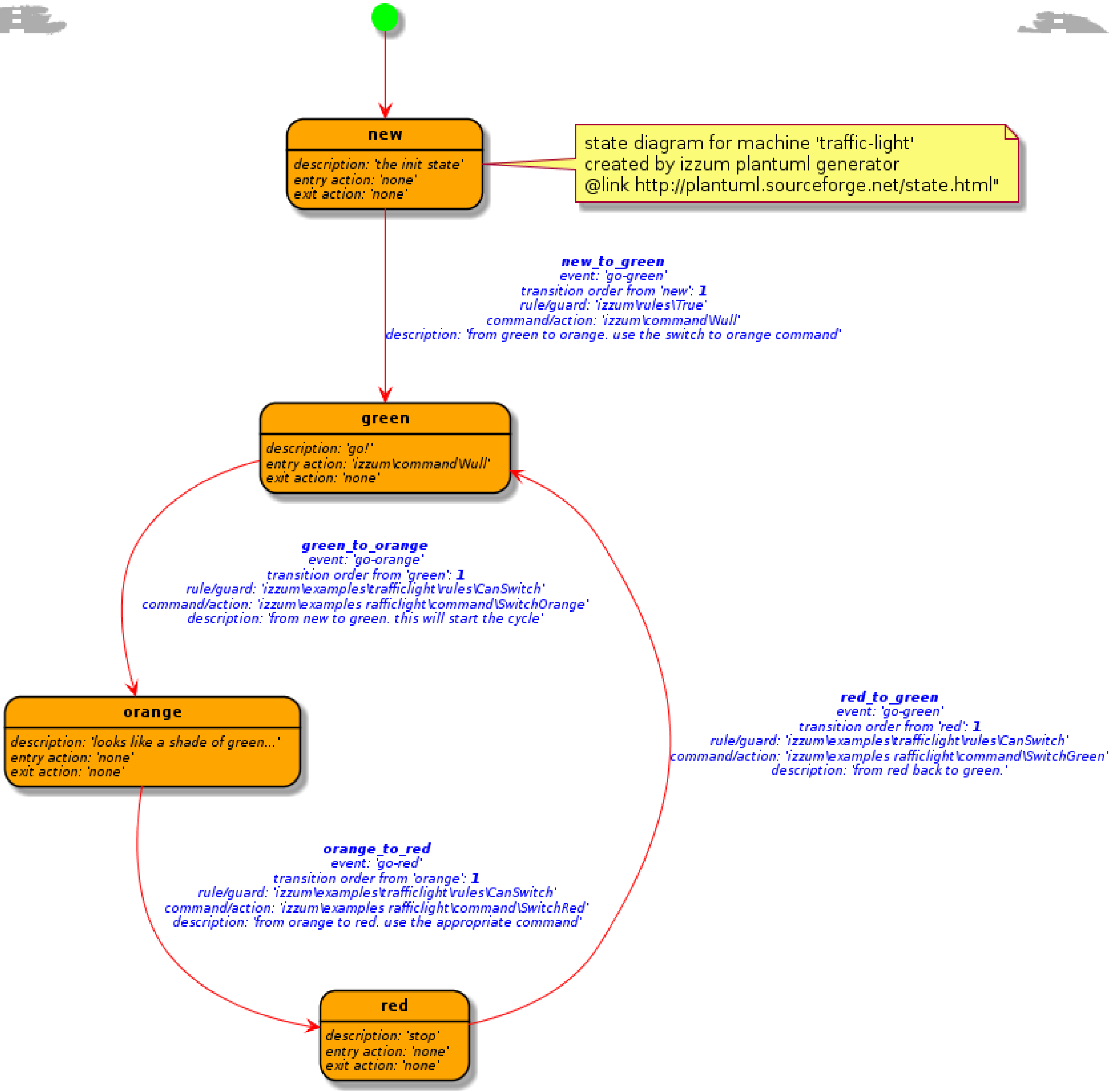
uml generation

- http://plantuml.com
  - Open-source tool that uses simple textual descriptions to draw UML diagrams
  - uses graphviz (http://www.graphviz.org/)
- allows generation of diagrams from statemachine data
  - state diagrams
  - history
  - statistics

plantuml syntax



```
state "new" as New
New: description: 'the init state'
New: entry / ''
New: exit / ''
state "green" as Green
Green: description: 'go!'
Green: entry / '\izzum\command\Null'
Green: exit / ''
New --> Green : new_to_green
event: 'go-green'\n\
transition order from 'new': 1
rule/guard: '\izzum\rules\True'
command/action: '\izzum\command\Null'
description: 'from green to orange. use the switch to orange command'
```



new

description: 'the init state'
entry action: 'none'
exit action: 'none'

state diagram for machine 'traffic-light'
created by izzum plantuml generator
@link http://plantuml.sourceforge.net/state.html"

new_to_green
event: 'go-green'
transition order from 'new': ***1***
rule/guard: 'izzum\rules\True'
command/action: 'izzum\command\Null'
description: 'from green to orange. use the switch to orange command'

green

description: 'go!'
entry action: 'izzum\command\Null'
exit action: 'none'

green_to_orange
event: 'go-orange'
transition order from 'green': ***1***
rule/guard: 'izzum\examples\trafficlight\rules\CanSwitch'
command/action: 'izzum\examples rafficlight\command\SwitchOrange'
description: 'from new to green. this will start the cycle'

orange

description: 'looks like a shade of green...'
entry action: 'none'
exit action: 'none'

red_to_green
event: 'go-green'
transition order from 'red': ***1***
rule/guard: 'izzum\examples\trafficlight\rules\CanSwitch'
command/action: 'izzum\examples rafficlight\command\SwitchGreen'
description: 'from red back to green.'

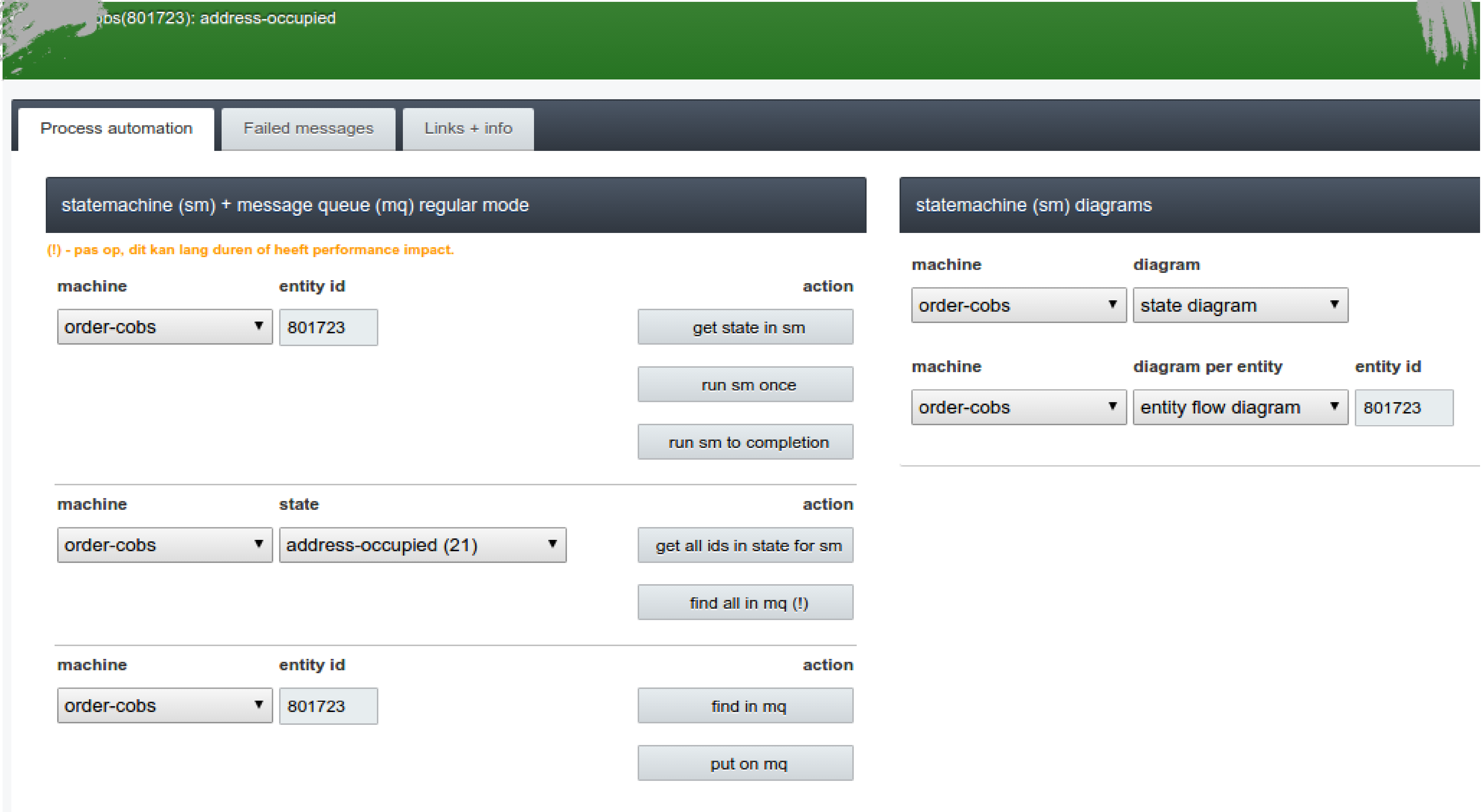
orange_to_red
event: 'go-red'
transition order from 'orange': ***1***
rule/guard: 'izzum\examples\trafficlight\rules\CanSwitch'
command/action: 'izzum\examples rafficlight\command\SwitchRed'
description: 'from orange to red. use the appropriate command'

red

description: 'stop'
entry action: 'none'
exit action: 'none'

how does that work at Telfort?

tools: process automation



obs(801723): address-occupied

statemachine (sm) + message queue (mq) regular mode

(!) - pas op, dit kan lang duren of heeft performance impact.

machine	entity id
order-cobs	801723

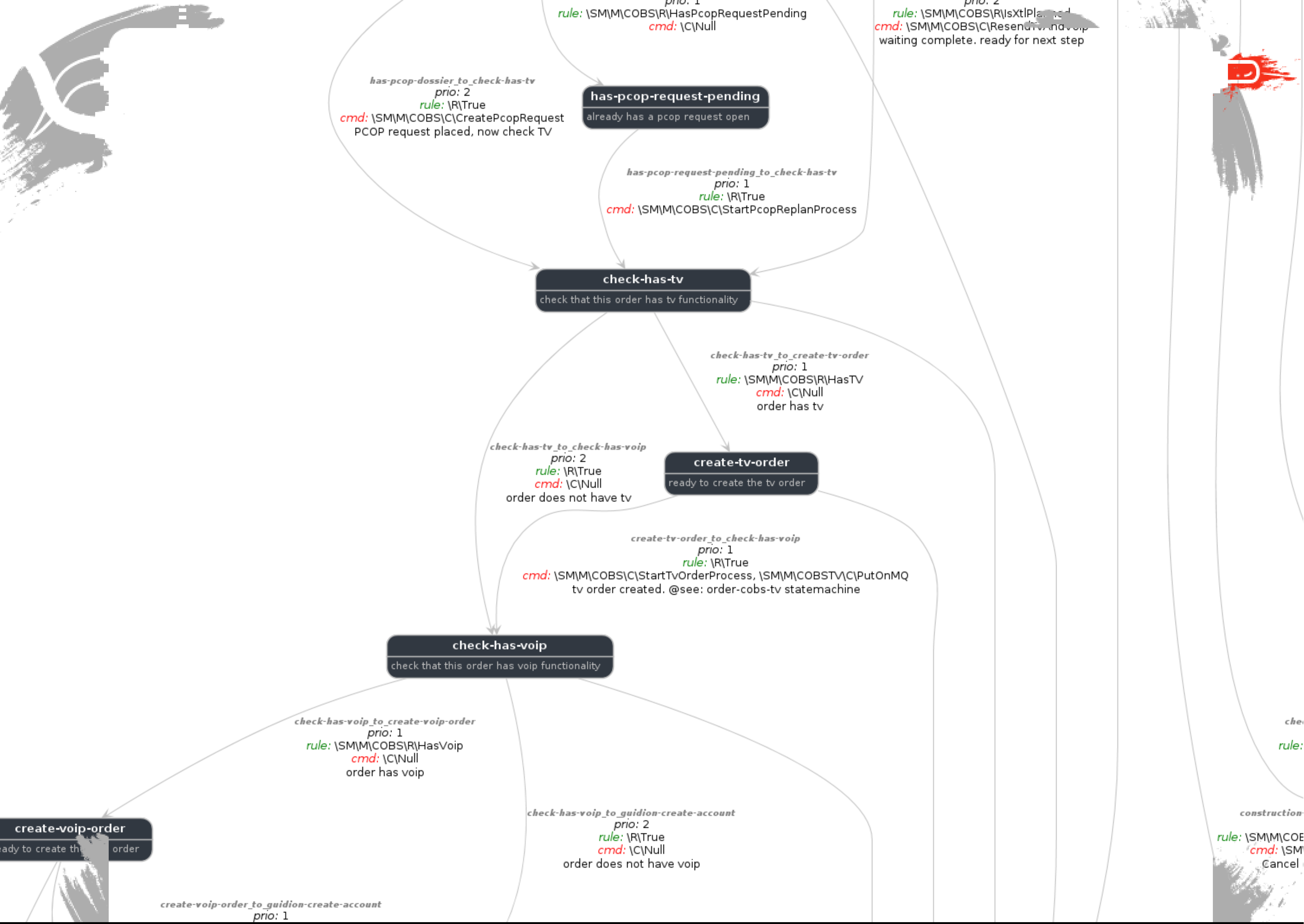
machine	state
order-cobs	address-occupied (21)

machine	entity id
order-cobs	801723

statemachine (sm) diagrams

machine	diagram
order-cobs	state diagram

machine	diagram per entity	entity id
order-cobs	entity flow diagram	801723



prio: 1
rule: \SM\M\COBS\R\HasPcopRequestPending
cmd: \C\Null

prio: 2
rule: \SM\M\COBS\R\IsXtlPlanned
cmd: \SM\M\COBS\C\ResendTvAndVoip
waiting complete. ready for next step

has-pcop-dossier_to_check-has-tv
prio: 2
rule: \R\True
cmd: \SM\M\COBS\C\CreatePcopRequest
PCOP request placed, now check TV

has-pcop-request-pending
already has a pcop request open

has-pcop-request-pending_to_check-has-tv
prio: 1
rule: \R\True
cmd: \SM\M\COBS\C\StartPcopReplanProcess

check-has-tv
check that this order has tv functionality

check-has-tv_to_create-tv-order
prio: 1
rule: \SM\M\COBS\R\HasTV
cmd: \C\Null
order has tv

check-has-tv_to_check-has-voip
prio: 2
rule: \R\True
cmd: \C\Null
order does not have tv

create-tv-order
ready to create the tv order

create-tv-order_to_check-has-voip
prio: 1
rule: \R\True
cmd: \SM\M\COBS\C\StartTvOrderProcess, \SM\M\COBSTV\C\PutOnMQ
tv order created. @see: order-cobs-tv statemachine

check-has-voip
check that this order has voip functionality

check-has-voip_to_create-voip-order
prio: 1
rule: \SM\M\COBS\R\HasVoip
cmd: \C\Null
order has voip

check-has-voip_to_guidion-create-account
prio: 2
rule: \R\True
cmd: \C\Null
order does not have voip

create-voip-order
ady to create th order

create-voip-order_to_guidion-create-account
prio: 1

che
rule:

construction-
rule: \SM\M\COB
cmd: \SM
Cancel

statemachine design patterns

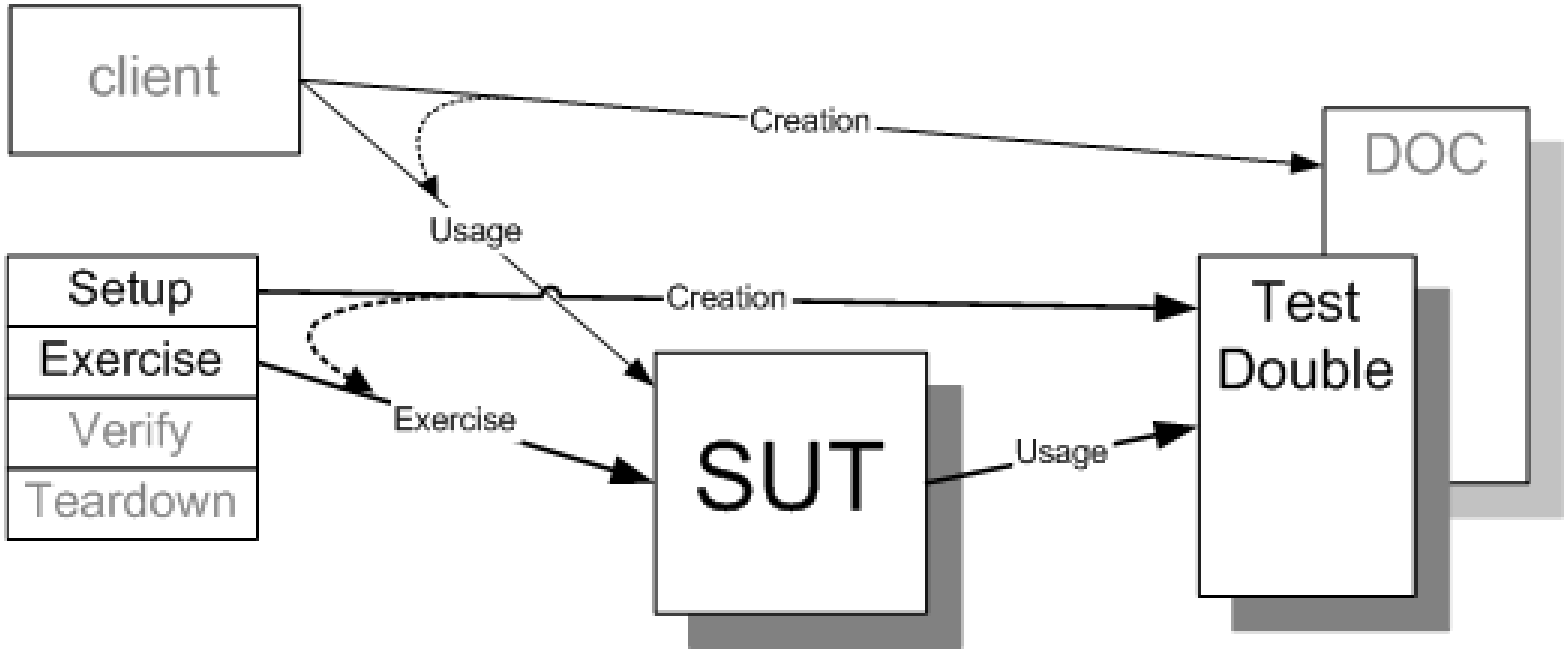
- conditional flow: go to state C from A or from A via B
- linear flow: one way out, mostly used for bookkeeping state
- funnel state: a state that functions as an entry to a final state with potentially many states pointing to it. the state has no logic associated with that flow but functions as a bookkeeping state
- two ways out: don't overcomplicate by only using two outgoing transitions
- self transition: transition to self
- polling state: state that has a rule that polls a third party service
- active state: a state named after the activity it will perform (activity on entry/exit)
- passive state: a state that performs no activity (activity on transition)
- bookkeeping state: does nothing, only records that is has been there

and what about quality control and testing?

unit and component testing

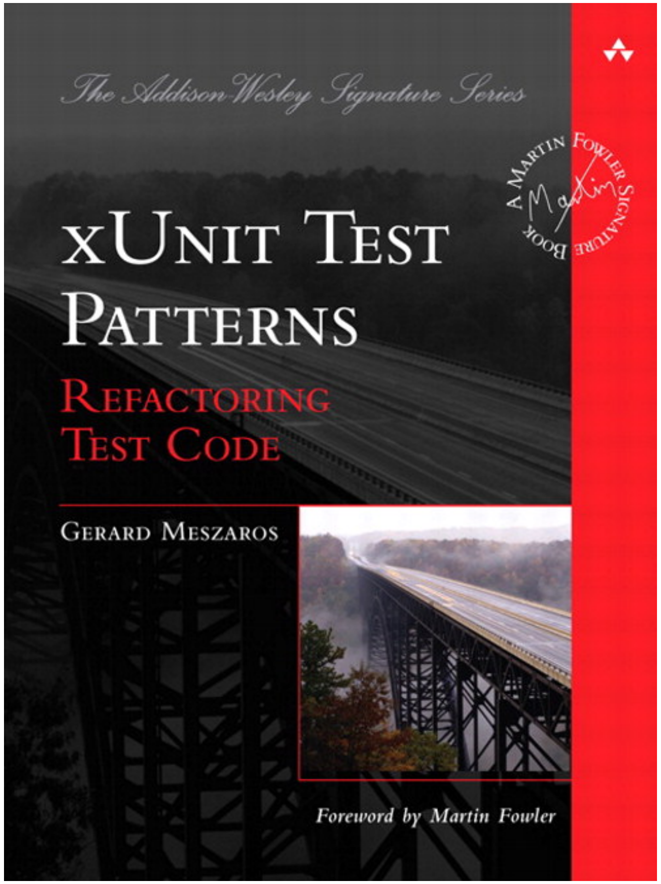
- core statemachine package is tested with high coverage
- tests your specific application code: rules and commands
  - they should do only one thing
  - they make use of tested domain models
  - they can be (component/unit)tested in isolation
  - they can be injected with test doubles as dependencies
    - constructor injection
    - setter injection
  - they are whitebox tested with mocks and stubs

dependency injection

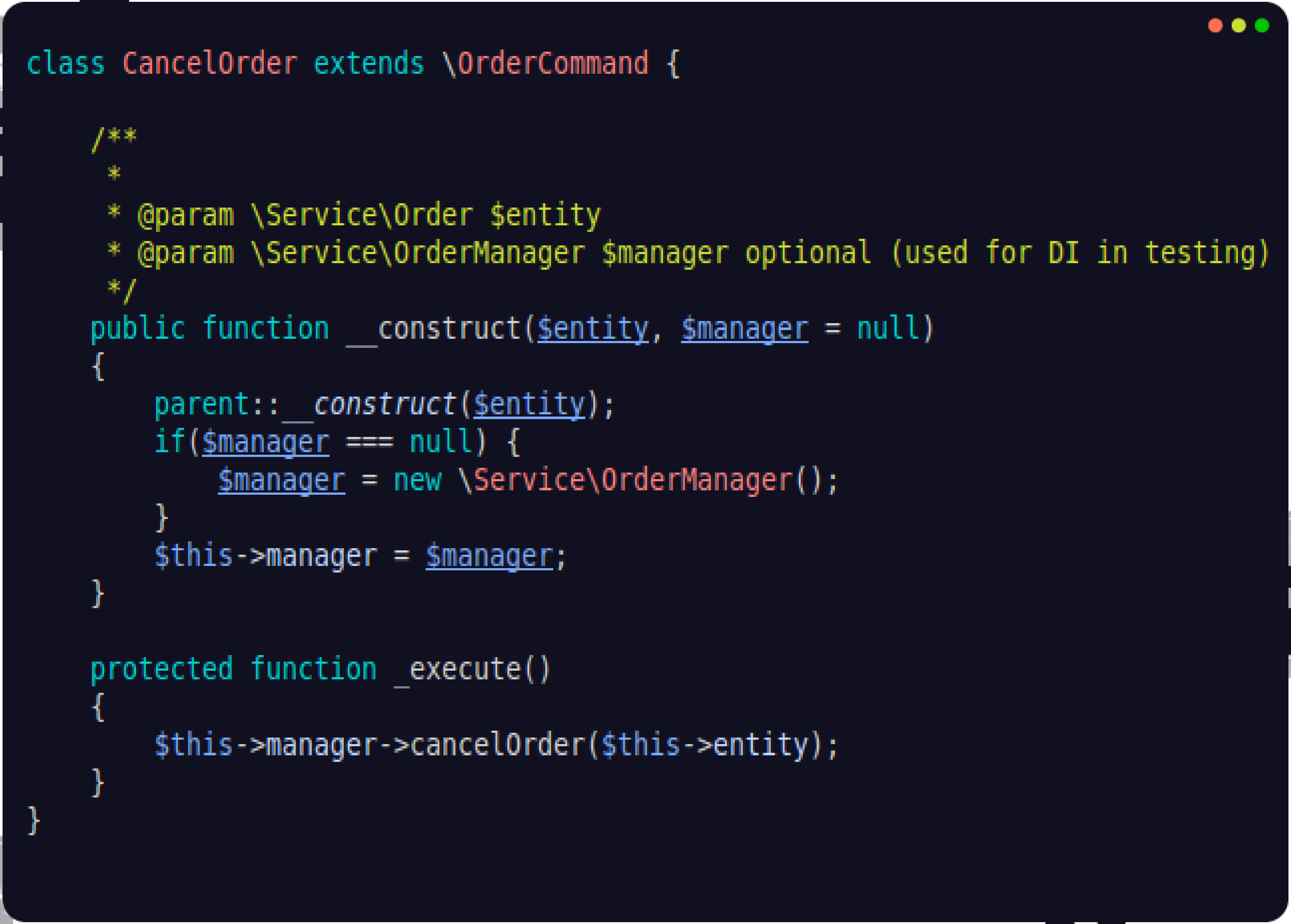


(x)unit test patterns

- http://xunitpatterns.com

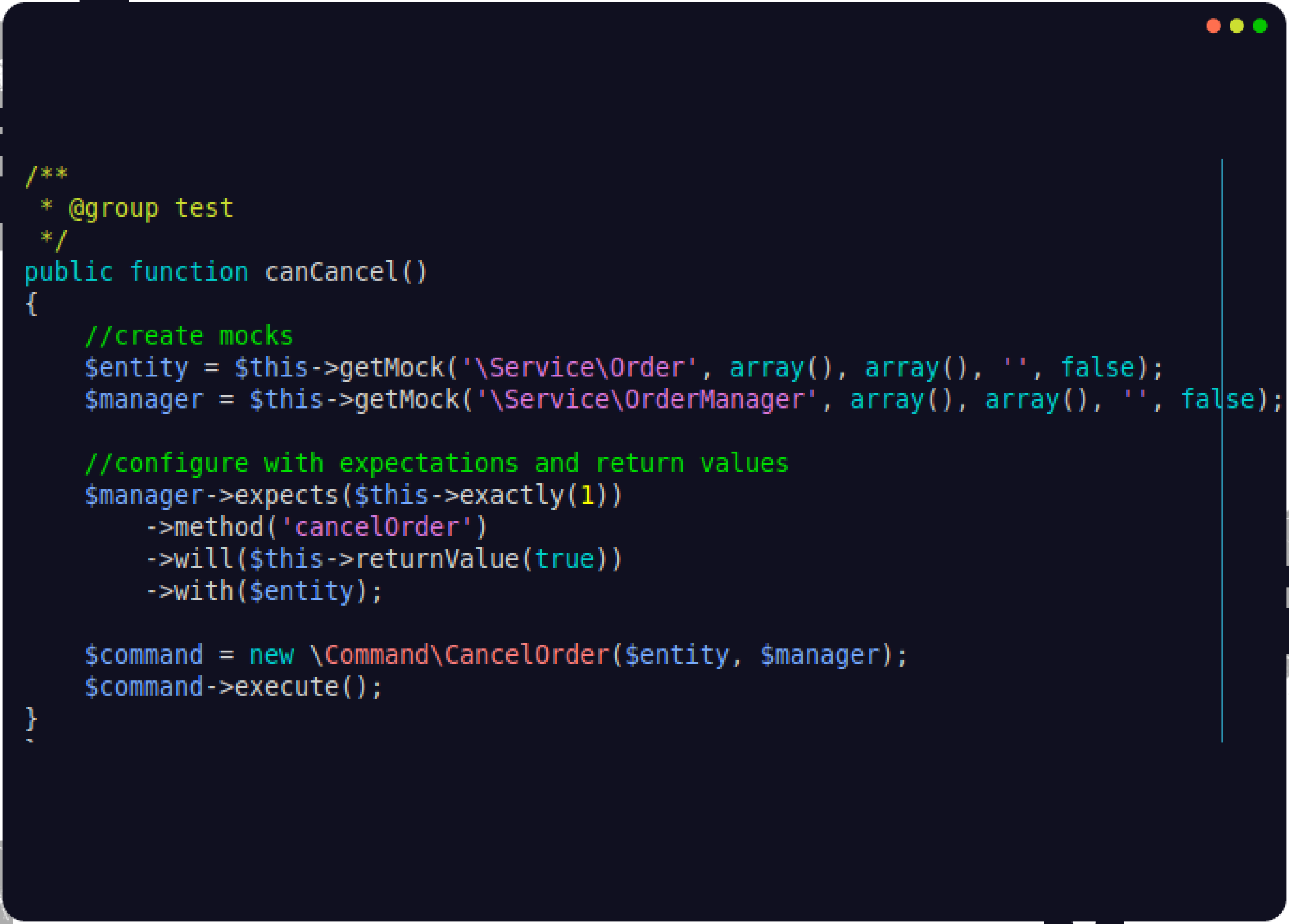

The Addison-Wesley Signature Series
A Martin Fowler Signature Book
xUnit Test Patterns
Refactoring Test Code
Gerard Meszaros
Foreword by Martin Fowler

command



```php
class CancelOrder extends \OrderCommand {

    /**
     *
     * @param \Service\Order $entity
     * @param \Service\OrderManager $manager optional (used for DI in testing)
     */
    public function __construct($entity, $manager = null)
    {
        parent::__construct($entity);
        if($manager === null) {
            $manager = new \Service\OrderManager();
        }
        $this->manager = $manager;
    }

    protected function _execute()
    {
        $this->manager->cancelOrder($this->entity);
    }
}
```

testing a command



```php
/**
 * @group test
 */
public function canCancel()
{
    //create mocks
    $entity = $this->getMock('\Service\Order', array(), array(), '', false);
    $manager = $this->getMock('\Service\OrderManager', array(), array(), '', false);

    //configure with expectations and return values
    $manager->expects($this->exactly(1))
        ->method('cancelOrder')
        ->will($this->returnValue(true))
        ->with($entity);

    $command = new \Command\CancelOrder($entity, $manager);
    $command->execute();
}
```

functional testing

- tooling and diagrams supports testers
  - visualization of flows through statemachines
  - rules and commands can be tested in isolation
  - easily skip to states in statemachines
  - can be used to automate testing (eg: Selenium)
- external services and dependencies
  - are mocked in chain testing
  - are mostly isolated api calls and data handling encapsulated in a command
- failures occur for a transition: 1 command or 1 rule
  - failures occur in isolation and are relatively easy to debug
  - base command/rule classes catches exceptions with the correct info from the dependent upon component

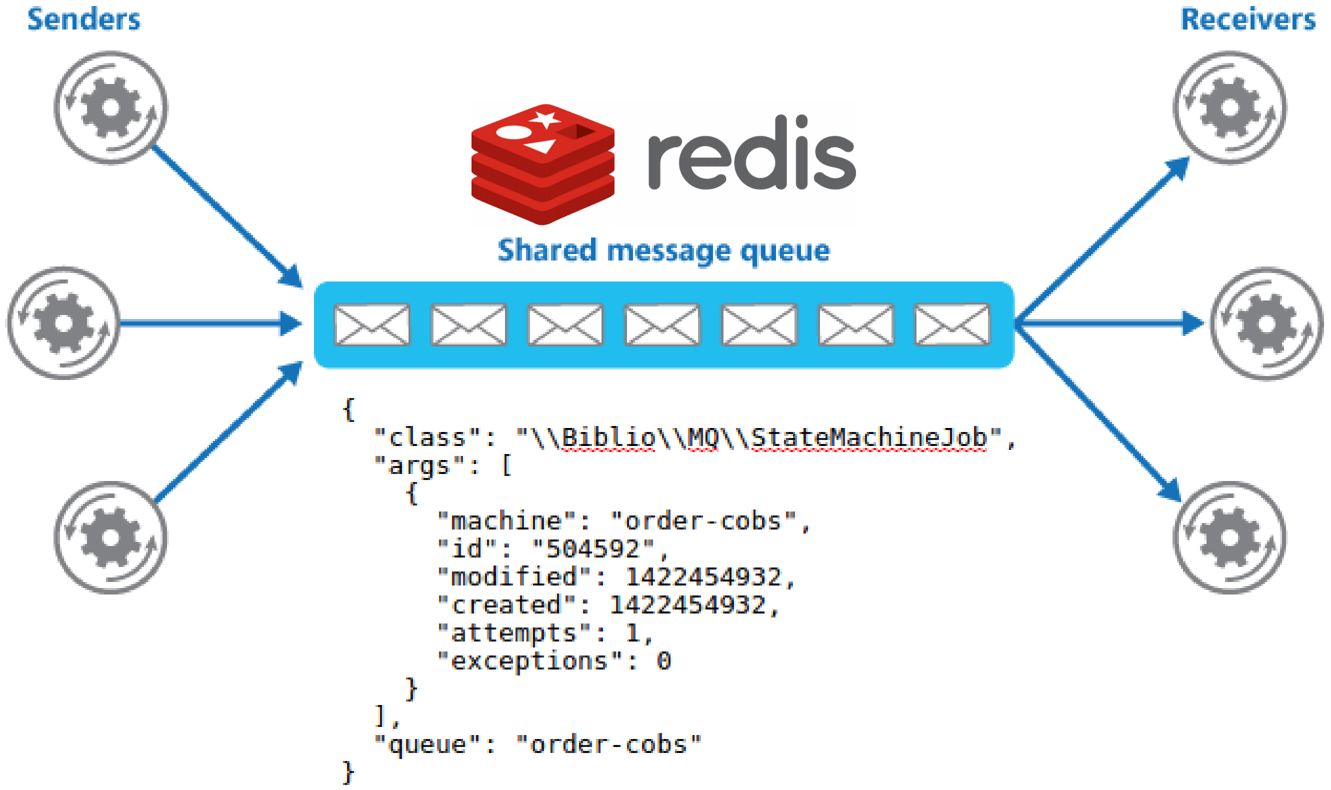
that's all good, but does it perform?

solving scalability

- *because:*
  - statemachines are identified by their {name, id}
    - these two pieces of information allow a factory to create the statemachine
    - state is preserved between processes
- *it is:*
  - easy to transmit the statemachine information in a message
- *and:*
  - we can have a message queue handle messages asynchronously and scale horizontally

redis as a message queue

- redis serves as a transient data store for process data



```
{
  "class": "\\Biblio\\MQ\\StateMachineJob",
  "args": [
    {
      "machine": "order-cobs",
      "id": "504592",
      "modified": 1422454932,
      "created": 1422454932,
      "attempts": 1,
      "exceptions": 0
    }
  ],
  "queue": "order-cobs"
}
```

machine and message queue

- seperating mechanism and policy
- statemachine *(dis)allows transitions and logic according to rules* (policy)
- message queue jobs *direct the statemachine* (mechanism)
  - *directing a statemachine can also be done via cronjobs, gui tools, application code etc.*

some numbers

- 13 statemachines for different processes handling about 100.000 customers
- 5 million transitions executed
- 21 million messages for statemachines processed
- 0.1% of those 21 million failed to transition because of exceptions *(bugs + 3d party dependencies)*
- new statemachine processes will handle over 500.000 customers

almost there, wrapping it up....

benefits of using the statemachine

- consistent and understandable behaviour for development teams
- logic is isolated in reusable rules and commands
- great process overview via uml generation
- facilitates unittesting via the implementation of rules and commands and seperating the domain models from the statemachine
- using statemachines scales well via message queue
- provides statistics via transition history
- there is good tooling to support users throughout the organisation
- new processes can be designed up front and implementation are easier by just coding the appropiate rules and commands
- the organisation understands statemachines so we can use the concept in our discussion of processes

that's all, thanks!

? questions ?

maybe (?) some time for a demo ...

graphics: **boudewijndanser.nl**

github: **rolfvreijdenberger/izzum**

contributions are welcome!