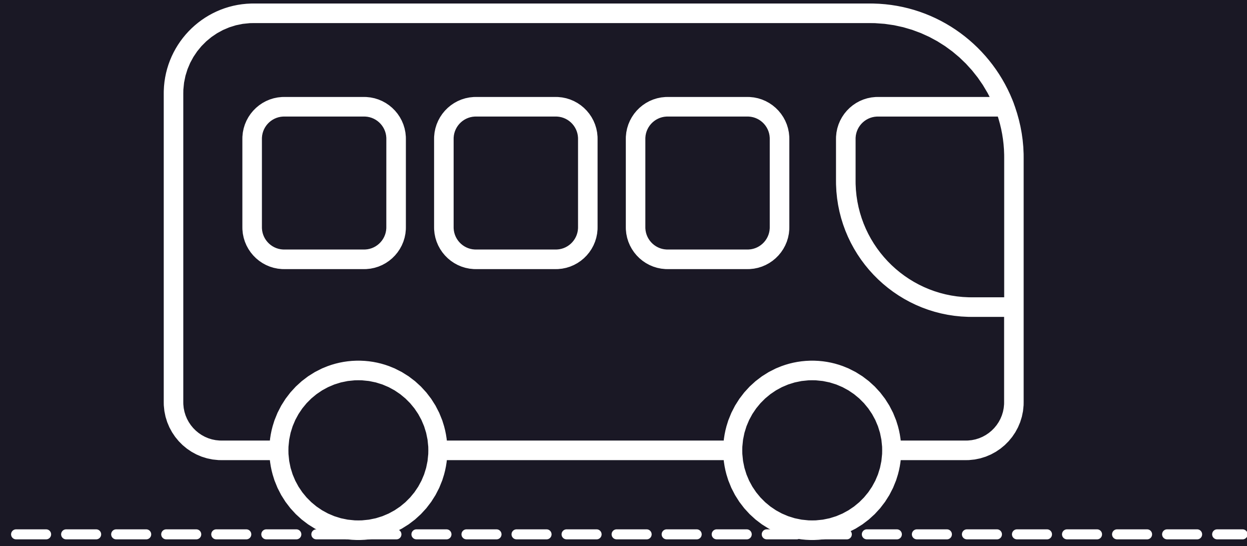
Extending Solidus with the new Event Bus



more flexibility for the topmost flexible e-commerce platform

Marc Busqué - @waiting-for-dev

Marc Busqué Pérez

nebulab

solidus

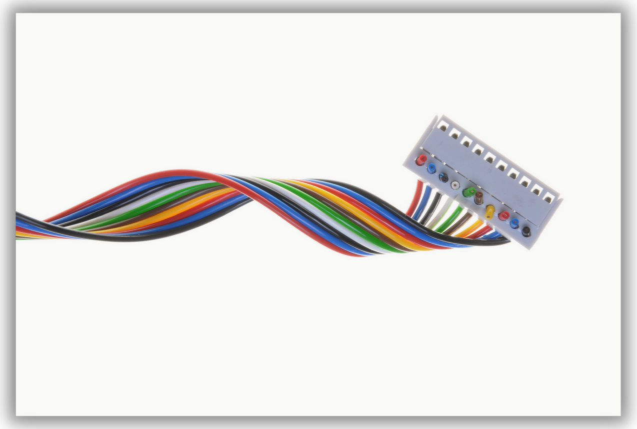
hanami dry-rb

@waiting-for-dev

@waiting_for_dev

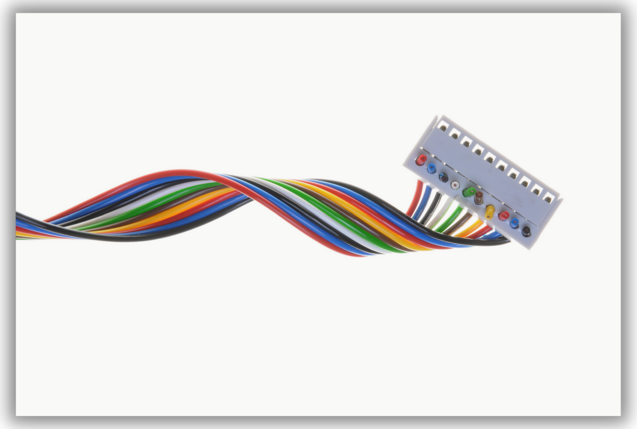
Event
BUS

Event
BUS



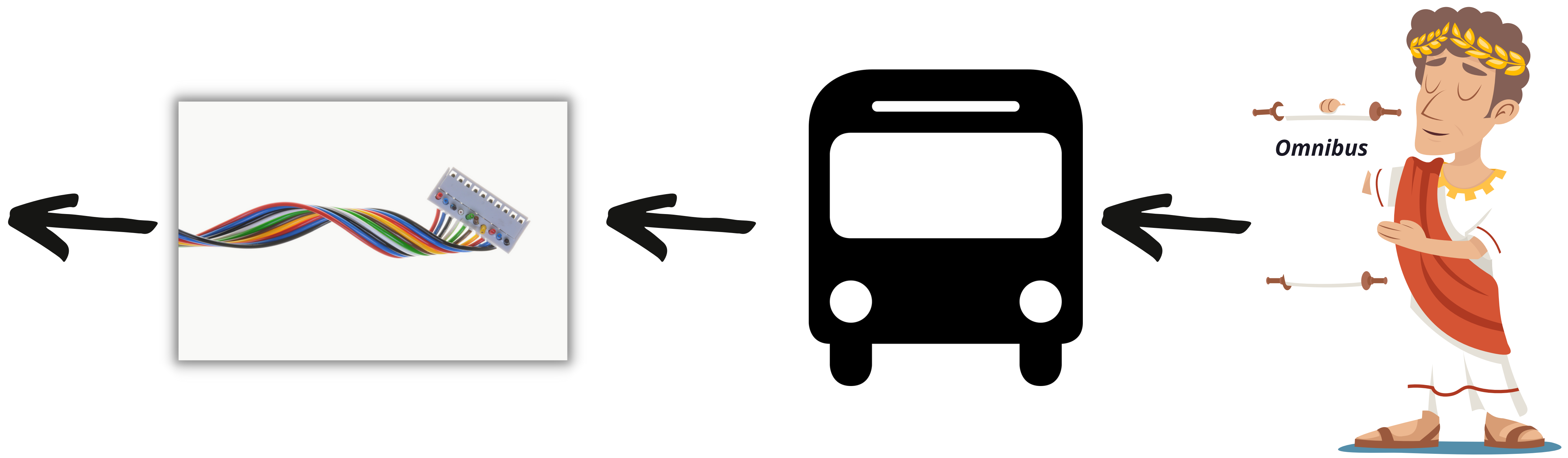
Event

BUS



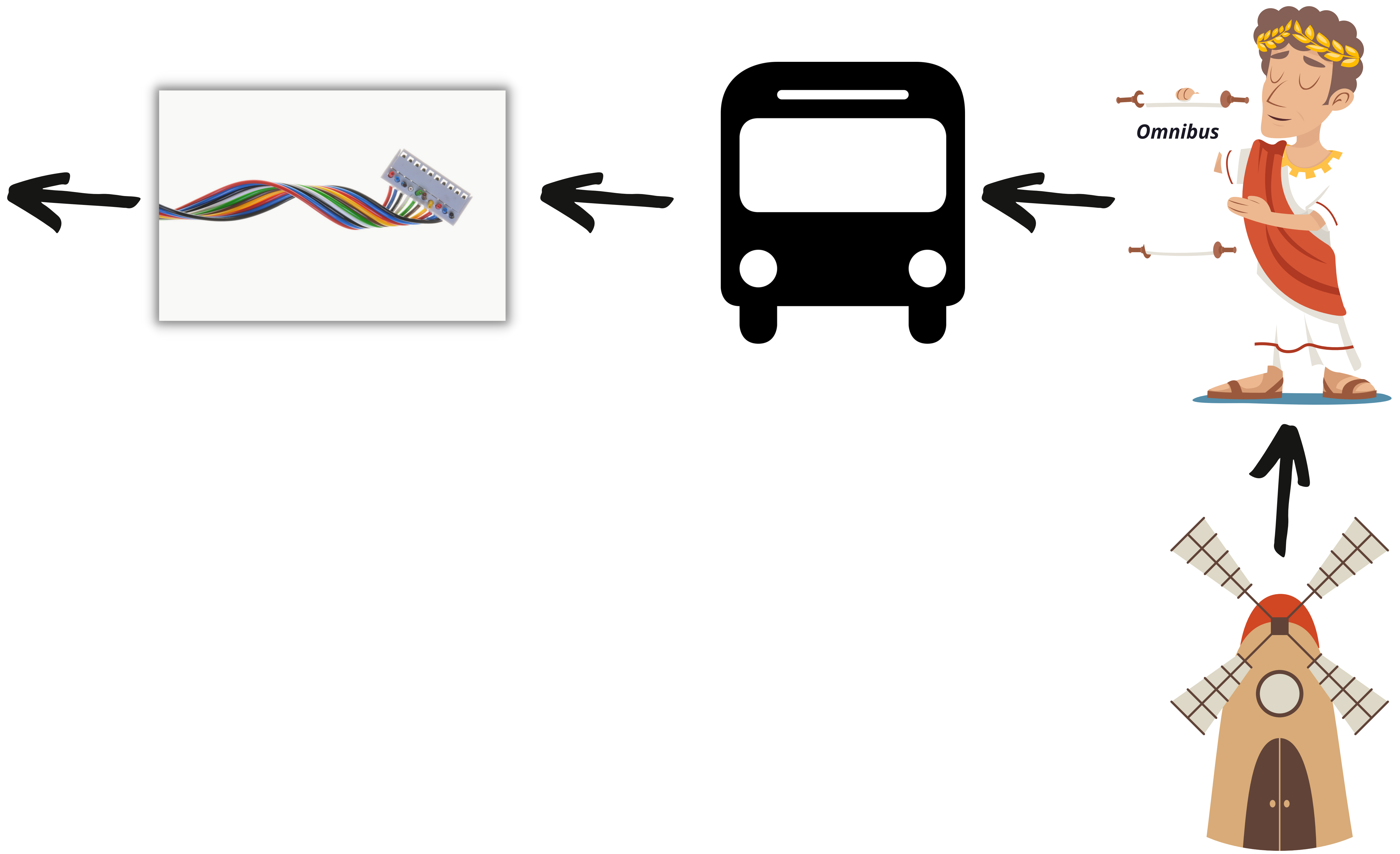
Event

BUS

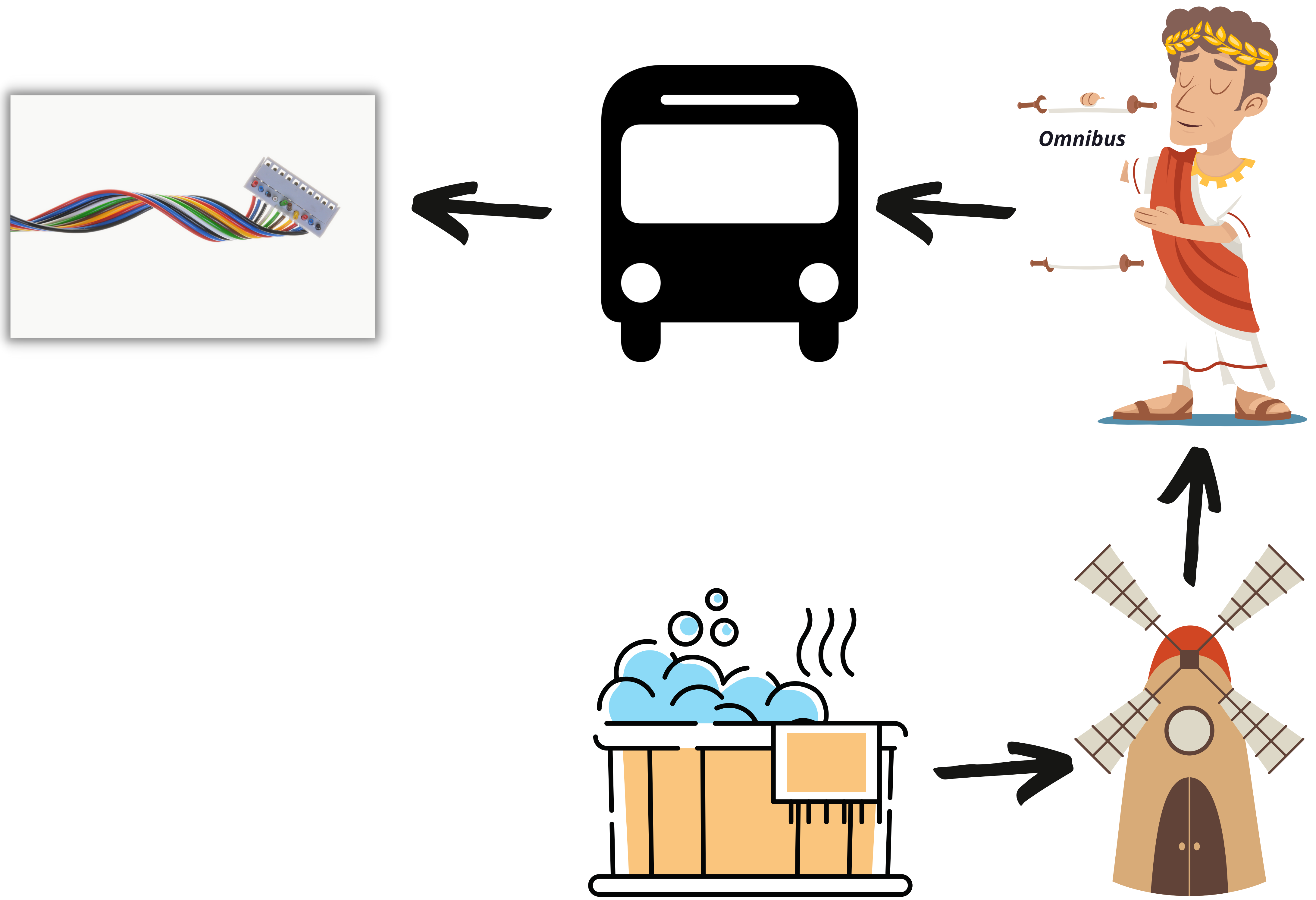


Event

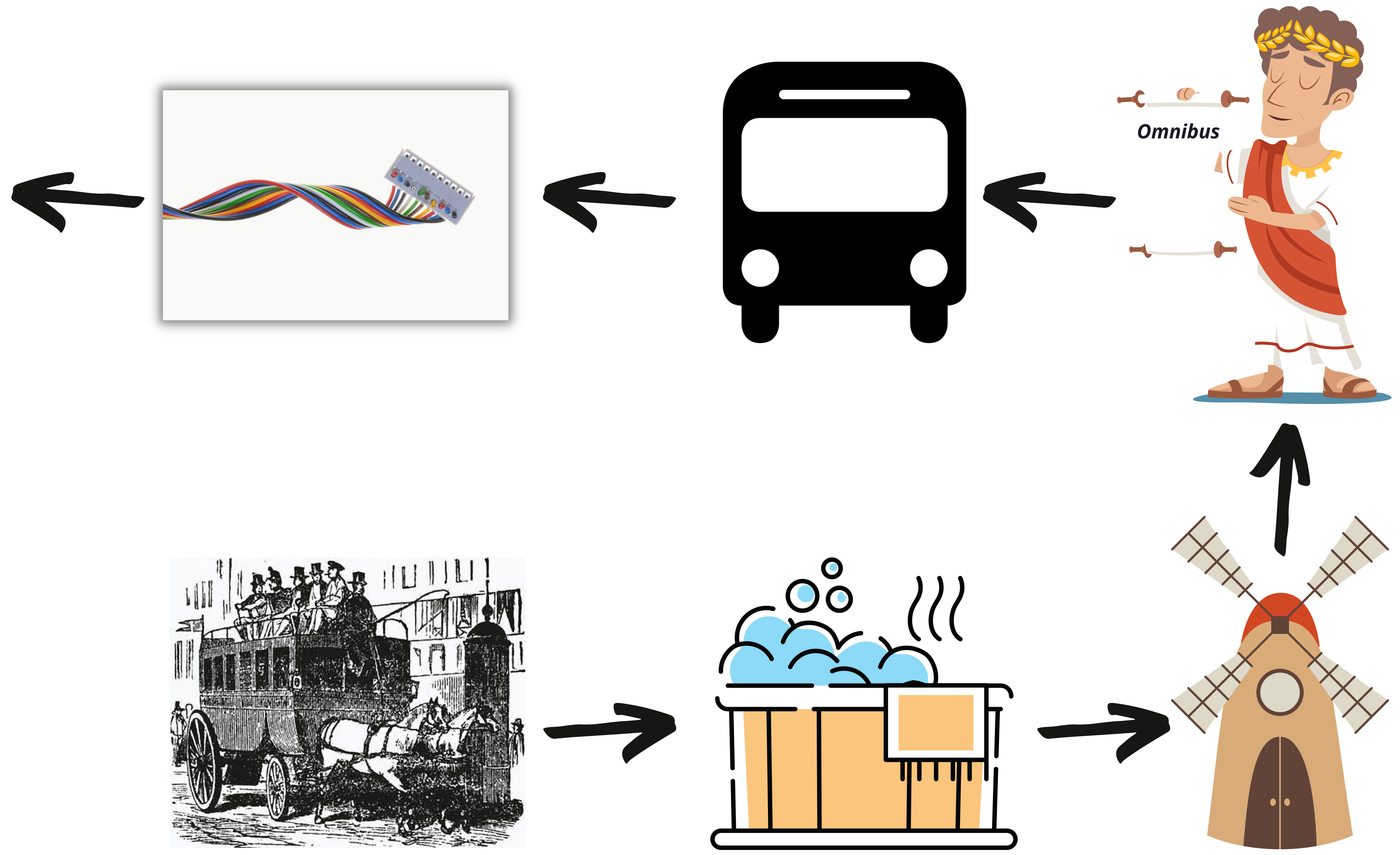
BUS

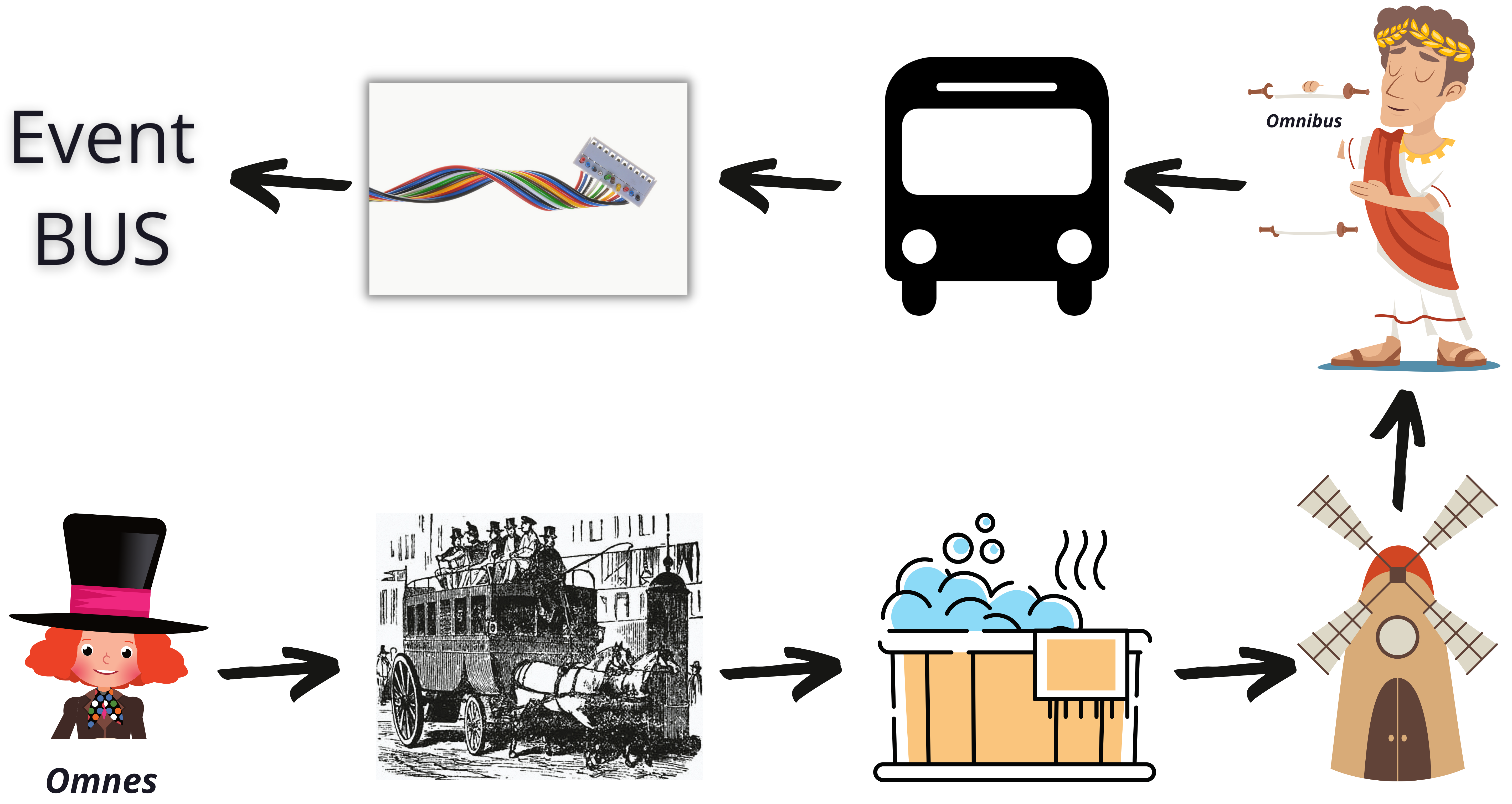


Event
BUS



Event BUS




Event
BUS
Omnibus
Omnes

1. Extending Solidus
2. Event notification
3. The Event Bus on Solidus

① EXTENGING SOLIDUS

1.1. Custom service classes
1.2. Monkey Patching
1.3. The Event Bus

1.1. Custom Service Classes

- Parts of the business logic are encapsulated in classes.
- We can replace those classes with custom ones.

E.g.: Only allow adding one item for each variant

```ruby
# config/initializers/spree.rb
Spree.config do |config|
  config.order_contents_class = 'MyStore::OrderContents'
  # ...
end
```

```ruby
# app/services/my_store/order_contents.rb
module MyStore
  class OrderContents < Spree::OrderContents
    def add(variant, _quantity = 1, options = {})
      return if order.contains?(variant)

      super(variant, 1, options)
    end
  end
end
```

1.2. Monkey patching

- Ruby allows us to reopen classes at any time.
- We can leverage that to override or modify the default behavior.

Module#prepend

Module#include

Module#class_eval

ActiveSupport::Concern

Direct reopening

E.g.: Require a minimum amount to checkout.

```ruby
# config/application.rb
module MyStore
  class Application < Rails::Application
    # ...
    overrides = "#{Rails.root}/app/overrides"
    Rails.autoloaders.main.ignore(overrides)
    config.to_prepare do
      Dir.glob("#{overrides}/**/*.rb").each do |override|
        load override
      end
    end
  end
end
```

```ruby
# app/overrides/my_store/spree/order/require_min_checkout.rb
module MyStore
  module Spree
    module Order
      module RequireMinCheckout
        def checkout_allowed?
          total > 30 && super
        end

        ::Spree::Order.prepend(self)
      end
    end
  end
end
```

1.2. Monkey patching

- Ruby allows us to reopen classes at any time.
- We can leverage that to override or modify the default behavior.

Not an extensibility pattern
Duct-tape solution
Break upgrades

E.g.: Require a minimum amount to checkout.

```ruby
# config/application.rb
module MyStore
  class Application < Rails::Application
    # ...
    overrides = "#{Rails.root}/app/overrides"
    Rails.autoloaders.main.ignore(overrides)
    config.to_prepare do
      Dir.glob("#{overrides}/**/*.rb").each do |override|
        load override
      end
    end
  end
end
```

```ruby
# app/overrides/my_store/spree/order/require_min_checkout.rb
module MyStore
  module Spree
    module Order
      module RequireMinCheckout
        def checkout_allowed?
          total > 30 && super
        end

        ::Spree::Order.prepend(self)
      end
    end
  end
end
```

1.3. The Event Bus

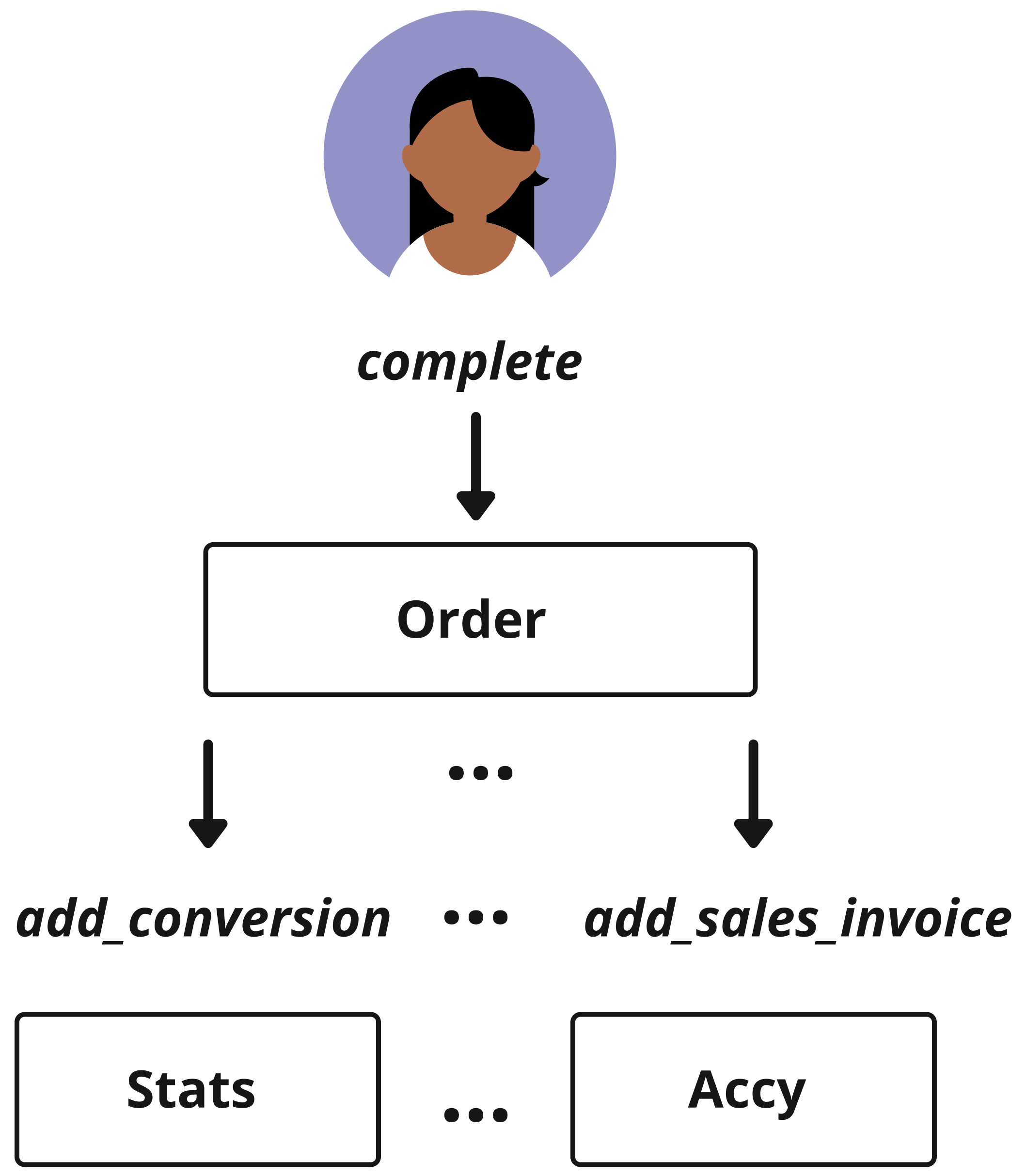
- Pub/Sub pattern.
- It allows us to extend behavior that is independent to the core domain model.

E.g.: Send an SMS to the user when an order is placed.

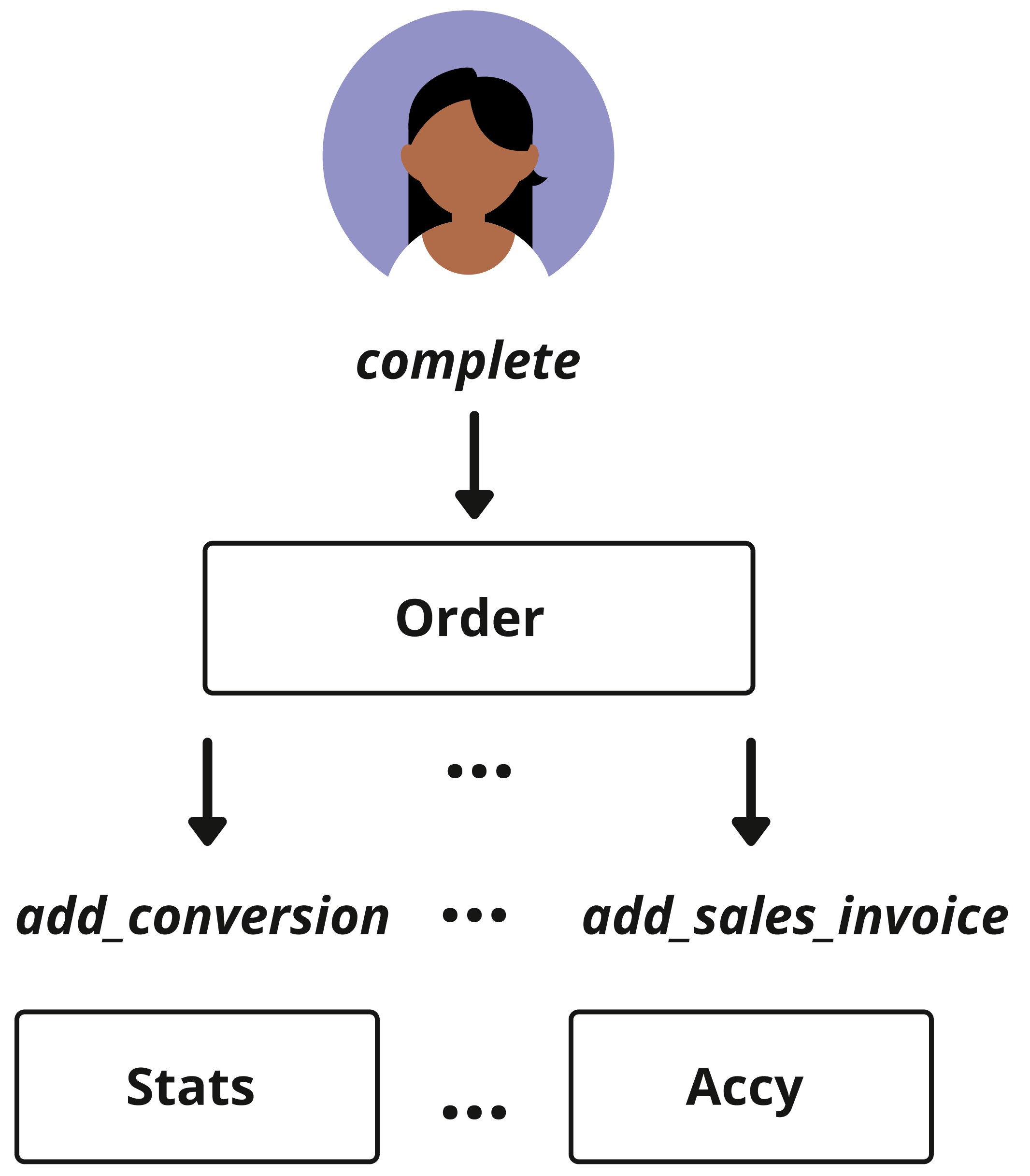
```ruby
# app/subscribers/my_store/sms_notification_subscriber.rb
module MyStore
  module SmsNotificationSubscriber
    include Spree::Event::Subscriber

    event_action :notify_order_completed, event_name: :order_finalized

    def notify_order_completed(event)
      order = event.payload[:order]
      SmsService.new.notify_order_completed(order)
    end
  end
end
```

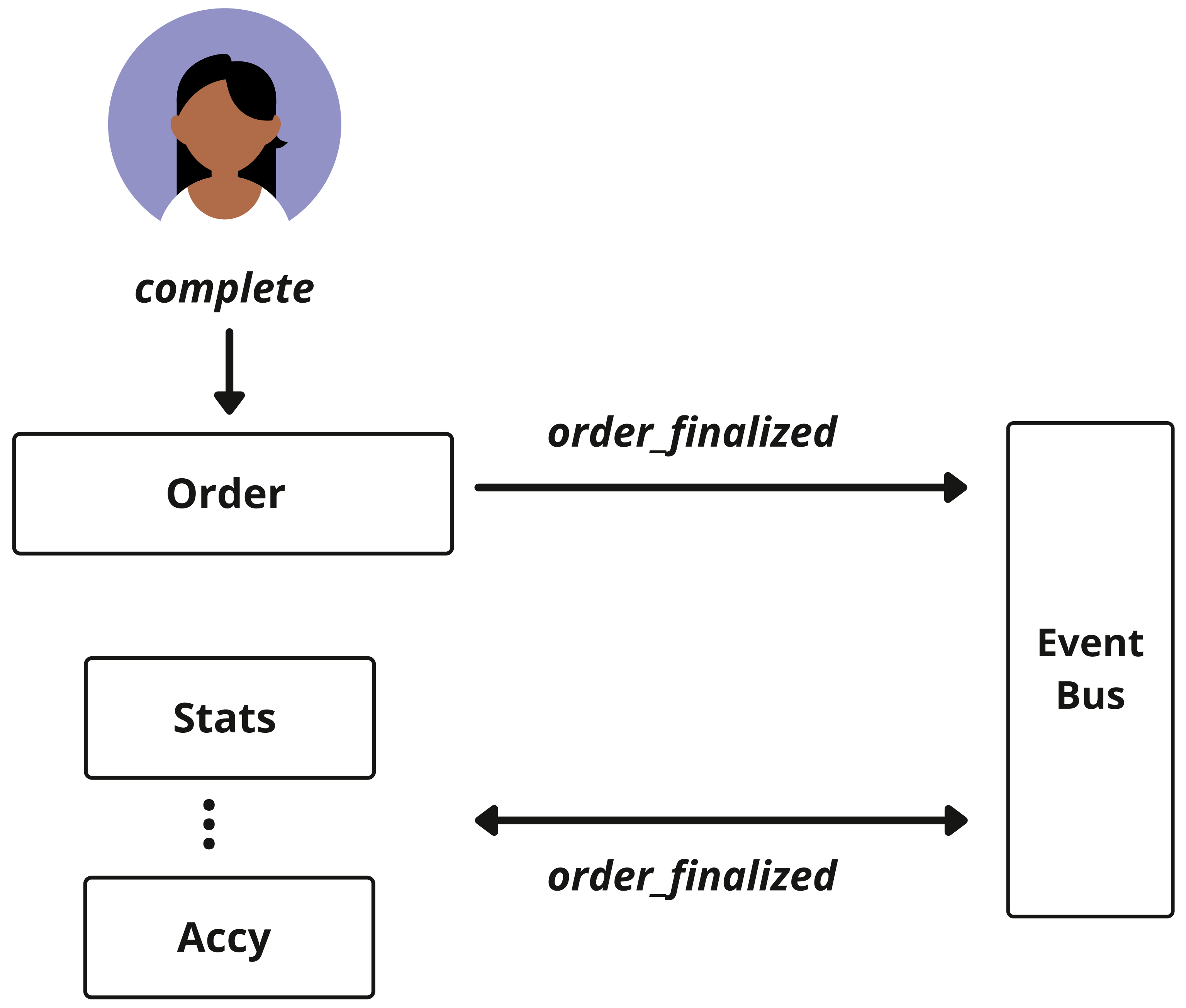
2 EVENT NOTIFICATION

2.1. What is Event Notification?



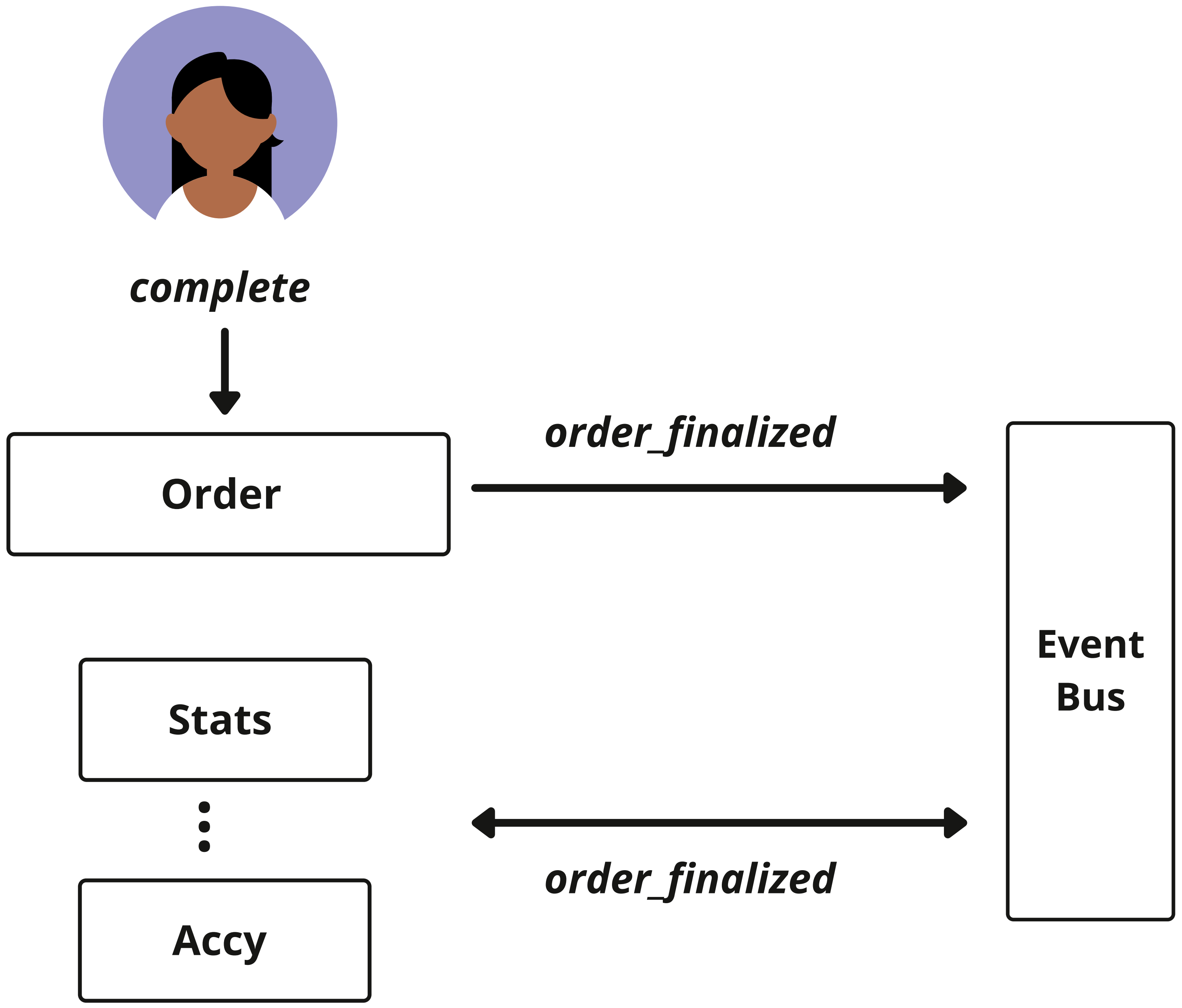
2.1. What is Event Notification?





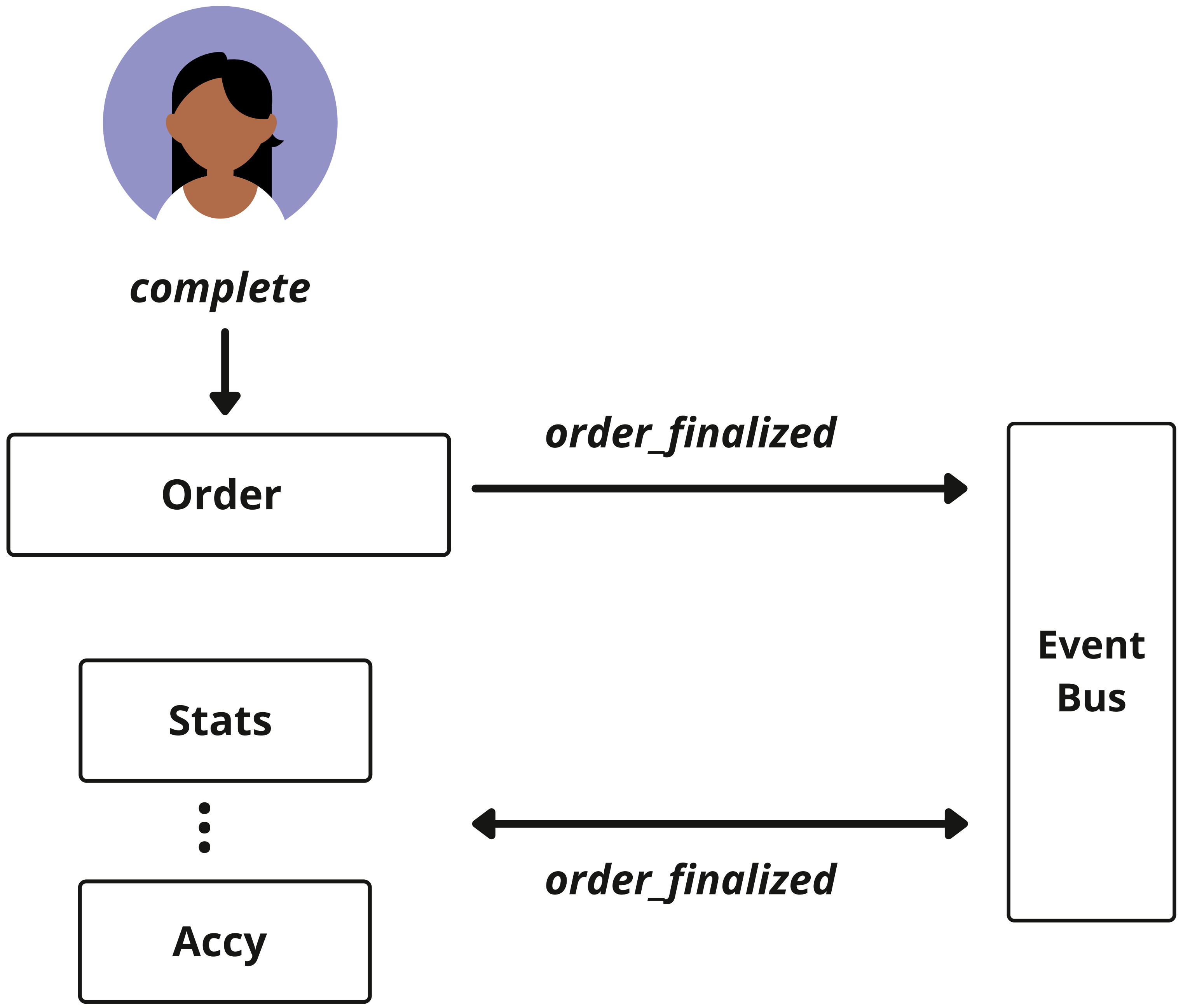
2.2. Pros

- Decoupling (dependency inversion).



2.2. Pros

- Decoupling (dependency inversion).
- Event storage.

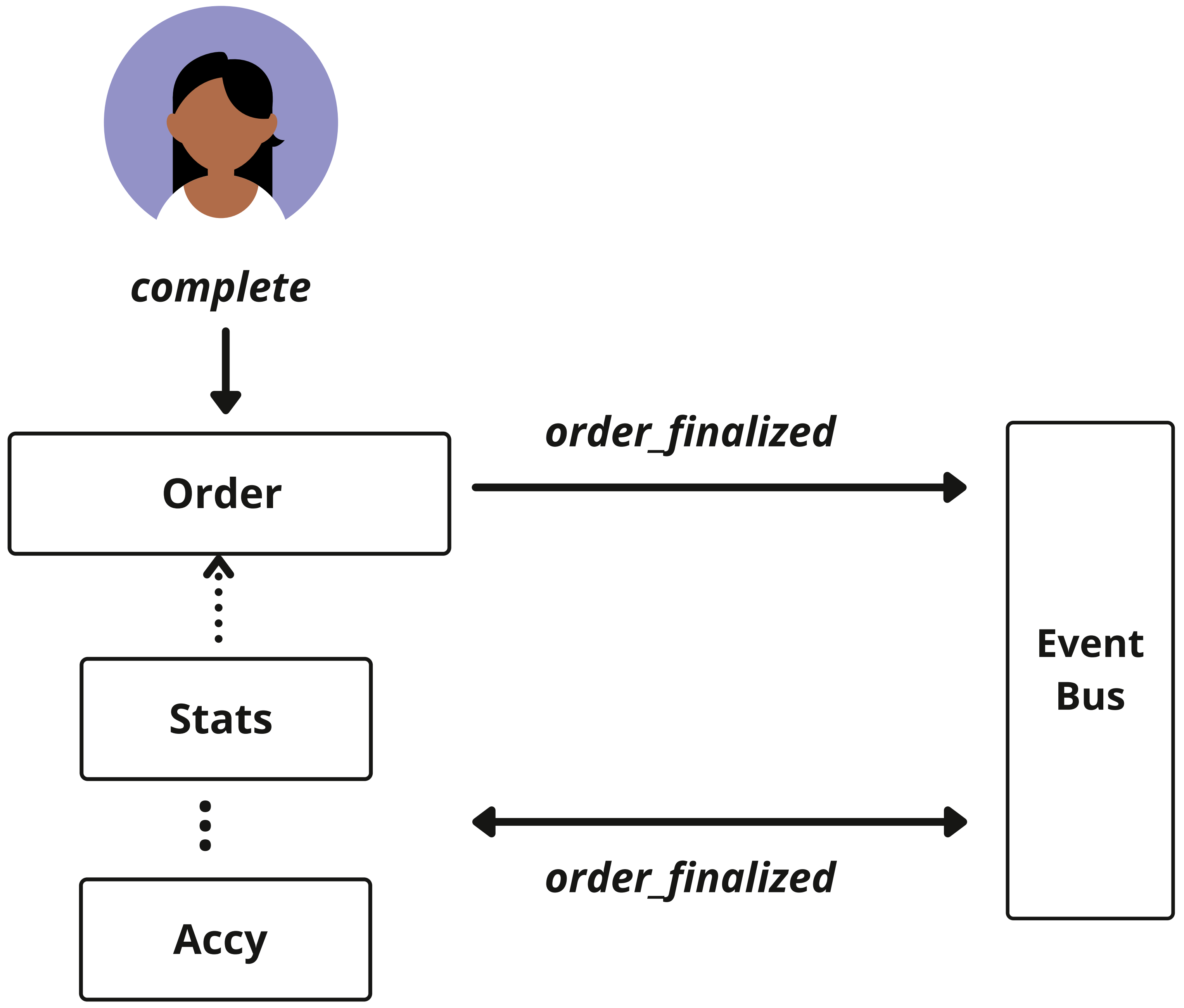


2.2. Pros

- Decoupling (dependency inversion).
- Event storage.

How much upstream data do we publish?

- Few: subscribers need to query back.

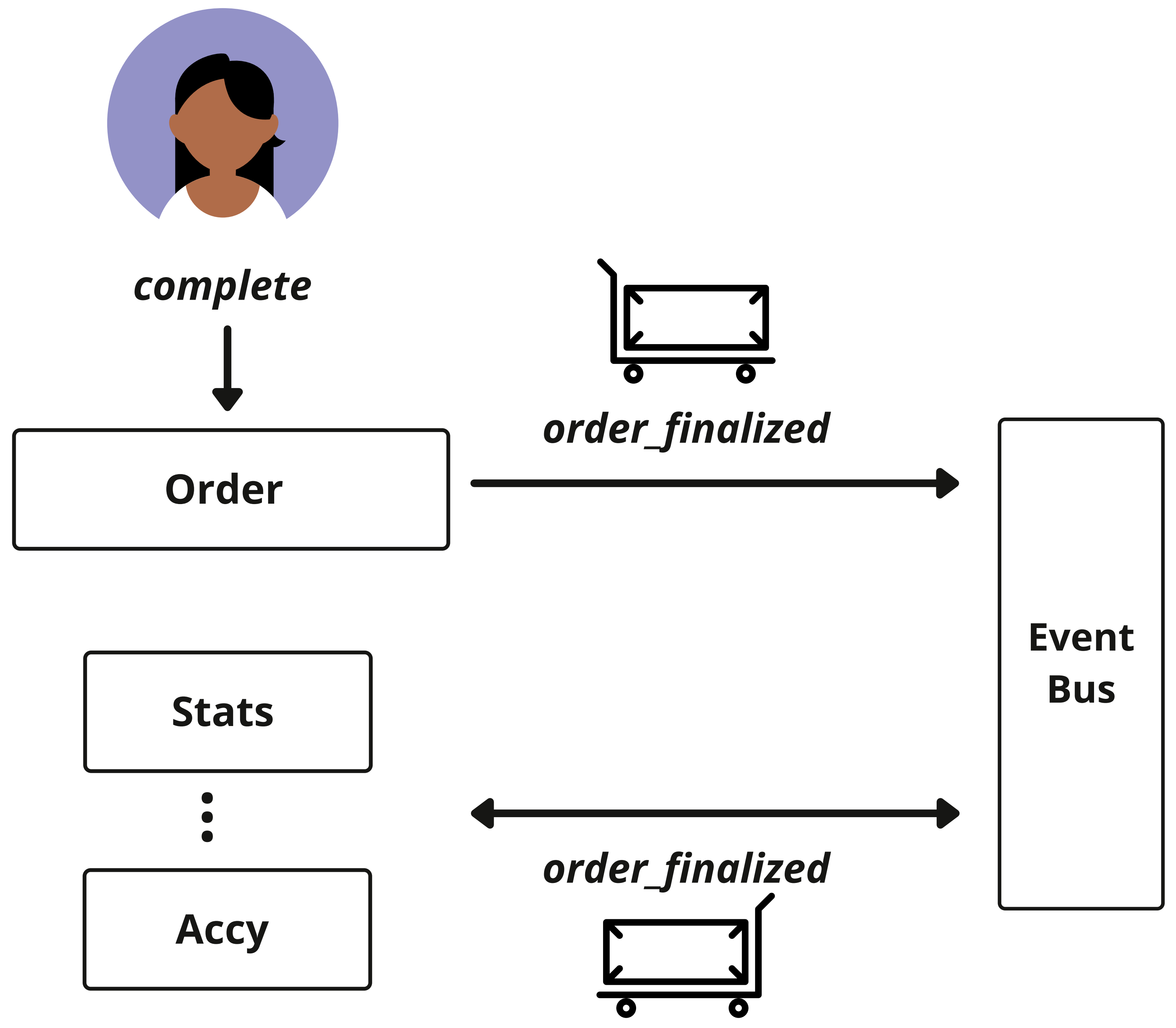


2.2. Pros

- Decoupling (dependency inversion).
- Event storage.

How much upstream data do we publish?

- Few: subscribers need to query back.
- All: the state is transferred.

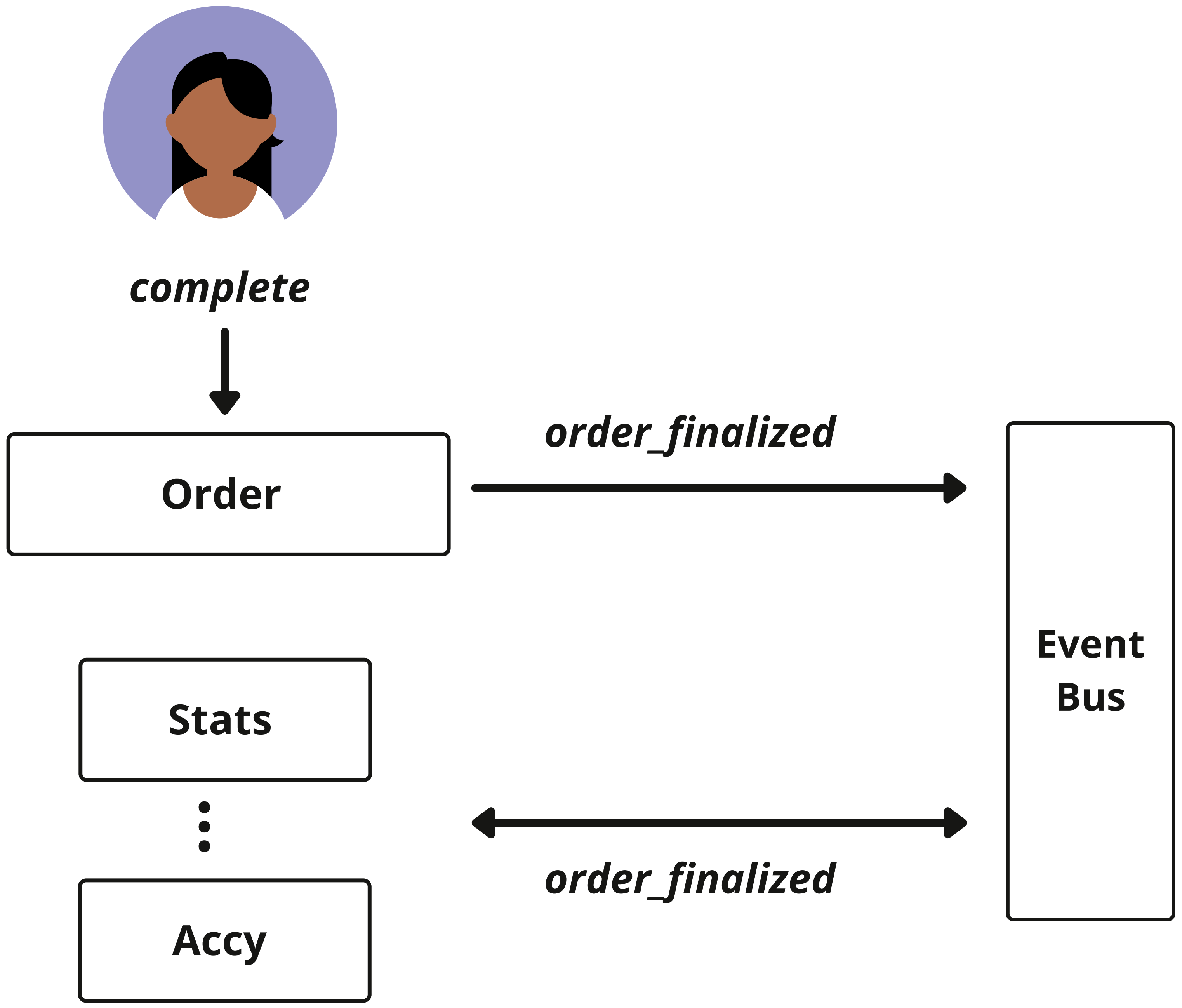


2.2. Pros

- Decoupling (dependency inversion).
- Event storage.

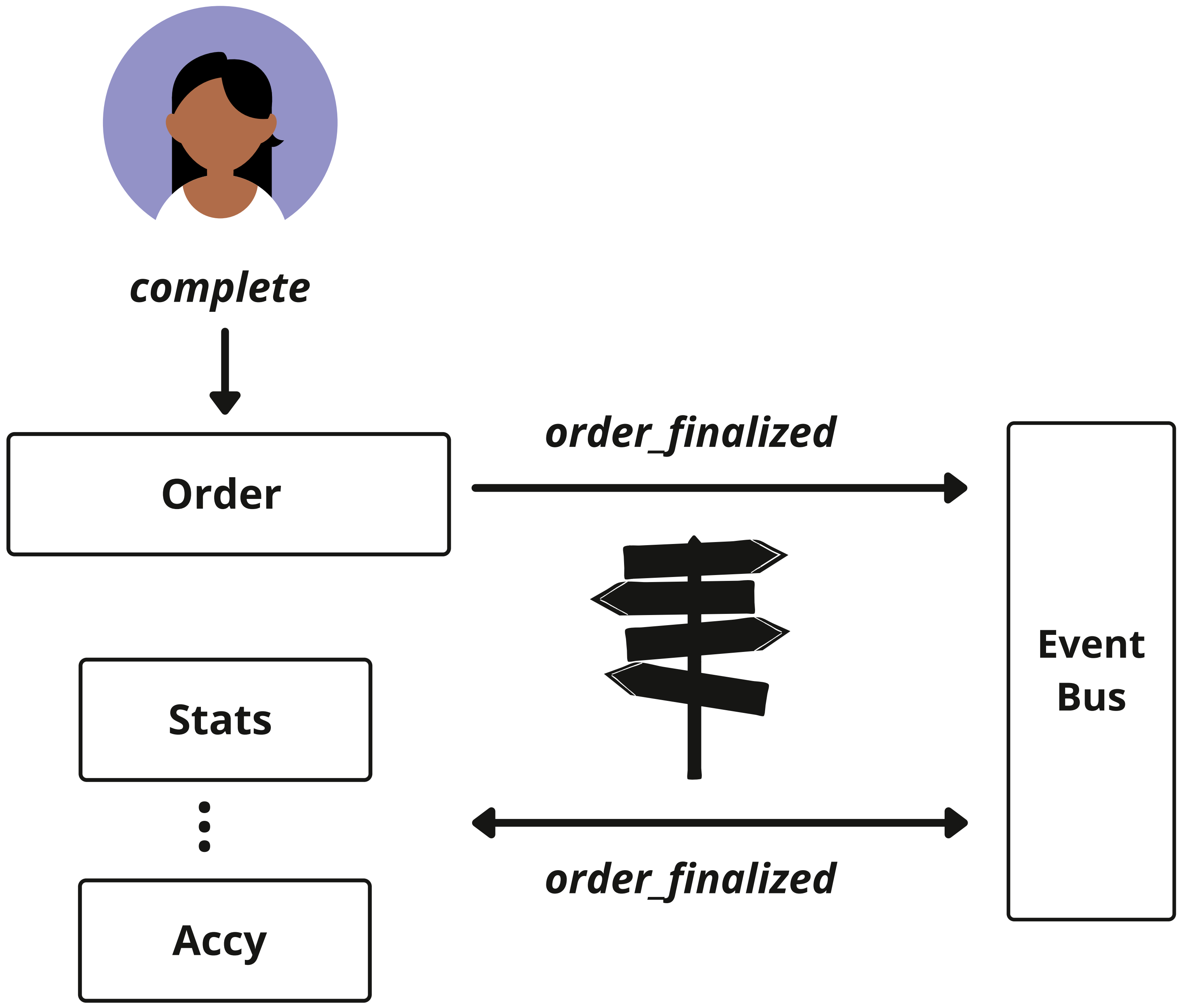
How much upstream data do we publish?

- Few: subscribers need to query back.
- All: the state is transferred.
- Change: event sourcing (reproducible state).



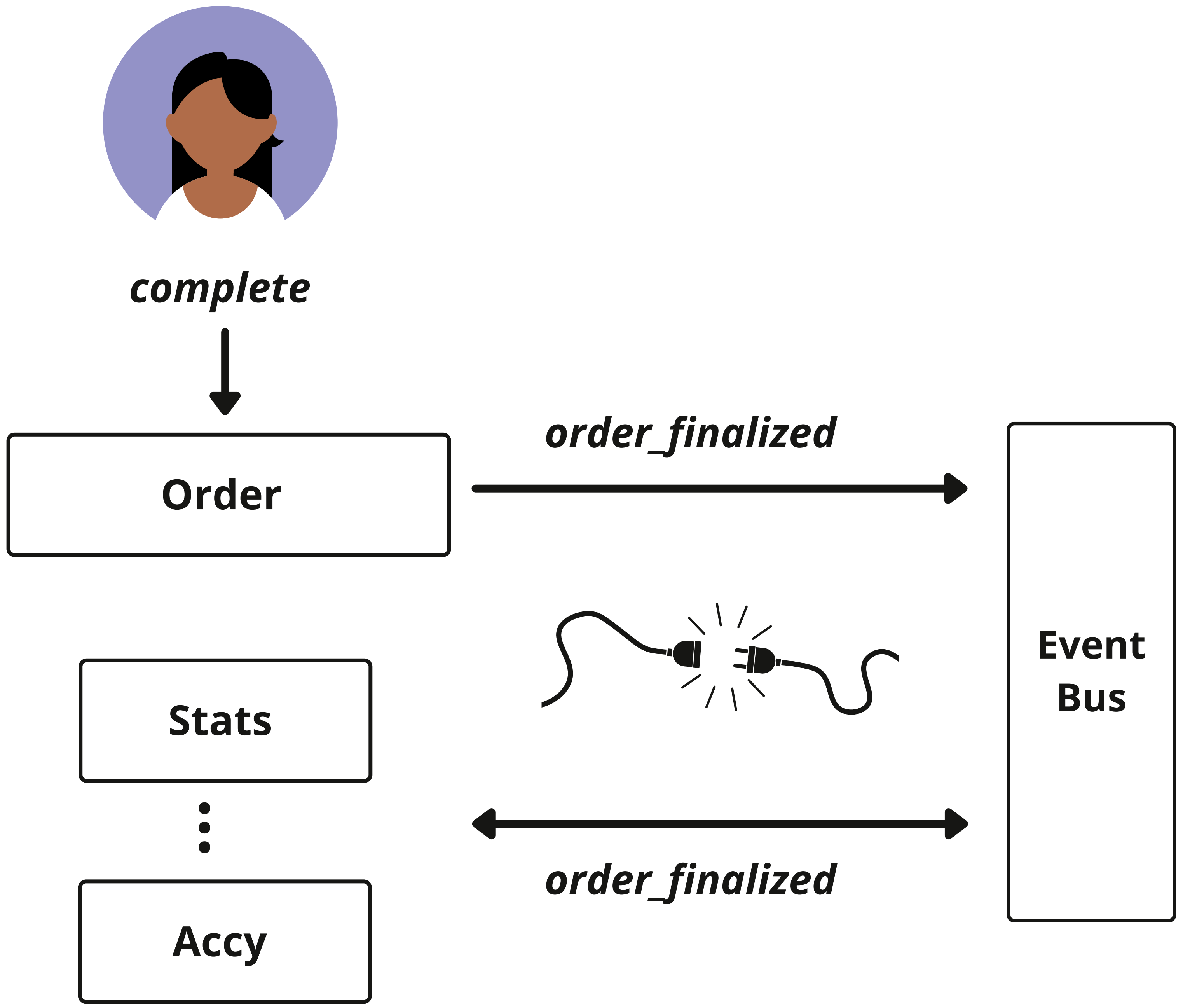
2.2. Cons

- Indirection:
  - Non-linear narrative.
  - Obervability.
  - Testing.



2.2. Cons

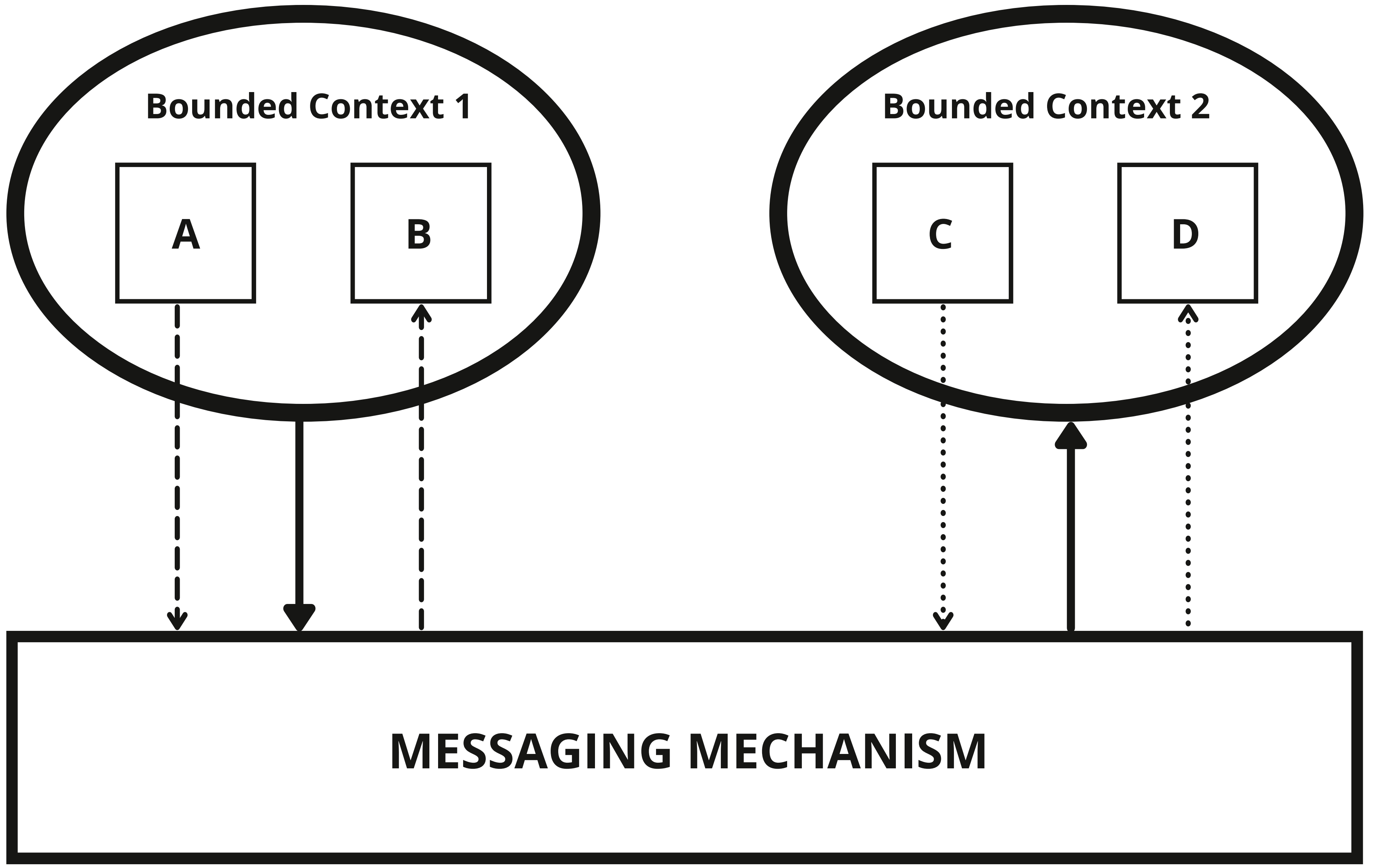
- Indirection:
  - Non-linear narrative.
  - Obervability.
  - Testing.
- Message delivery issues.



2.3. When to use

- Communication between different transactional boundaries.

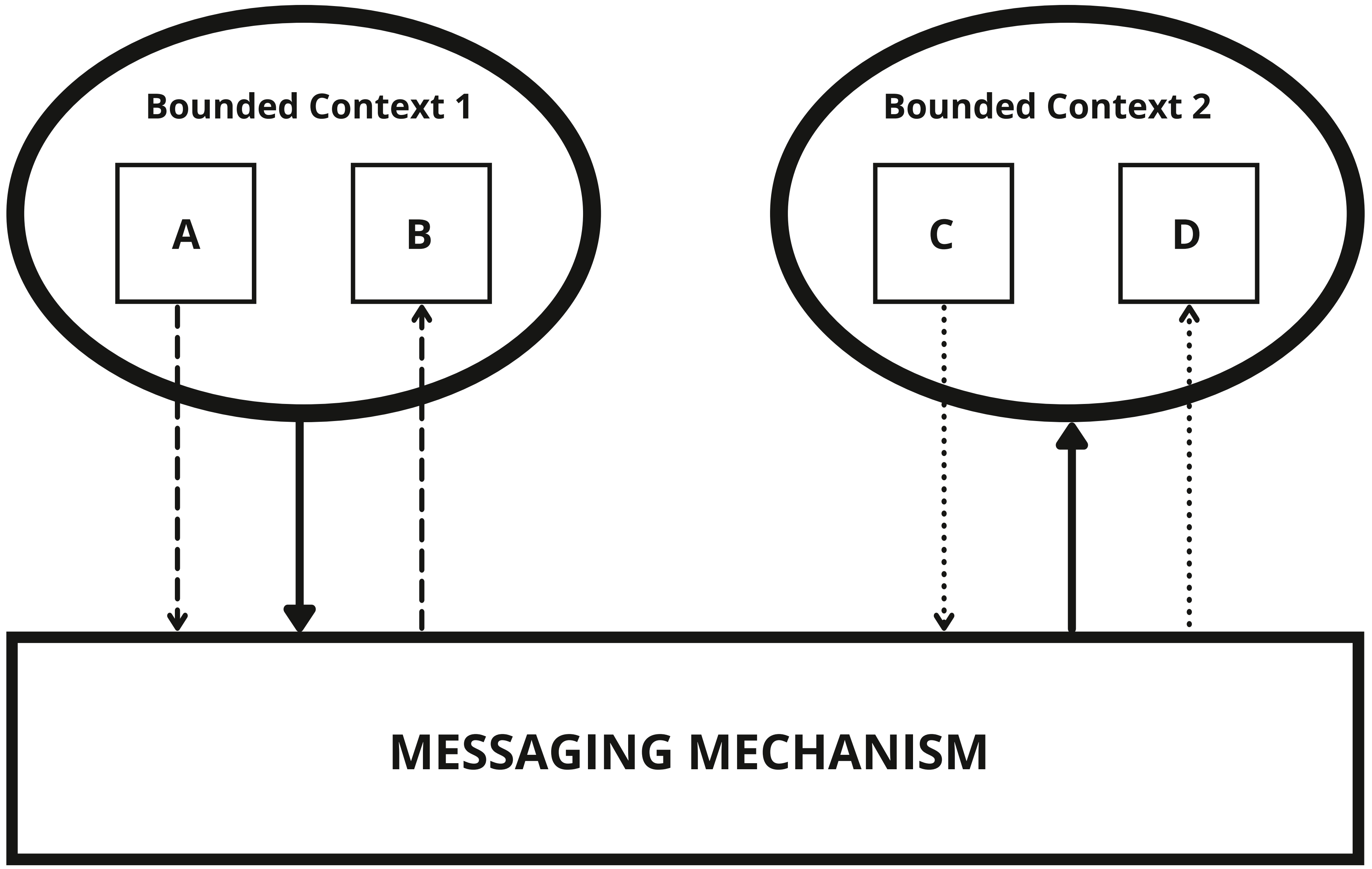
E.g.: DDD



2.3. When to use

- Communication between different transactional boundaries.
- Careful of passive-agressive commands (Martin Fowler).

E.g.: DDD



2.3. When to use

- Communication between different transactional boundaries.
- Careful of passive-agressive commands (Martin Fowler).

E.g.: order_finalized

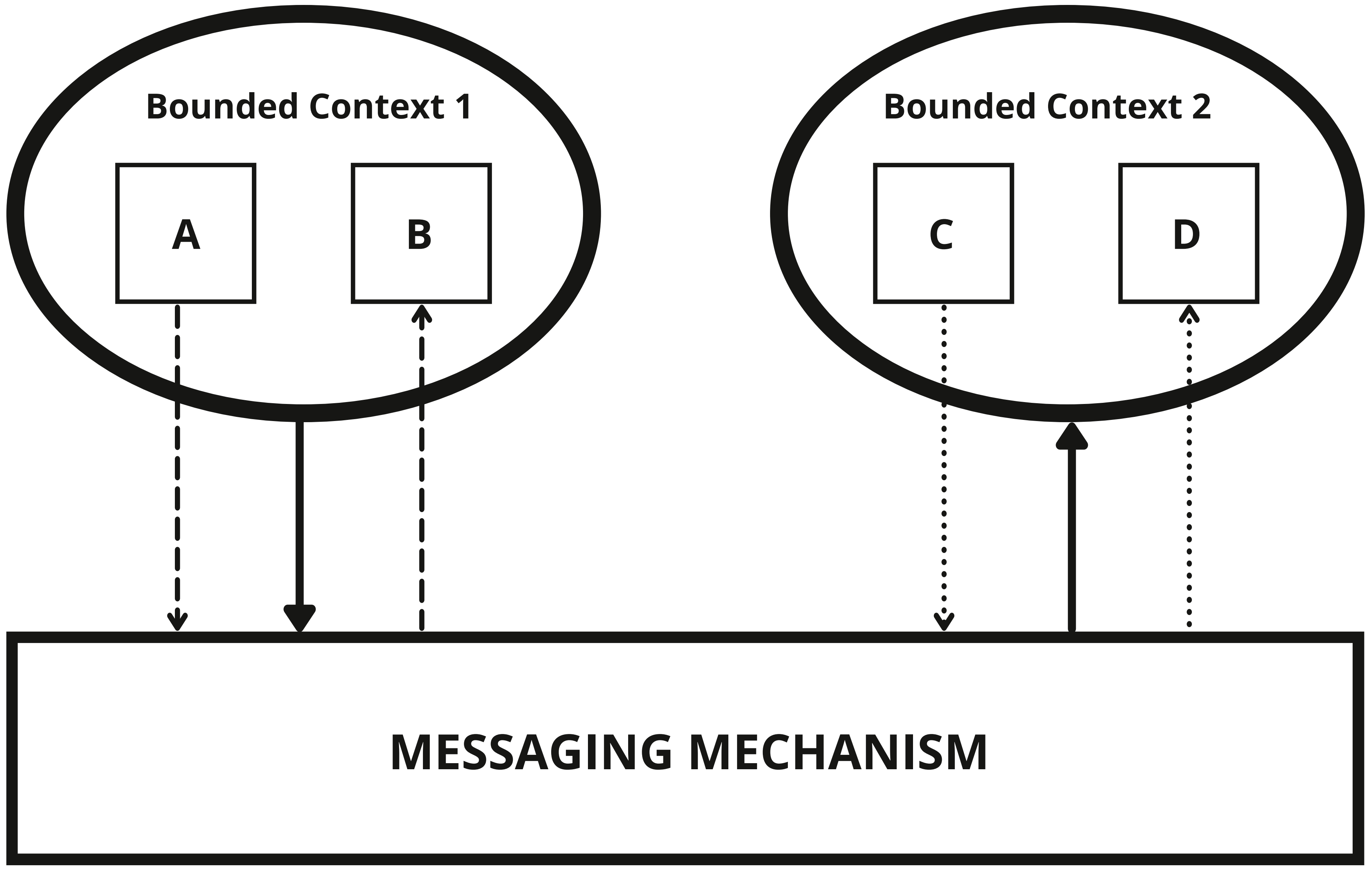
✓ send an email

✓ collect stats

✗ check user before marking as completed

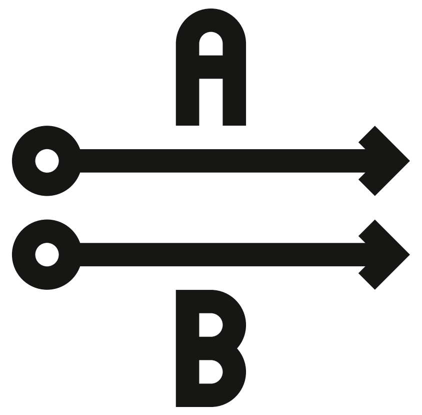
✗ add free item to the order

E.g.: DDD



2.4. How to use

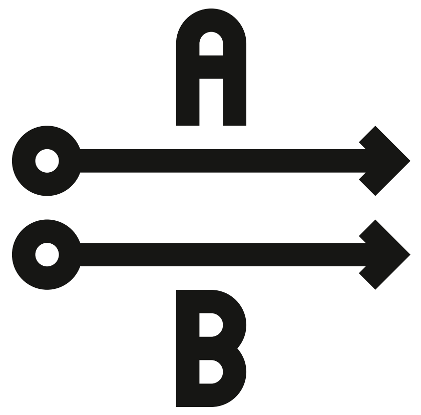
ASYNC



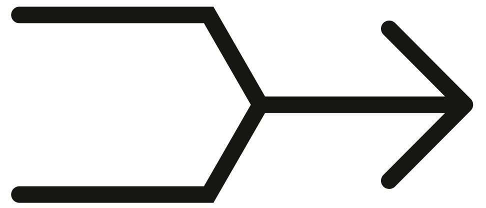
independent

2.4. How to use

ASYNC



independent



idempotent

2.4. How to use

ASYNC

independent

idempotent

fault-tolerant

2.5. Event Notification on Solidus

👍 Free-form event content (payload).

👎 No persistence.

```ruby
# solidus:core/app/models/spree/order.rb
module Spree
  class Order
    # ...
    def finalize!
      #...
      Spree::Event.fire :order_finalized, order: self
    end
  end
end
```

```ruby
# app/subscribers/my_store/sms_notification_subscriber.rb
module MyStore
  module SmsNotificationSubscriber
    include Spree::Event::Subscriber

    event_action :notify_order_completed, event_name: :order_finalized

    def notify_order_completed(event)
      order = event.payload[:order]
      SmsService.new.notify_order_completed(order)
    end
  end
end
```

2.5. Event Notification on Solidus

👍 Free-form event content (payload).

👎 No persistence.

👍 Sync: agnostic of the adapter.

👎 Easy to cross boundaries.

```
# solidus:core/app/models/spree/order.rb
module Spree
  class Order
    # ...
    def finalize!
      #...
      Spree::Event.fire :order_finalized, order: self
    end
  end
end
```

```
# app/subscribers/my_store/sms_notification_subscriber.rb
module MyStore
  module SmsNotificationSubscriber
    include Spree::Event::Subscriber

    event_action :notify_order_completed, event_name: :order_finalized

    def notify_order_completed(event)
      order = event.payload[:order]
      SmsService.new.notify_order_completed(order)
    end
  end
end
```

③ THE EVENT BUS ON SOLIDUS

3.1. Basic usage: Pub/Sub
3.2. Event registration
3.3. Testability
3.4. Observability
3.5. What's next?

3.1. Basic usage: Pub/Sub

- Fire with name and payload.

```ruby
# solidus:core/app/models/spree/order.rb
module Spree
  class Order
    # ...
    def finalize!
      #...
      Spree::Event.fire :order_finalized, order: self
    end
  end
end
```

```ruby
# app/subscribers/my_store/sms_notification_subscriber.rb
module MyStore
  module SmsNotificationSubscriber
    include Spree::Event::Subscriber

    event_action :notify_order_completed, event_name: :order_finalized

    def notify_order_completed(event)
      order = event.payload[:order]
      SmsService.new.notify_order_completed(order)
    end
  end
end
```

3.1. Basic usage: Pub/Sub

- Fire with name and payload.
- Subscriber modules:
  - Match with *event_action* (and *event_name).*

```ruby
# solidus:core/app/models/spree/order.rb
module Spree
  class Order
    # ...
    def finalize!
      #...
      Spree::Event.fire :order_finalized, order: self
    end
  end
end
```

```ruby
# app/subscribers/my_store/sms_notification_subscriber.rb
module MyStore
  module SmsNotificationSubscriber
    include Spree::Event::Subscriber

    event_action :notify_order_completed, event_name: :order_finalized

    def notify_order_completed(event)
      order = event.payload[:order]
      SmsService.new.notify_order_completed(order)
    end
  end
end
```

3.1. Basic usage: Pub/Sub

- Fire with name and payload.
- Subscriber modules:
    - Match with *event_action* (and *event_name).*
- *Subscriber block:*
    - We can subscribe to all the events matching a regex.

```ruby
# solidus:core/app/models/spree/order.rb
module Spree
  class Order
    # ...
    def finalize!
      #...
      Spree::Event.fire :order_finalized, order: self
    end
  end
end
```

```ruby
# app/services/my_store/my_service.rb
# ...
Spree::Event.subscribe(:order_finalized) do |event|
  order = event.payload[:order]
  SmsService.new.notify_order_completed(order)
end
# Spree::Event.subscribe(/^order_.+$/) do |event|
#   # ...
# end
```

3.2. Event registration

- Register an event before using it.
    - Avoids subscribing to invalid events.
    - Avoids name collision.

```ruby
# config/initializers/spree.rb
Spree.config do |config|
  # ...
end

Spree::Event.register :custom_event
```

```ruby
Spree::Event.subscribe(:cstm_evnt) do |event|
  # ...
end
```

```
'cstm_evnt' is not registered as a valid event name.

Did you mean? custom_event

All known events are:

  'order_finalized', 'order_recalculated', 'reimbursement_reimbursed', 'reimbursement_errored', 'custom_event'

You can register the new events at the end of the `spree.rb` initializer:

Spree::Event.register('cstm_evnt')
```

3.2. Event registration

- Register an event before using it.
  - Avoids subscribing to invalid events.
  - Avoids name collision.
- Register at the end of *config/initializers/spree.rb*.

```ruby
# config/initializers/spree.rb
Spree.config do |config|
  # ...
end

Spree::Event.register :custom_event
```

```ruby
Spree::Event.subscribe(:cstm_evnt) do |event|
  # ...
end
```

```
'cstm_evnt' is not registered as a valid event name.

Did you mean? custom_event

All known events are:

  'order_finalized', 'order_recalculated', 'reimbursement_reimbursed', 'reimbursement_errored', 'custom_event'

You can register the new events at the end of the `spree.rb` initializer:

Spree::Event.register('cstm_evnt')
```

3.3. Event testability

- Scope a block to only some listeners. It allows keeping the side effects of other listeners out of the way.

```ruby
# spec/rails_helper.rb
require 'spree/event/test_interface'
Spree::Event.enable_test_interface
```

```ruby
# spec/subscribers/my_store/sms_notification_subscriber.rb
require 'rails_helper'

RSpec.describe MyStore::SmsNotificationSubscriber do
  let(:sms_queue) { SmsService.test_queue }

  it 'sends an SMS when an order is finalized' do
    order = create(:spree_order)

    Spree::Event.performing_only(described_class) do
      Spree::Event.fire(:order_finalized, order: order)
    end

    expect(sms_queue.count).to be(1)
  end
end
```

3.3. Event testability

- Scope a block to only some listeners. It allows keeping the side effects of other listeners out of the way.
- Fine-grained control with `Spree::Event::Subscriber.listeners`.

```ruby
# spec/rails_helper.rb
require 'spree/event/test_interface'
Spree::Event.enable_test_interface
```

```ruby
# spec/subscribers/my_store/sms_notification_subscriber.rb
require 'rails_helper'

RSpec.describe MyStore::SmsNotificationSubscriber do
  let(:sms_queue) { SmsService.test_queue }

  it 'sends an SMS when an order is finalized' do
    order = create(:spree_order)
    listeners = described_class.listeners(:order_finalized)

    Spree::Event.performing_only(listeners) do
      Spree::Event.fire(:order_finalized, order: order)
    end

    expect(sms_queue.count).to be(1)
  end
end
```

3.3. Event testability

- Scope a block to only some listeners. It allows keeping the side effects of other listeners out of the way.
- Fine-grained control with `Spree::Event::Subscriber.listeners`.
- Stub helpers.

```ruby
# spec/services/my_store/custom_service_spec.rb
require 'rails_helper'
require 'spree/testing_support/event_helpers'

RSpec.describe MyStore::CustomService do
  include Spree::TestingSupport::EventHelpers

  describe '#call' do
    stub_spree_events
    order = create(:spree_order)

    described_class.new.call(order)

    expect(:custom_event).to have_been_fired.with(
      a_hash_containing(order: order)
    )
  end
end
```

3.4. Event observability

- An event contains the firing time and the location of the firing code.

```ruby
Spree::Event.subscribe(:order_finalized) do |event|
  puts event.firing_time
  puts event.caller_location
end
# 2022-01-01 00:00:00 UTC
# /path/to/file/that/fired/the/event:99:in `<main>'
```

3.4. Event observability

- An event contains the firing time and the location of the firing code.
- A firing allows inspecting the number of subscribers executed, and for each of them:
    - The execution time.
    - The associated listener.
    - The result.
    - A benchmark measurement.

```ruby
firing = Spree::Event.fire :order_finalized, order: order
puts firing.event.inspect
puts firing.executions.count
puts firing.executions[0].then do |execution|
  puts execution.execution_time
  puts execution.listener
  puts execution.result
  puts execution.benchmark
end
# #<Spree::Event::Event...>
# 3
# 2022-01-01 00:00:00 UTC
# #<Spree::Event::Listener...>
# #<Spree::Order...>
#   0.179883   0.036038   0.215921   (  0.220189)
```

3.5. What's next?

docs

3.5. What's next?

DOC

docs

more events

3.5. What's next?

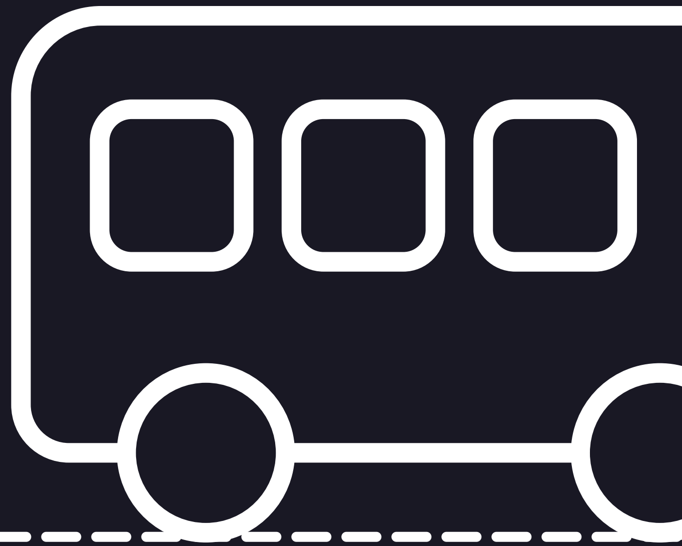
DOC

docs

more events

remove old adapter

Thanks!!



https://github.com/waiting-for-dev/solidusconf7_event_bus

Marc Busqué - @waiting-for-dev