Implementation of a **multithreaded compile-time ECS** in C++14

http://vittorioromeo.info
vittorio.romeo@outlook.com

Bloomberg
vromeo5@bloomberg.net

Meeting C++

Implementation of a **multithreaded compile-time ECS** in C++14

http://vittorioromeo.info
vittorio.romeo@outlook.com

Bloomberg
vromeo5@bloomberg.net

Meeting C++

http://github.com/SuperV1234/meetingcpp2016

Overview of the talk

- **“Entity-component-system”** pattern in general.
- Overview of **ECST**.
  - Design and core values.
  - Features/limitations.
- Complete usage example.
- Architecture of **ECST**.
- Implementation of **ECST**.
- Future ideas.

Entity-component-system

Concepts, benefits and drawbacks of the ECS architectural pattern.

http://vittorioromeo.info
vittorio.romeo@outlook.com

What is an **entity**?

- Something **tied to a concept**.
- Has related **data** and/or **logic**.
- We deal with **many entities**.
  - We may want to **track specific instances**.
- Can be **created** and **destroyed.**
- Examples:
  - **Game objects:** *player, bullet, car.*
  - **GUI widgets:** *window, textbox, button.*

Encoding entities – OOP inheritance

- An **entity type** is a **polymorphic class**.
- **Data** is stored inside the class.
- **Logic** is handled using **runtime** polymorphism.
- **Very easy** to implement.
- **Cache-unfriendly**.
- **Runtime overhead**.
- **Lack of flexibility**.

Encoding entities – OOP inheritance

- An **entity type** is a **polymorphic class**.
- **Data** is stored inside the class.
- **Logic** is handled using **runtime** polymorphism.
- **Very easy** to implement.
- **Cache-unfriendly**.
- **Runtime overhead**.
- **Lack of flexibility**.

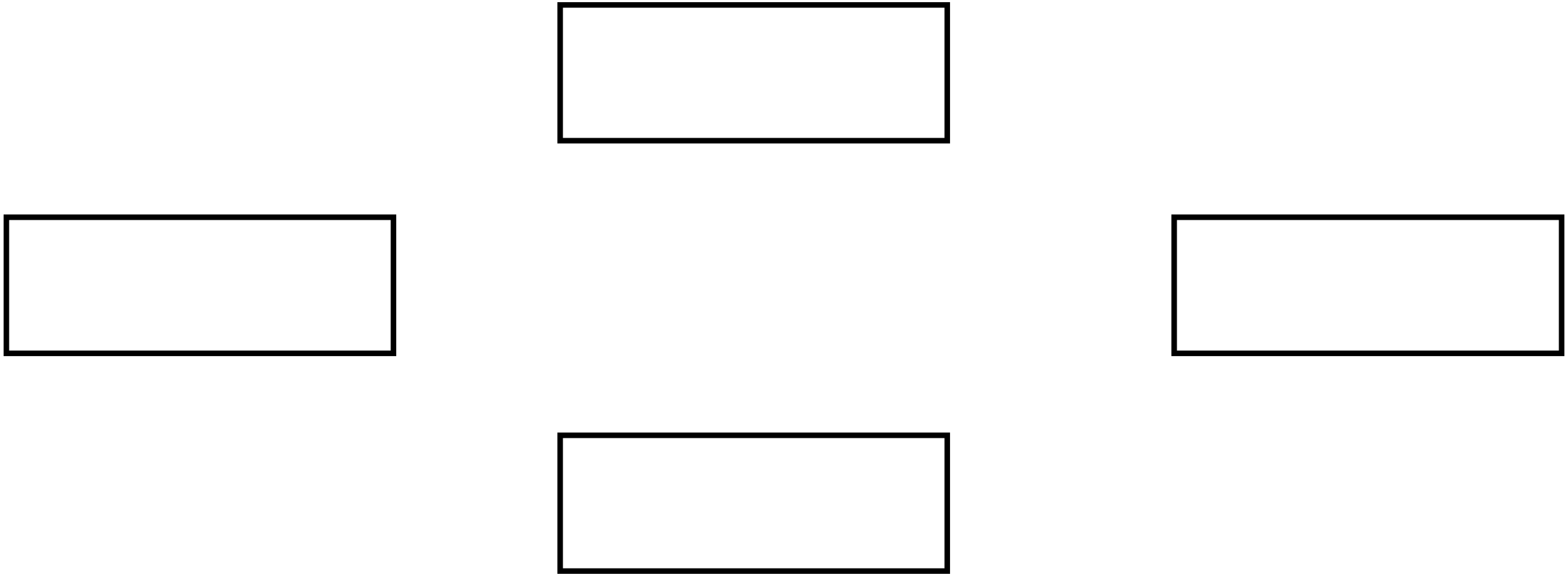
Encoding entities – OOP inheritance

- An **entity type** is a **polymorphic class**.
- **Data** is stored inside the class.
- **Logic** is handled using **runtime** polymorphism.
- **Very easy** to implement.
- **Cache-unfriendly**.
- **Runtime overhead**.
- **Lack of flexibility**.

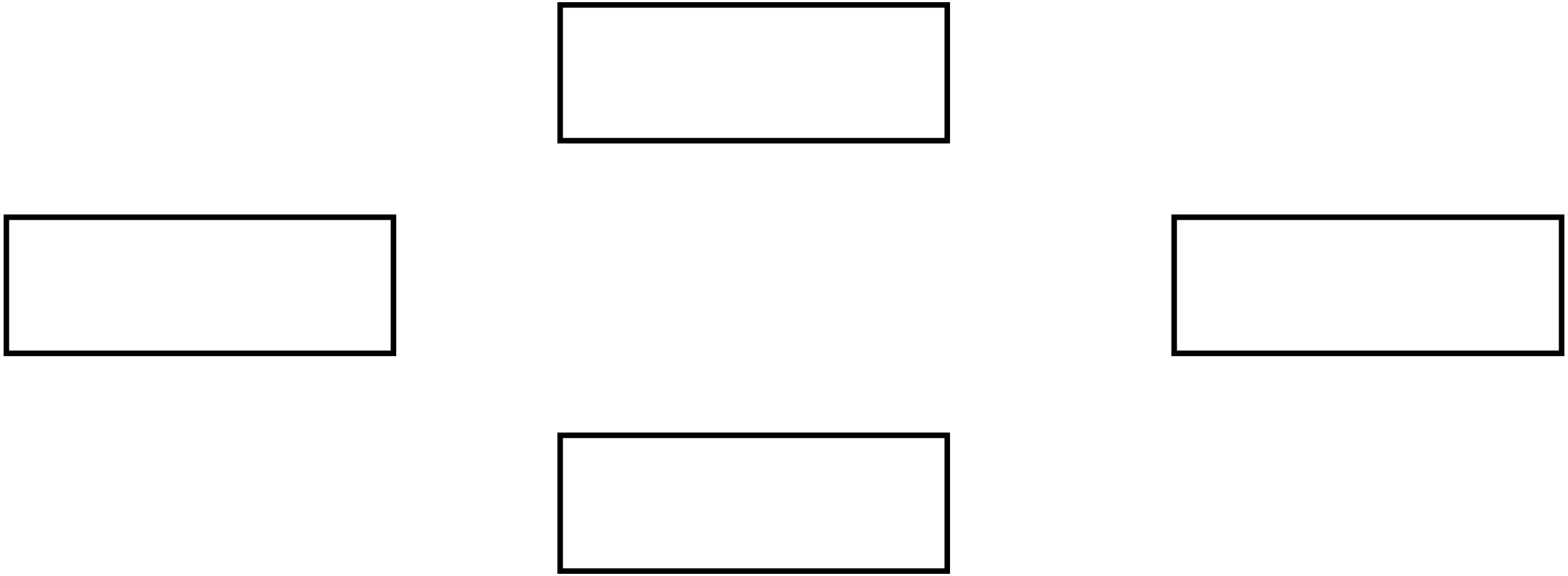
Encoding entities – OOP inheritance

- An **entity type** is a **polymorphic class**.
- **Data** is stored inside the class.
- **Logic** is handled using **runtime** polymorphism.
- **Very easy** to implement.
- **Cache-unfriendly**.
- **Runtime overhead**.
- **Lack of flexibility**.

Encoding entities – OOP inheritance



Encoding entities – OOP inheritance



Encoding entities – OOP inheritance

```cpp
struct Entity
{
    virtual ~Entity() { }
    virtual void update() { }
    virtual void draw() { }
};
```

```cpp
struct Skeleton : Entity
{
    std::vector<Bone> bones;
    void update() override
    {
        // do things skeletons do
    }
};
```

Encoding entities – OOP inheritance

```cpp
struct Entity
{
    virtual ~Entity() { }
    virtual void update() { }
    virtual void draw() { }
};
```

```cpp
struct Skeleton : Entity
{
    std::vector<Bone> bones;
    void update() override
    {
        // do things skeletons do
    }
};
```

Encoding entities – OOP composition

- An **entity** is an aggregate of **components**.
- Components **store data** and have **logic**.
- **Logic** is handled using **runtime** polymorphism.
- **Easy** to implement.
- More **flexible**.
- **Cache-unfriendly**.
- **Runtime overhead**.

Encoding entities – OOP composition

- An **entity** is an aggregate of **components**.
- Components **store data** and have **logic**.
- **Logic** is handled using **runtime** polymorphism.
- **Easy** to implement.
- More **flexible**.
- **Cache-unfriendly**.
- **Runtime overhead**.

Encoding entities – OOP composition

- An **entity** is an aggregate of **components**.
- Components **store data** and have **logic**.
- **Logic** is handled using **runtime** polymorphism.
- **Easy** to implement.
- More **flexible**.
- **Cache-unfriendly**.
- **Runtime overhead**.

Encoding entities – OOP composition

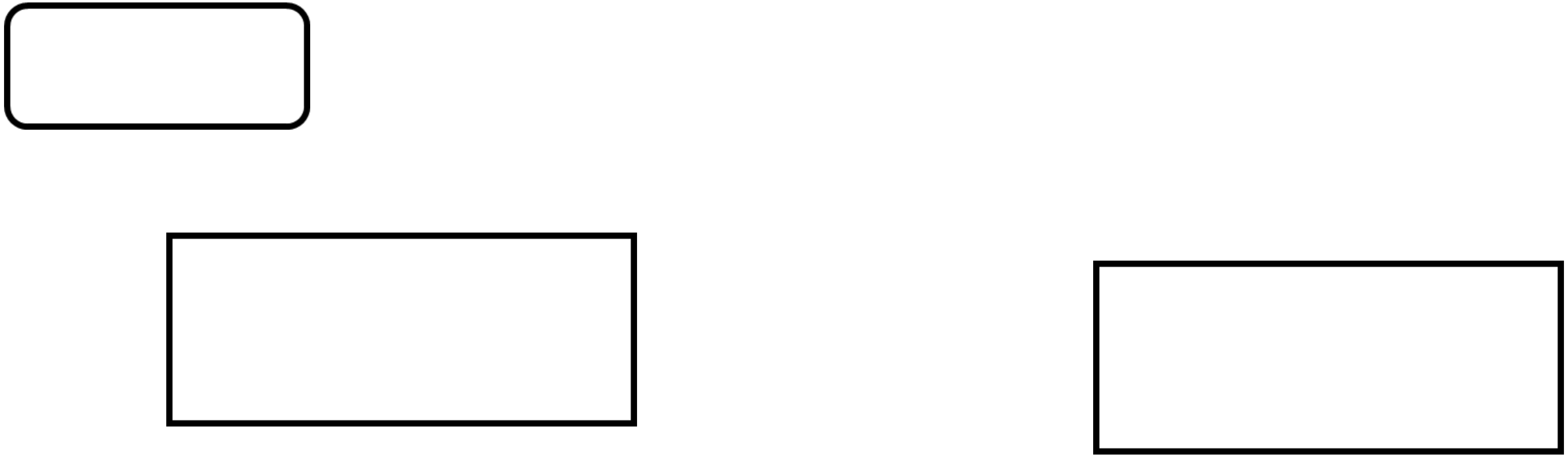
- An **entity** is an aggregate of **components**.
- Components **store data** and have **logic**.
- **Logic** is handled using **runtime** polymorphism.
- **Easy** to implement.
- More **flexible**.
- **Cache-unfriendly**.
- **Runtime overhead**.

Encoding entities – OOP composition

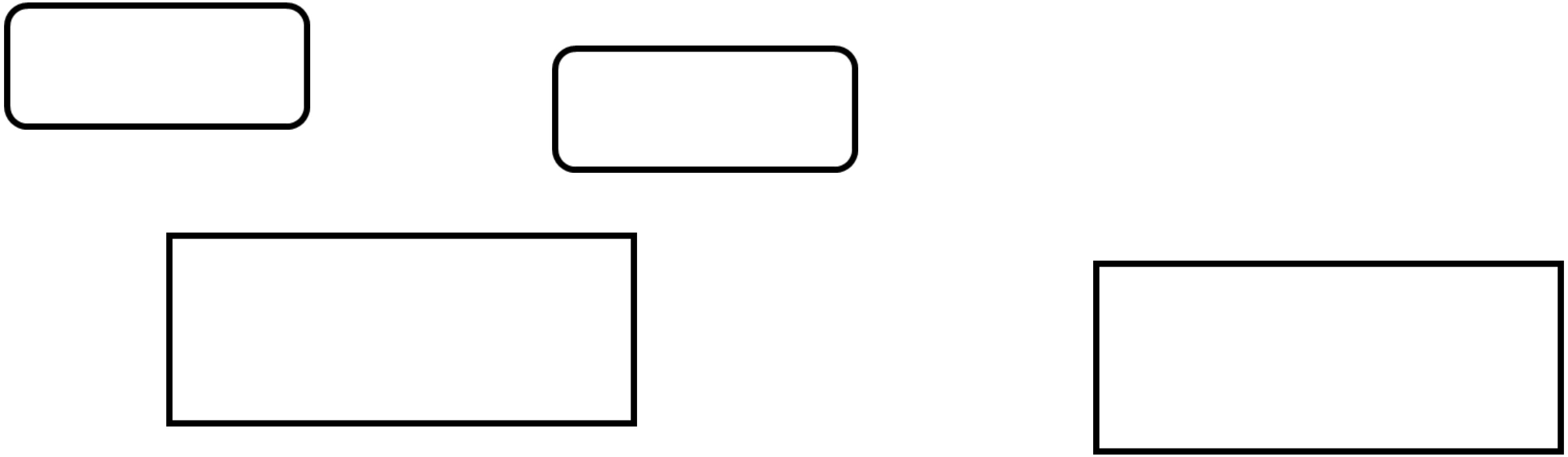
- An **entity** is an aggregate of **components**.
- Components **store data** and have **logic**.
- **Logic** is handled using **runtime** polymorphism.
- **Easy** to implement.
- More **flexible**.
- **Cache-unfriendly**.
- **Runtime overhead**.

Encoding entities – OOP composition

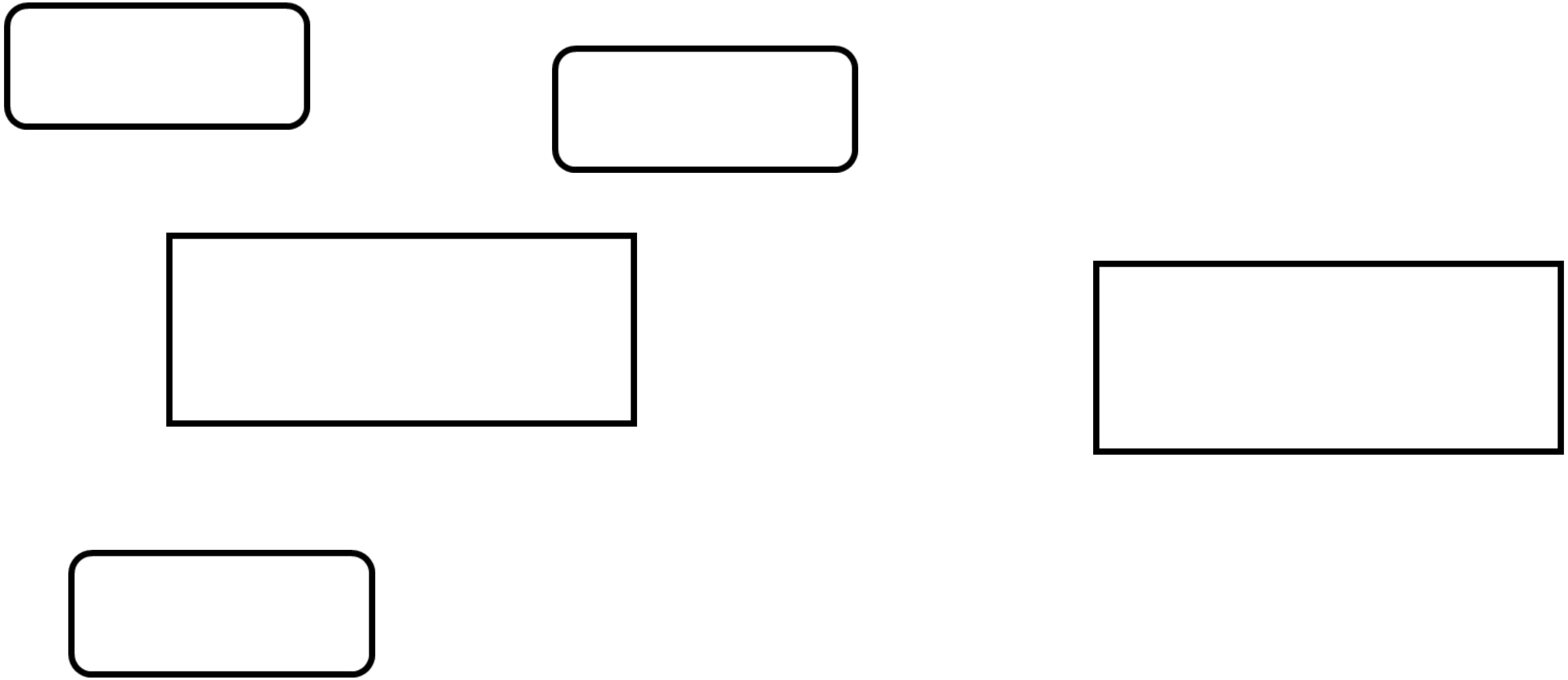
Encoding entities – OOP composition



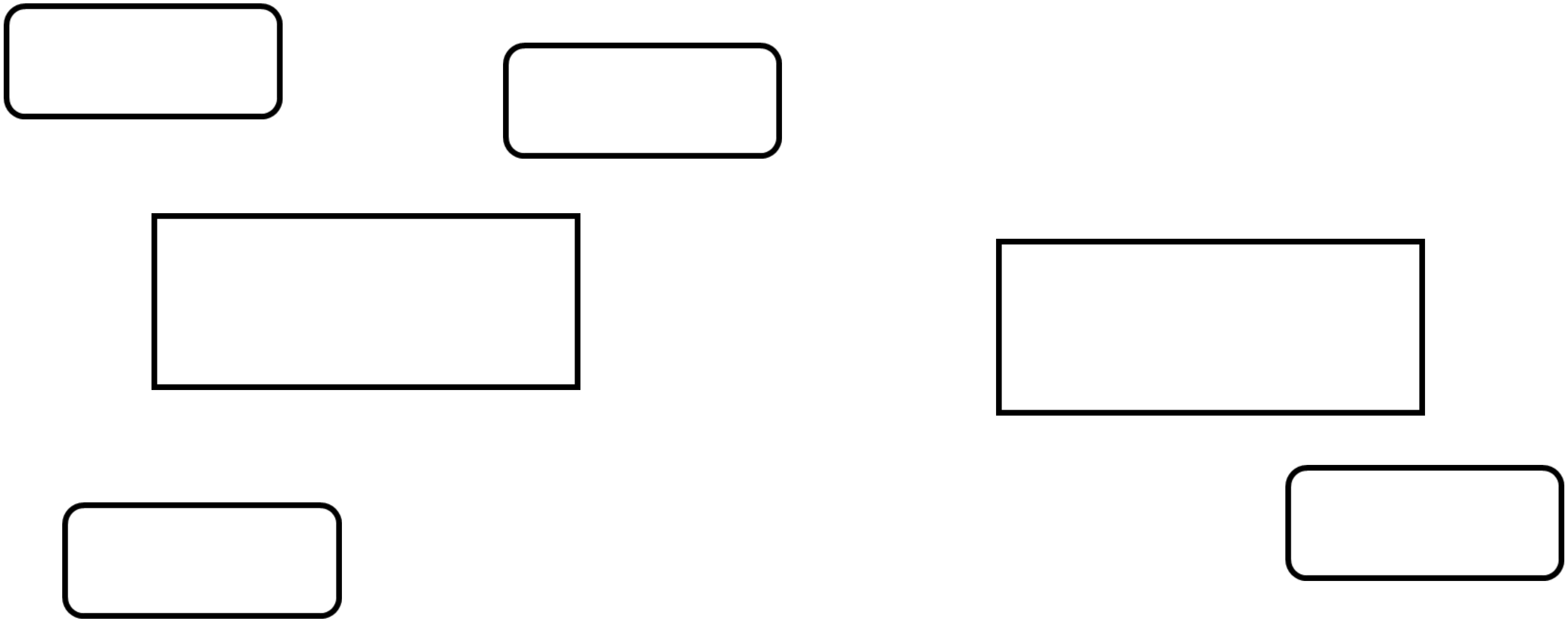
Encoding entities – OOP composition



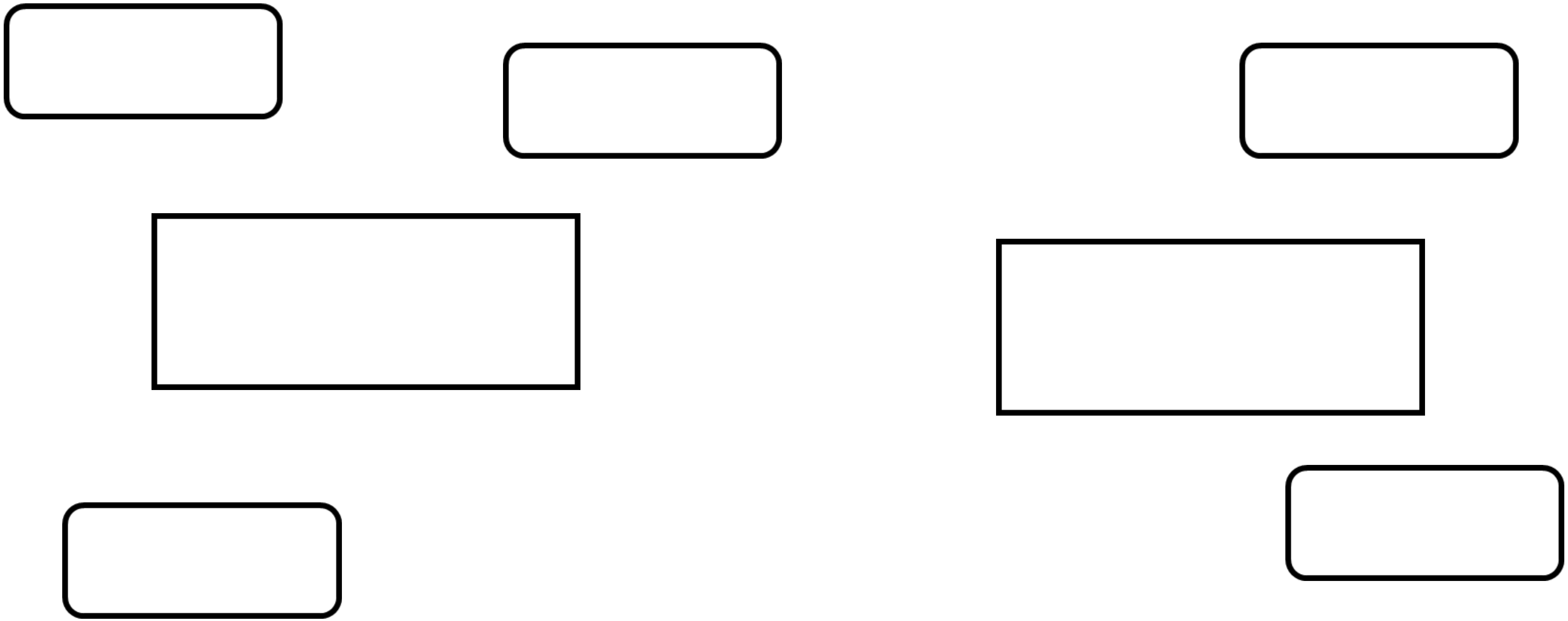
Encoding entities – OOP composition



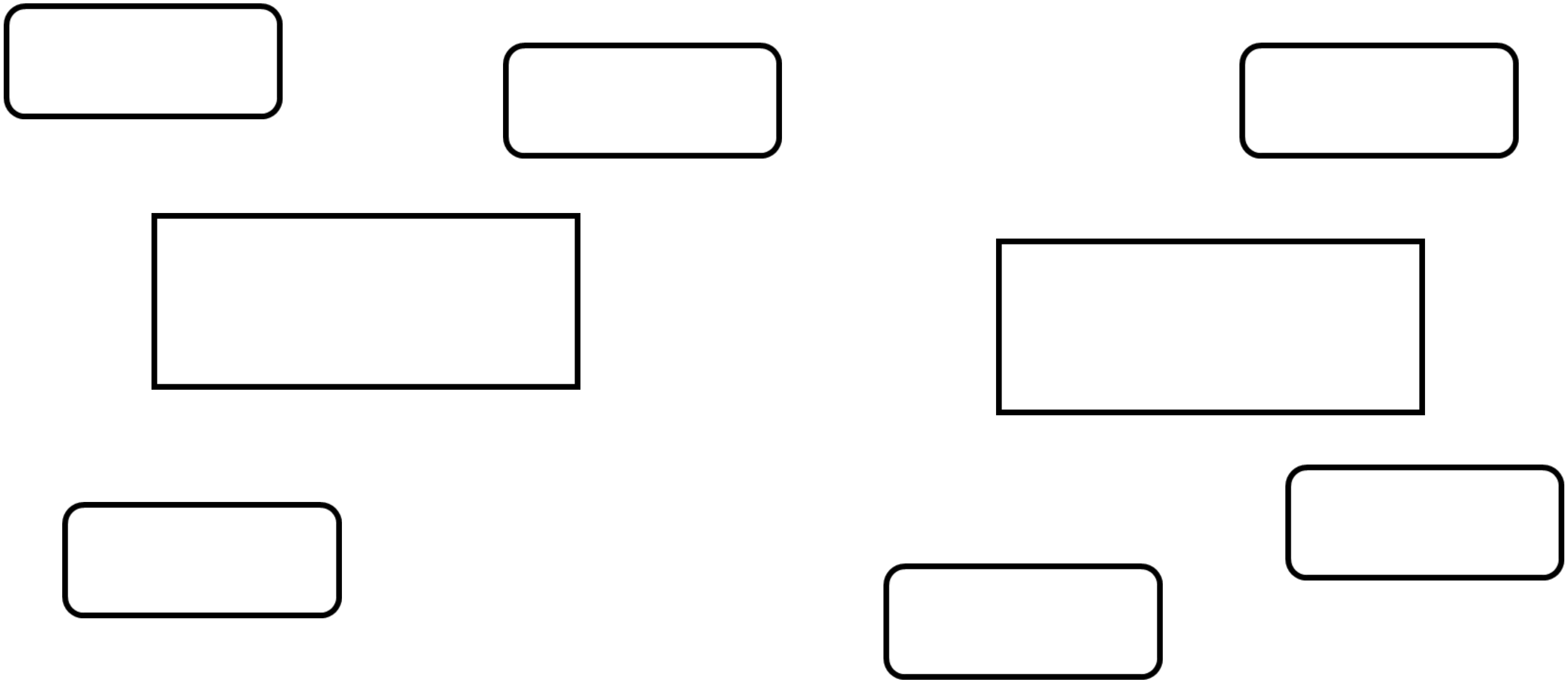
Encoding entities – OOP composition



Encoding entities – OOP composition



Encoding entities – OOP composition



Encoding entities – OOP composition

```cpp
struct Component
{
    virtual ~Component() { }
    virtual void update() { }
    virtual void draw() { }
};

struct Entity
{
    std::vector<std::unique_ptr<Component>> components;

    void update() { for(auto& c : components) c->update(); }
    void draw() { for(auto& c : components) c->draw(); }
};
```

Encoding entities – OOP composition

```cpp
struct Component
{
    virtual ~Component() { }
    virtual void update() { }
    virtual void draw() { }
};

struct Entity
{
    std::vector<std::unique_ptr<Component>> components;

    void update() { for(auto& c : components) c->update(); }
    void draw() { for(auto& c : components) c->draw(); }
};
```


http://vittorioromeo.info
vittorio.romeo@outlook.com

Encoding entities – OOP composition

```cpp
struct BonesComponent : Component
{
    std::vector<Bone> bones;
    void update() override
    {
        // do things skeletons do
    }
};

auto makeSkeleton()
{
    Entity e;
    e.components.emplace_back(std::make_unique<BonesComponent>());
    return e;
}
```

Encoding entities – OOP composition

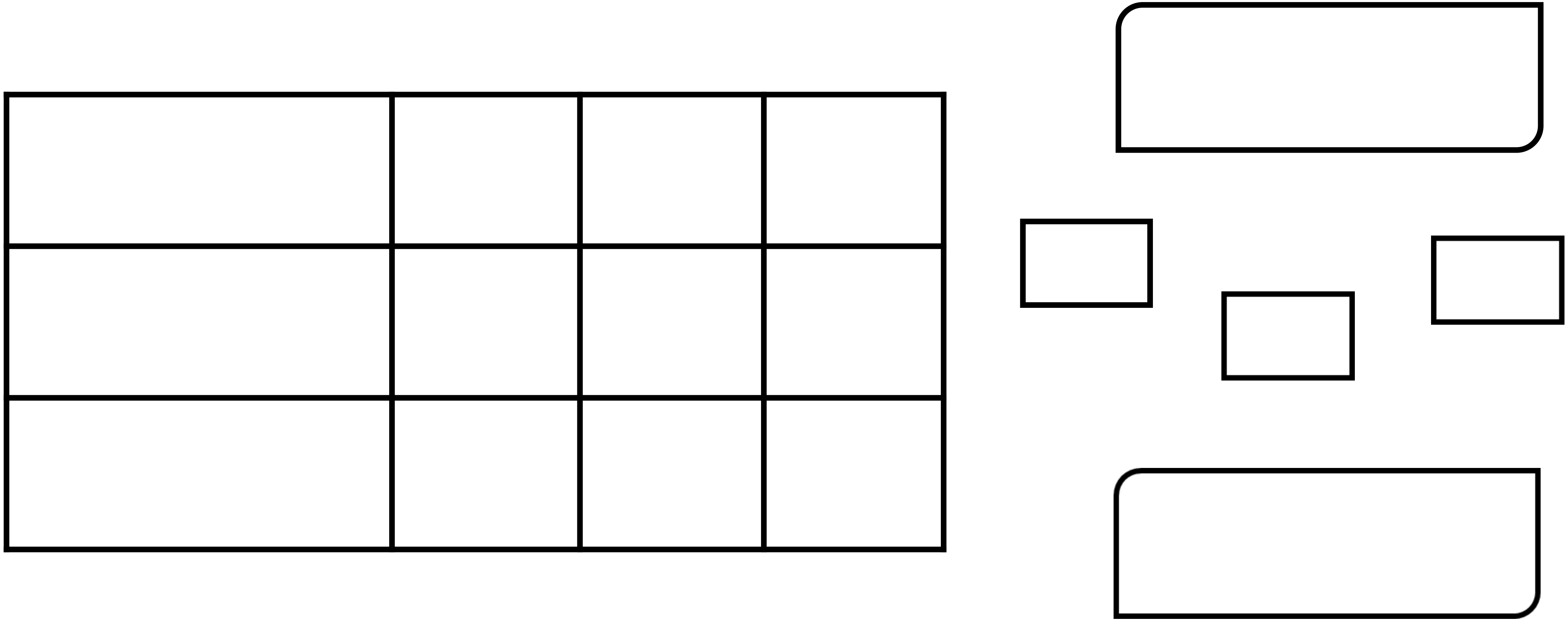
```
struct BonesComponent : Component
{
    std::vector<Bone> bones;
    void update() override
    {
        // do things skeletons do
    }
};

auto makeSkeleton()
{
    Entity e;
    e.components.emplace_back(std::make_unique<BonesComponent>());
    return e;
}
```

Encoding entities – DOD composition

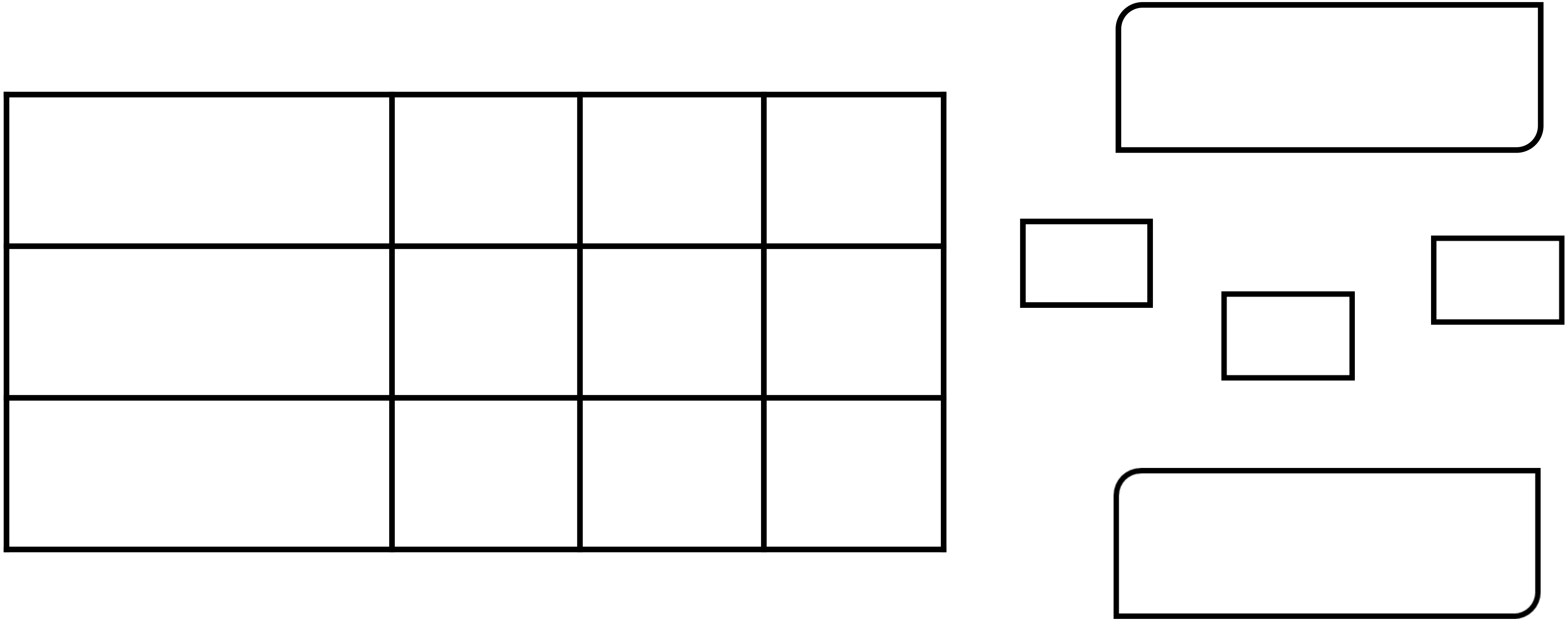
- An **entity** is a numerical **ID**.
- Components only store **data** (logicless).
- **Logic** is handled using **systems**.
- Potentially **cache-friendly**.
- Minimal **runtime overhead.**
- Great **flexibility.**
- **Hard** to implement.

ID	NAME	KEYBOARD	STYLE	FOCUS	MOUSE
0	TextBox	✓	✓	✓	
1	Button		✓	✓	✓
2	Browser	✓		✓	✓

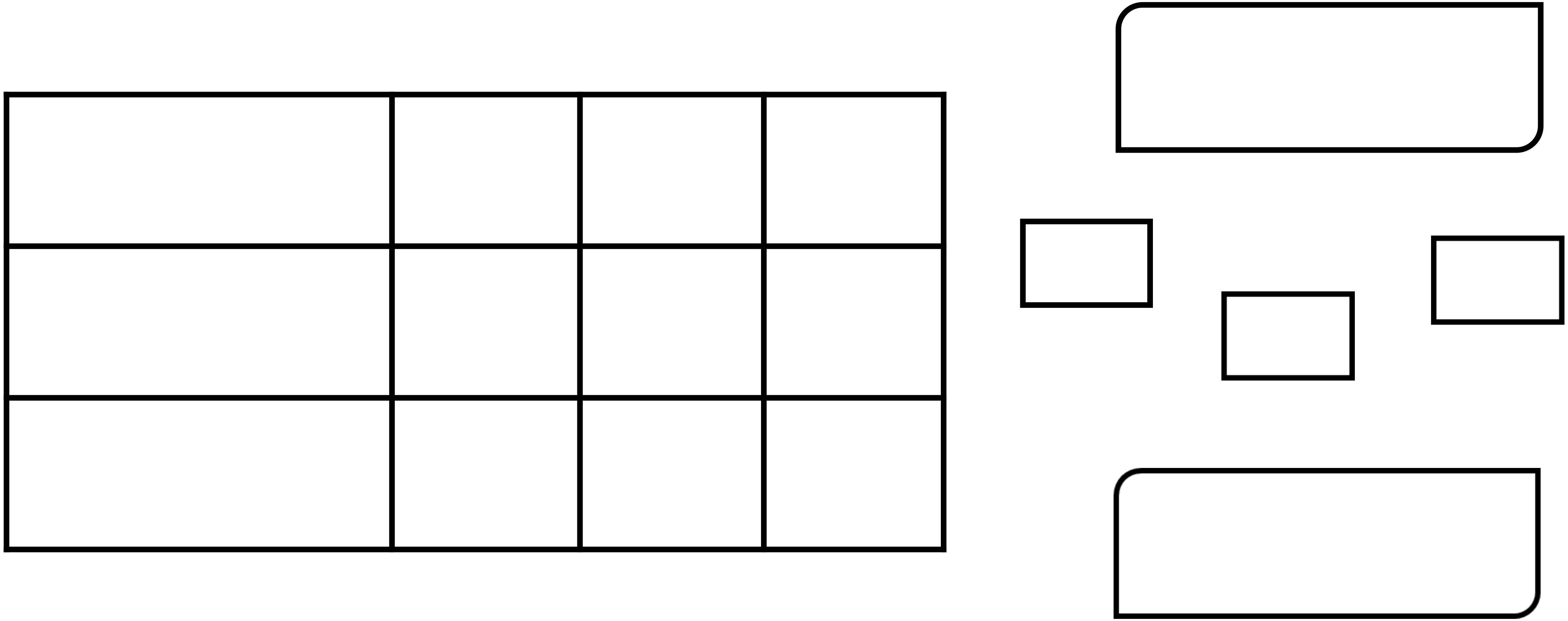
Encoding entities – DOD composition



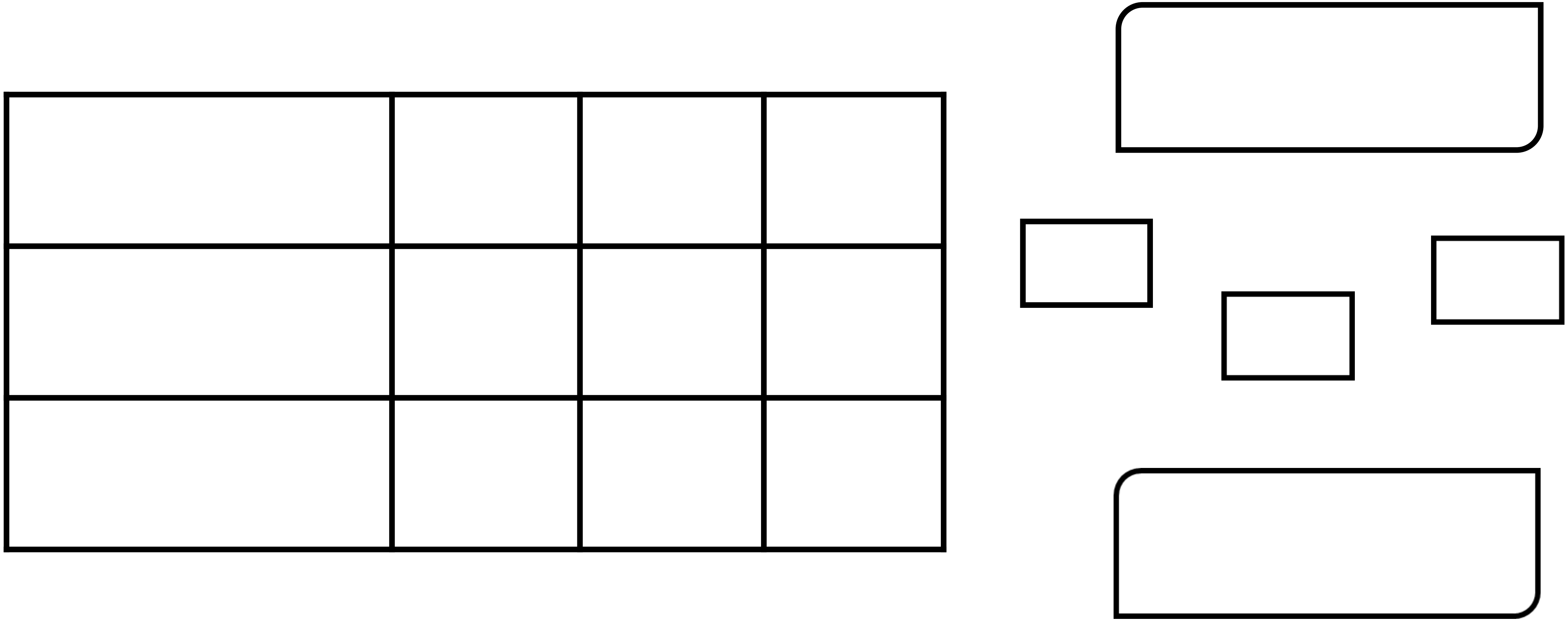
Encoding entities – DOD composition



Encoding entities – DOD composition



Encoding entities – DOD composition



Encoding entities – DOD composition

Encoding entities – DOD composition

Encoding entities – DOD composition

Encoding entities – DOD composition

Encoding entities – DOD composition

Encoding entities – DOD composition

Encoding entities – DOD composition

```cpp
using Entity = std::size_t;
constexpr std::size_t maxEntities{1000};

struct BonesComponent { /* ... */ };
struct AIComponent { /* ... */ };
struct SpriteComponent { /* ... */ };

struct Manager
{
    std::array<BonesComponent, maxEntities> bonesComponents;
    std::array<AIComponent, maxEntities> aiComponents;
    std::array<SpriteComponent, maxEntities> spritesComponents;

    // ...
};
```

Encoding entities – DOD composition

```
using Entity = std::size_t;
constexpr std::size_t maxEntities{1000};

struct BonesComponent { /* ... */ };
struct AIComponent { /* ... */ };
struct SpriteComponent { /* ... */ };

struct Manager
{
    std::array<BonesComponent, maxEntities> bonesComponents;
    std::array<AIComponent, maxEntities> aiComponents;
    std::array<SpriteComponent, maxEntities> spritesComponents;

    // ...
};
```

Encoding entities – DOD composition

```cpp
using Entity = std::size_t;
constexpr std::size_t maxEntities{1000};

struct BonesComponent { /* ... */ };
struct AIComponent { /* ... */ };
struct SpriteComponent { /* ... */ };

struct Manager
{
    std::array<BonesComponent, maxEntities> bonesComponents;
    std::array<AIComponent, maxEntities> aiComponents;
    std::array<SpriteComponent, maxEntities> spritesComponents;

    // ...
};
```

Encoding entities – DOD composition

```cpp
using Entity = std::size_t;
constexpr std::size_t maxEntities{1000};

struct BonesComponent { /* ... */ };
struct AIComponent { /* ... */ };
struct SpriteComponent { /* ... */ };

struct Manager
{
    std::array<BonesComponent, maxEntities> bonesComponents;
    std::array<AIComponent, maxEntities> aiComponents;
    std::array<SpriteComponent, maxEntities> spritesComponents;

    // ...
};
```

Encoding entities – DOD composition

```cpp
struct SkeletonSystem
{
    Manager& manager;

    void process()
    {
        manager.forEntitiesWith<AIComponent, BonesComponent>(
            [](auto& data, auto id)
            {
                auto& ai = data.get<AIComponent>(id);
                auto& bones = data.get<BonesComponent>(id);

                // ...
            });
    }
};
```

Storing components - **one array per type**

- **Very easy** to implement.
- Suitable for **most** projects.
- Easy to **add/remove** components at runtime.
- Can be «**cache-friendlier**».
- **Wasteful** of memory.

Storing components – **mega-array**

- Potentially **cache-friendlier**.
- **Minimizes** memory waste.
- **Very hard** to implement.
  - Requires **indexing tables**.
  - **Hard** to deal with component **addition/removal.**
  - **Very hard** to **keep track** of free «slots» in the mega-array.

ECST in a nutshell

What is it?

ECST in a nutshell – basics

- Given some **component** and **system** types…
- ECST allows users to **declaratively** define at **compile-time**:
  - **Component storage layouts.**
    - Used to leverage cache and easily play around with *SOA*/*AOS* layouts.
  - **Relationships/dependencies between systems.**
    - Used to automatically parallelize system execution.
  - **Parallelism strategies.**
    - System-specific parallelization choices. *(e.g. split system **S0** in 4 sub-tasks)*
  - **Application-wide settings.**
    - *(e.g. fixed entity limit VS dynamic entity limit)*
- Application logic is **completely independent** from the choices above.

ECST in a nutshell – declarative settings

- Compile-time data structure: **option maps**.
  - Made easy thanks to **boost::hana**!
- User syntax: **fluent method chaining.**

```
constexpr auto s =
    ecst::settings::make()
        .allow_inner_parallelism()
        .fixed_entity_limit(sz_v<50000>)
        .component_signatures(csl)
        .system_signatures(ssl)
        .scheduler(st::atomic_counter);
```

```
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

Simple C++ classes.
No concepts to satisfy.
No ECST-specific classes to derive from.

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

Wrappers around types that can be used as values.
Equivalent to:
`boost::hana::type_c`

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::`**
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

```
constexpr auto cs_physics =
    cs::make(ct::acceleration, ct::velocity)
        .contiguous_buffer();
```

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use `ecst::context`.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

```
constexpr auto ssig_render =
    ss::make(st::render)
        .parallelism(none)
        .dependencies(st::velocity)
        .read(ct::position, ct::sprite)
        .output(ss::output<std::vector<Vertex>>);
```

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

ECST in a nutshell – how t

```
constexpr auto s =
    ecst::settings::make()
        .allow_inner_parallelism()
        .fixed_entity_limit(sz_v<50000>)
        .component_signatures(csl)
        .system_signatures(ssl)
        .scheduler(st::atomic_counter);
```

1. Define component types and t
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

ECST in a nutshell – how to use

1. Define component types and tags.
2. Define system types and tags.
3. Define **component signatures**.
4. Define **system signatures**.
5. Define **context settings**.
6. Instantiate and use **`ecst::context`**.
7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

ECST in a nutshell – how to use

7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

ECST in a nutshell – how to use

```
context.step([&](auto& proxy)
{
    proxy.execute_systems()(
        sea::t(st::acceleration, st::velocity)
            .for_subtasks([dt](auto& s, auto& data)
            {
                s.process(dt, data);
            }));
});
```

7. Access **system interfaces** and **component data** through **proxy objects** provided by the context.

Overview of **ECST**

Design, features, limitations and examples.

Core values and concepts

- **«Compile-time»** ECS.
- Customizable **settings/policy-based** design.
- User-friendly syntax, independent from settings.
- **Multithreading** support.
  - Avoid explicit locking.
  - **«Dataflow»** programming.
- Modern **`boost::hana`** metaprogramming.
- Clean, modern and safe C++14.

«Compile-time» ECS

- Component and system **types** are known at **compile-time**.
- Component **instances** can be created and destroyed at **run-time**.
- **Entities** can be created, destroyed and tracked at **run-time**.
- Combining it with run-time solutions is possible.
- Minimal run-time overhead.
- Better optimization opportunities.
- Longer compilation times.
- Unwieldy error messages.

«Compile-time» ECS

```cpp
struct position
{
    float _x;
    float _y;
};

struct velocity
{
    float _x;
    float _y;
};

struct acceleration
{
    float _x;
    float _y;
};
```

```cpp
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

```cpp
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

«Compile-time» ECS

```cpp
struct position
{
    float _x;
    float _y;
};

struct velocity
{
    float _x;
    float _y;
};

struct acceleration
{
    float _x;
    float _y;
};
```

```cpp
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

```cpp
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

«Compile-time» ECS

```
struct position
{
    float _x;
    float _y;
};

struct velocity
{
    float _x;
    float _y;
};

struct acceleration
{
    float _x;
    float _y;
};
```

```
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

```
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

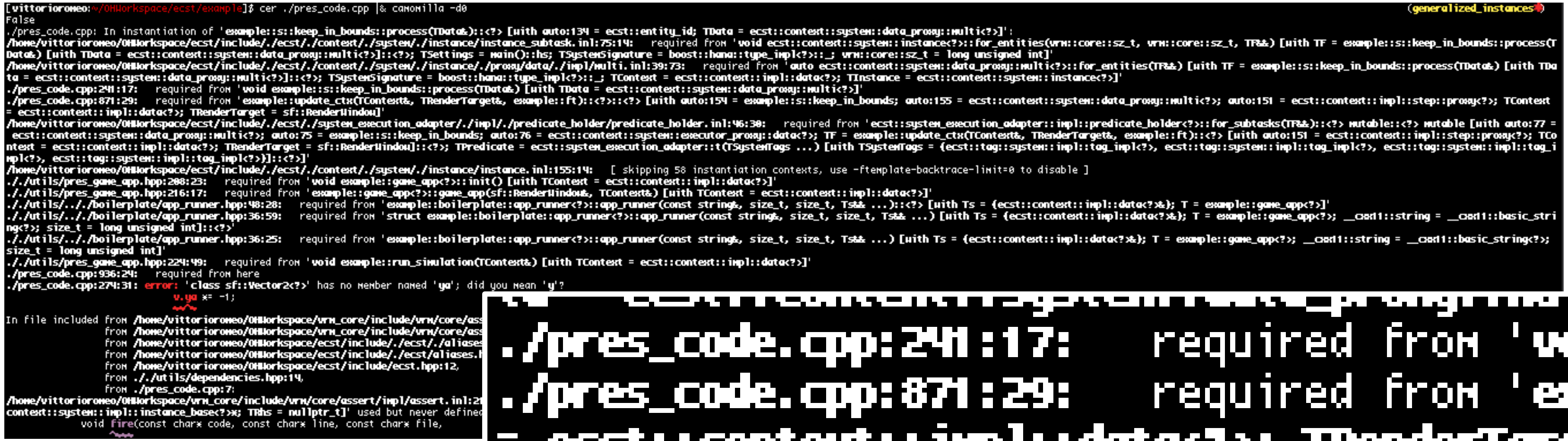
«Compile-time» ECS – **g++ errors**

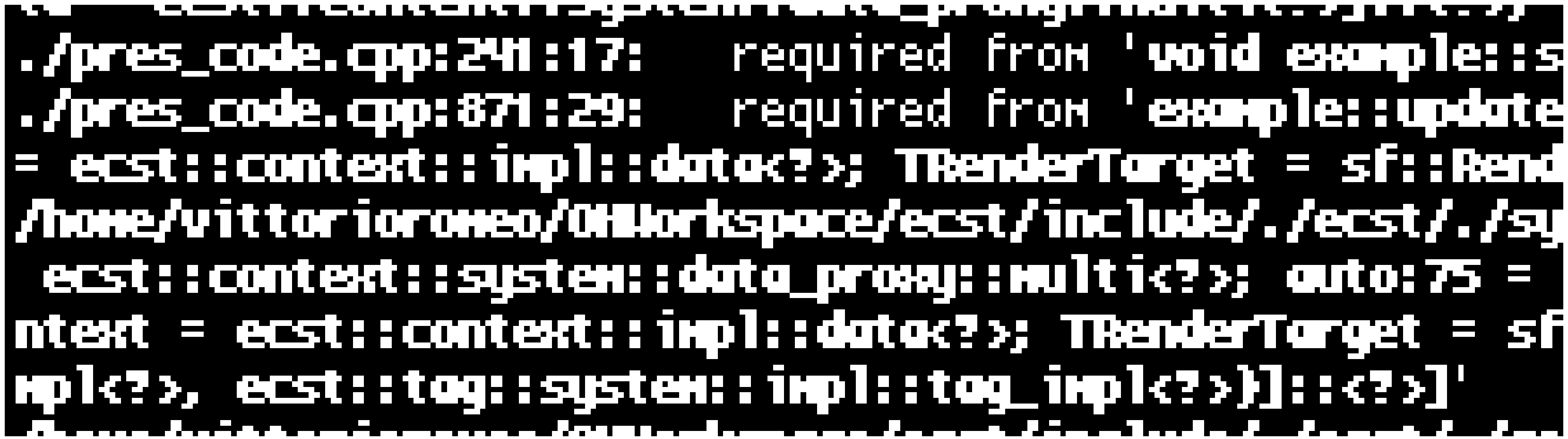
```
tegy::none::impl::parameters, std::tuple<ecst::signature::system::impl::tag_impl<example::s::spatial_partition> >, std::tuple<>, ecst::signature::system::output::impl::o_
ion>, ecst::inner_parallelism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn::v_cores_getter>, std::tuple<ecst::signature:
xample::c::velocity>, ecst::signature::system::impl::mutate_impl<example::c::position>, ecst::signature::system::impl::read_impl<example::c::circle> >, ecst::signature::s
::signature::system::impl::data<ecst::signature::system::impl::tag_impl<example::s::solve_contacts>, ecst::inner_parallelism::strategy::none::impl::parameters, std::tuple
:impl::mutate_impl<example::c::velocity>, ecst::signature::system::impl::mutate_impl<example::c::position>, ecst::signature::system::impl::read_impl<example::c::circle> >,
:system::impl::tag_impl<example::s::render_colored_circle>, ecst::inner_parallelism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_
cts> >, std::tuple<ecst::signature::system::impl::read_impl<example::c::circle>, ecst::signature::system::impl::read_impl<example::c::position>, ecst::signature::system::
rtex, std::allocator<sf::Vertex> > > > >, ecst::settings::impl::scheduler_wrapper<ecst::scheduler::s_atomic_counter> >; TSystemSignature = ecst::signature::system::impl::
ism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn::v_cores_getter>, std::tuple<ecst::signature::system::impl::tag_impl<ex
ecst::signature::system::impl::read_impl<example::c::position>, ecst::signature::system::impl::read_impl<example::c::color> >, ecst::signature::system::output::impl::o_da
instance<TSettings, TSystemSignature>::make_other_entity_range_provider(vrm::core::sz_t, vrm::core::sz_t) [with TSettings = ecst::settings::impl::data<ecst::settings::imp
pl<std::integral_constant<long unsigned int, 20000ul> >, std::tuple<ecst::signature::component::impl::data<ecst::signature::component::impl::tag_impl<example::c::position
locity> >, ecst::signature::component::impl::data<ecst::signature::component::impl::tag_impl<example::c::acceleration> >, ecst::signature::component::impl::data<ecst::sign
ignature::component::impl::tag_impl<example::c::circle> >, ecst::signature::component::impl::data<ecst::signature::component::impl::tag_impl<example::c::mass> > >, std::tu
ion>, ecst::inner_parallelism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn::v_cores_getter>, std::tuple<>, std::tuple<ec
pl<example::c::acceleration> >, ecst::signature::system::output::impl::o_none>, ecst::signature::system::impl::data<ecst::signature::system::impl::tag_impl<example::s::ve
:strategy::split_evenly_fn::v_cores_getter>, std::tuple<ecst::signature::system::impl::tag_impl<example::s::acceleration> >, std::tuple<ecst::signature::system::impl::mut
::signature::system::output::impl::o_none>, ecst::signature::system::impl::data<ecst::signature::system::impl::tag_impl<example::s::keep_in_bounds>, ecst::inner_paralleli
cores_getter>, std::tuple<ecst::signature::system::impl::tag_impl<example::s::velocity> >, std::tuple<ecst::signature::system::impl::mutate_impl<example::c::velocity>, ec
pl<example::c::circle> >, ecst::signature::system::output::impl::o_none>, ecst::signature::system::impl::data<ecst::signature::system::impl::tag_impl<example::s::spatial_
ism::strategy::split_evenly_fn::v_cores_getter>, std::tuple<ecst::signature::system::impl::tag_impl<example::s::keep_in_bounds> >, std::tuple<ecst::signature::system::impl
t::signature::system::output::impl::o_data<std::vector<example::sp_data, std::allocator<example::sp_data> > > >, ecst::signature::system::impl::data<ecst::signature::syst
::tuple<ecst::signature::system::impl::tag_impl<example::s::spatial_partition> >, std::tuple<>, ecst::signature::system::output::impl::o_none>, ecst::signature::system::i
ategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn::v_cores_getter>, std::tuple<ecst::signature::system::impl::tag_impl<example::
ture::system::impl::mutate_impl<example::c::position>, ecst::signature::system::impl::read_impl<example::c::circle> >, ecst::signature::system::output::impl::o_data<std::
cst::signature::system::impl::tag_impl<example::s::solve_contacts>, ecst::inner_parallelism::strategy::none::impl::parameters, std::tuple<ecst::signature::system::impl::t
locity>, ecst::signature::system::impl::mutate_impl<example::c::position>, ecst::signature::system::impl::read_impl<example::c::circle> >, ecst::signature::system::output
s::render_colored_circle>, ecst::inner_parallelism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn::v_cores_getter>, std::
e::system::impl::read_impl<example::c::circle>, ecst::signature::system::impl::read_impl<example::c::position>, ecst::signature::system::impl::read_impl<example::c::color
> > > >, ecst::settings::impl::scheduler_wrapper<ecst::scheduler::s_atomic_counter> >; TSystemSignature = ecst::signature::system::impl::data<ecst::signature::system::imp
pl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn::v_cores_getter>, std::tuple<ecst::signature::system::impl::tag_impl<example::s::solve_contacts> >, std:
ad_impl<example::c::position>, ecst::signature::system::impl::read_impl<example::c::color> >, ecst::signature::system::output::impl::o_data<std::vector<sf::Vertex> > >; w
settings::impl::data<ecst::settings::impl::threading::multi<ecst::settings::impl::system_parallelism::enabled>, ecst::settings::impl::fixed_impl<std::integral_constant<lo
ent::impl::tag_impl<example::c::position> >, ecst::signature::component::impl::data<ecst::signature::component::impl::tag_impl<example::c::velocity> >, ecst::signature::c
nature::component::impl::data<ecst::signature::component::impl::tag_impl<example::c::color> >, ecst::signature::component::impl::data<ecst::signature::component::impl::ta
:tag_impl<example::c::mass> > >, std::tuple<ecst::signature::system::impl::data<ecst::signature::system::impl::tag_impl<example::s::acceleration>, ecst::inner_parallelism
res_getter>, std::tuple<>, std::tuple<ecst::signature::system::impl::mutate_impl<example::c::velocity>, ecst::signature::system::impl::read_impl<example::c::acceleration>
```

«Compile-time» ECS – clang++ errors

```
pres_code.cpp:116:70: error: no member named '_x' in 'example::c::position'
                       const auto& p0 = data.get(ct::position, eid)._x;
                                        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ^
/home/vittorioromeo/OHWorkspace/ecs_thesis/project/include/ecst/context/./system/./instance/instance.inl:112:32: note: in ins
      'example::s::render_colored_circle::process(ecst::context::system::impl::execute_data<ecst::signature::system::impl::da
      ecst::inner_parallelism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn
      std::tuple<ecst::signature::system::impl::read_impl<example::c::circle>, ecst::signature::system::impl::read_impl<examp
      ecst::signature::system::output::impl::o_data<std::vector<sf::Vertex, std::allocator<sf::Vertex> > > >, ecst::context::
      ecst::settings::impl::fixed_impl<std::integral_constant<unsigned long, 20000> >, std::tuple<ecst::signature::component:
      ecst::signature::component::impl::data<ecst::signature::component::impl::tag_impl<example::c::velocity> >, ecst::signat
      ecst::signature::component::impl::data<ecst::signature::component::impl::tag_impl<example::c::color> >, ecst::signature
      ecst::signature::component::impl::data<ecst::signature::component::impl::tag_impl<example::c::mass> > >, std::tuple<ecs
      ecst::inner_parallelism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn
      ecst::signature::system::impl::read_impl<example::c::acceleration> >, ecst::signature::system::output::impl::o_none>, e
      ecst::inner_parallelism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn
      std::tuple<ecst::signature::system::impl::mutate_impl<example::c::position>, ecst::signature::system::impl::read_impl<e
      ecst::signature::system::impl::data<ecst::signature::system::impl::tag_impl<example::s::keep_in_bounds>, ecst::inner_pa
      std::tuple<ecst::signature::system::impl::tag_impl<example::s::velocity> >, std::tuple<ecst::signature::system::impl::m
      ecst::signature::system::impl::read_impl<example::c::circle> >, ecst::signature::system::output::impl::o_none>, ecst::s
      ecst::inner_parallelism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn
      std::tuple<ecst::signature::system::impl::read_impl<example::c::position>, ecst::signature::system::impl::read_impl<exa
      ecst::signature::system::impl::data<ecst::signature::system::impl::tag_impl<example::s::filler>, ecst::inner_parallelis
      ecst::signature::system::output::impl::o_none>, ecst::signature::system::impl::data<ecst::signature::system::impl::tag_
      ecst::inner_parallelism::strategy::split_evenly_fn::impl::parameters<ecst::inner_parallelism::strategy::split_evenly_fn
      std::tuple<ecst::signature::system::impl::mutate_impl<example::c::velocity>, ecst::signature::system::impl::mutate_impl
      ecst::signature::system::output::impl::o_data<std::vector<example::contact, std::allocator<example::contact> > >, ecs
```

«Compile-time» ECS – camomilla-processed errors





«Compile-time» ECS – **camomilla**-processed **errors**

«Compile-time» ECS – **camomilla**-processed **errors**

http://github.com/SuperV1234/camomilla

Customizable **settings/policy-based** design

- Behavior and storage layouts can be chosen at compile-time.
- Library users can create their own settings easily, including:
    - Scheduling algorithms.
    - Parallelization strategies.
    - Entity and component data structures.
- Focus on **composability**.

```cpp
template <typename TComponent>
class hash_map
{
public:
    using entity_id = ecst::impl::entity_id;
    using component_type = TComponent;

    struct metadata
    {
    };

private:
    std::unordered_map<sz_t, TComponent> _data;

    auto valid_index(sz_t i) const noexcept
    {
        return _data.count(i) > 0;
    }

    auto entity_id_to_index(entity_id eid) const noexcept
    {
        return vrmc::to_sz_t(eid);
    }

    template <typename TSelf>
    decltype(auto) get_impl(
        TSelf&& self, entity_id eid, const metadata&) noexcept
    {
        auto i = self.entity_id_to_index(eid);
```

Syntax: independent from settings

- Chosen settings and strategies do not affect the syntax of the application logic.

```
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

```
proxy.execute_systems_overload( // .
    [](s::acceleration& s, auto& data)
    {
        s.process(data);
    },
    [](s::velocity& s, auto& data)
    {
        s.process(data);
    },
    [&window](s::rendering& s, auto& data)
    {
        s.process(window, data);
    });
```

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
    - **Outer**: independent system chains can run in separate parallel tasks.
    - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
    - **Outer**: independent system chains can run in separate parallel tasks.
    - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

Multithreading support

- Enabled by default, customizable and can be optionally disabled.
- Two levels of parallelism:
  - **Outer**: independent system chains can run in separate parallel tasks.
  - **Inner**: system logic can be independently performed on entity subsets in separate tasks.

«Type-value encoding»

- Types can be wrapped into values and vice versa.
- **Greatly** simplifies metaprogramming and compile-time computations.
- Core principle of **`boost::hana`**.

```
constexpr auto parallelism_policy =
    ipc::none_below_threshold::v(sz_v<10000>,
        ips::split_evenly_fn::v_cores()
    );
```

«Type-value encoding»

- Types can be wrapped into values and vice versa.
- **Greatly** simplifies metaprogramming and compile-time computations.
- Core principle of **`boost::hana`**.

```
constexpr auto parallelism_policy =
    ipc::none_below_threshold::v(sz_v<10000>,
        ips::split_evenly_fn::v_cores()
    );
```

Number wrapped in a compile-time type.
Can be passed around as a value at compile-time.

«Type-value encoding»

- Types can be wrapped into values and vice versa.
- **Greatly** simplifies metaprogramming and compile-time computations.
- Core principle of **`boost::hana`**.

```
constexpr auto parallelism_policy =
    ipc::none_below_threshold::v(sz_v<10000>,
        ips::split_evenly_fn::v_cores()
    );
```

«Type-value encoding»

- Types can be wrapped into values and vice versa.
- **Greatly** simplifies metaprogramming and compile-time computations.
- Core principle of **`boost::hana`**.

```
constexpr auto parallelism_policy =
    ipc::none_below_threshold::v(sz_v<10000>,
        ips::split_evenly_fn::v_cores()
    );
```

Type wrapped into a `constexpr` value. Can be passed around as a value at compile-time.

«Type-value encoding»

- Types can be wrapped into values and vice versa.
- **Greatly** simplifies metaprogramming and compile-time computations.
- Core principle of **`boost::hana`**.

```
constexpr auto parallelism_policy =
    ipc::none_below_threshold::v(sz_v<10000>,
        ips::split_evenly_fn::v_cores()
        );
```

«Type-value encoding»

```cpp
/// @brief Given a list of system tags, returns a list of system IDs.
template <typename TSystemSignatureList, typename TSystemTagList>
constexpr auto tag_list_to_id_list(
    TSystemSignatureList ssl, TSystemTagList stl)
{
    return boost::hana::transform(stl, [ssl](auto x)
        {
            return signature_list::system::id_by_tag(ssl, x);
        });
}
```

«Type-value encoding»

```
/// @brief Given a list of system tags, returns a list of system IDs.
template <typename TSystemSignatureList, typename TSystemTagList>
constexpr auto tag_list_to_id_list(
    TSystemSignatureList ssl, TSystemTagList stl)
{
    return boost::hana::transform(stl, [ssl](auto x)
        {
            return signature_list::system::id_by_tag(ssl, x);
        });
}
```

«Type-value encoding»

```cpp
/// @brief Given a list of system tags, returns a list of system IDs.
template <typename TSystemSignatureList, typename TSystemTagList>
constexpr auto tag_list_to_id_list(
    TSystemSignatureList ssl, TSystemTagList stl)
{
    return boost::hana::transform(stl, [ssl](auto x)
        {
            return signature_list::system::id_by_tag(ssl, x);
        });
}
```

«Type-value encoding»

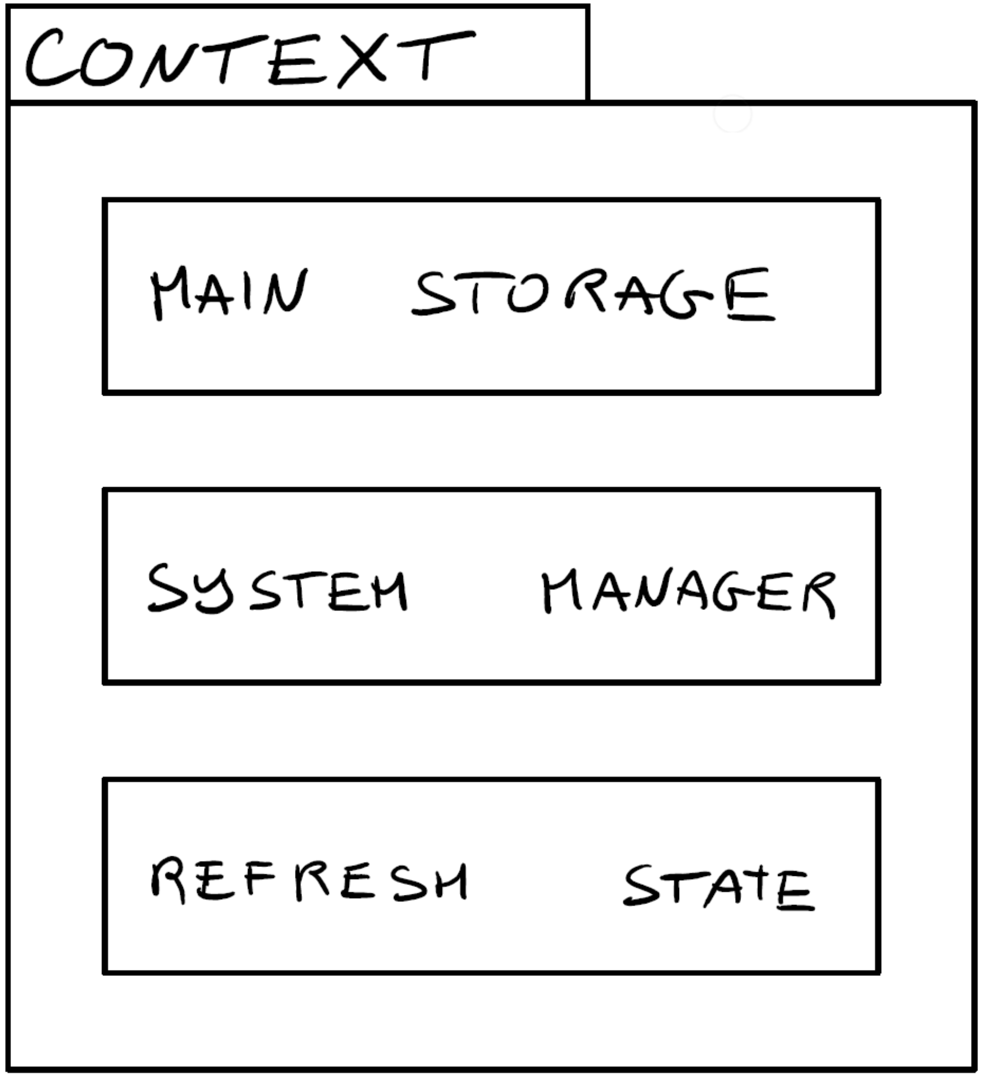
```cpp
/// @brief Given a list of system tags, returns a list of system IDs.
template <typename TSystemSignatureList, typename TSystemTagList>
constexpr auto tag_list_to_id_list(
    TSystemSignatureList ssl, TSystemTagList stl)
{
    return boost::hana::transform(stl, [ssl](auto x)
        {
            return signature_list::system::id_by_tag(ssl, x);
        });
}
```

Code example.

Simple particle simulation implemented using ECST.

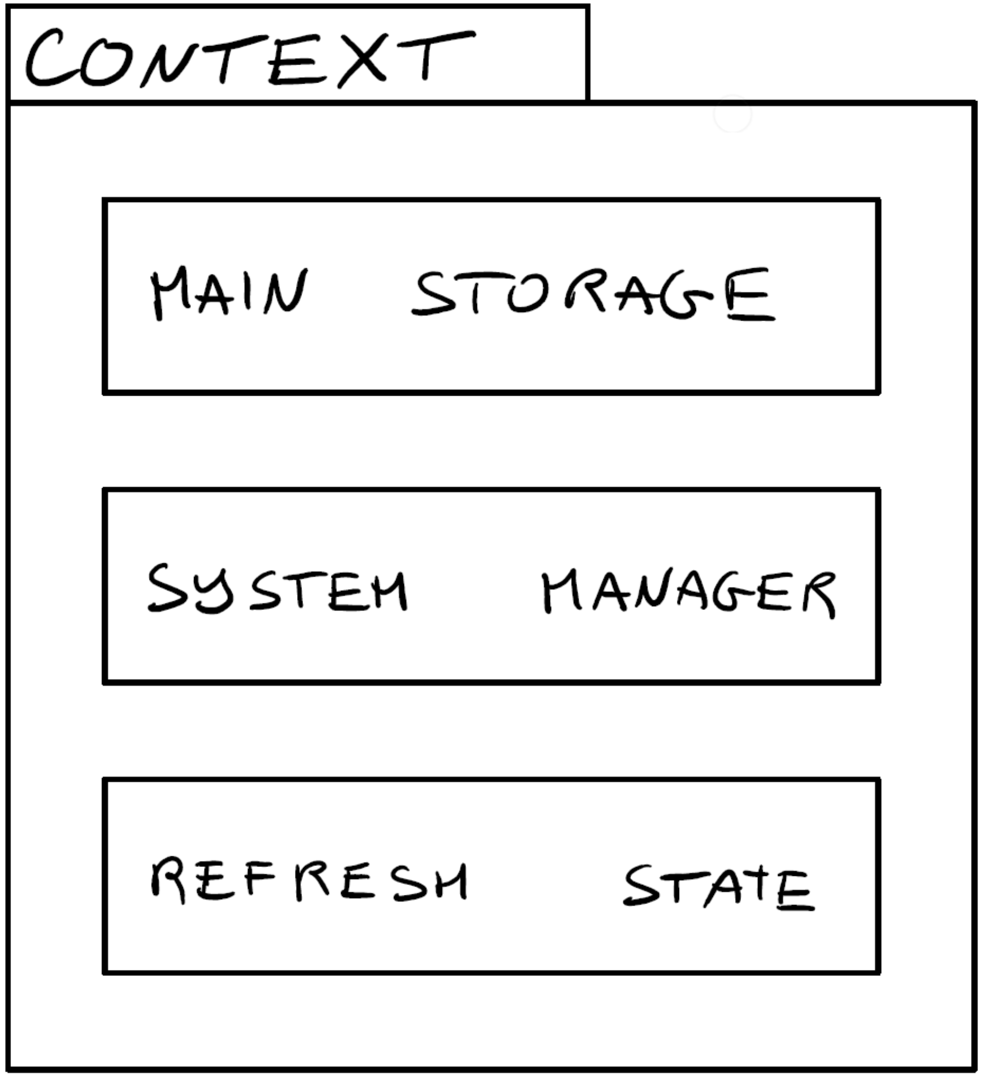
Architecture of **ECST**.

"Components" of the entity component system.



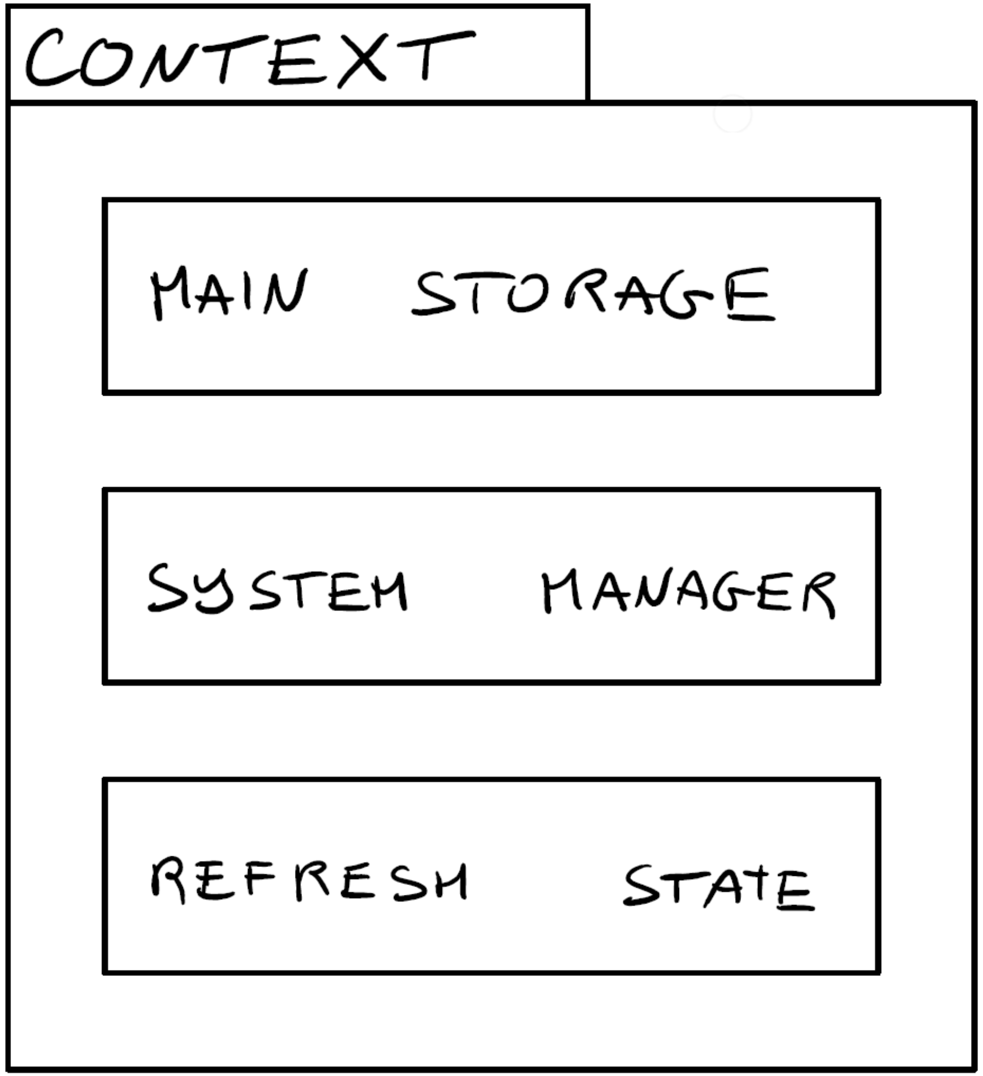

```
constexpr auto component_signatures = setup_components();
constexpr auto system_signatures = setup_systems();
constexpr auto context_settings =
    setup_settings(component_signatures, system_signatures);

auto ctx = ecst::context::make(context_settings);
```

- Movable class that aggregates the main components of **ECST**.




```
constexpr auto component_signatures = setup_components();
constexpr auto system_signatures = setup_systems();
constexpr auto context_settings =
    setup_settings(component_signatures, system_signatures);

auto ctx = ecst::context::make(context_settings);
```

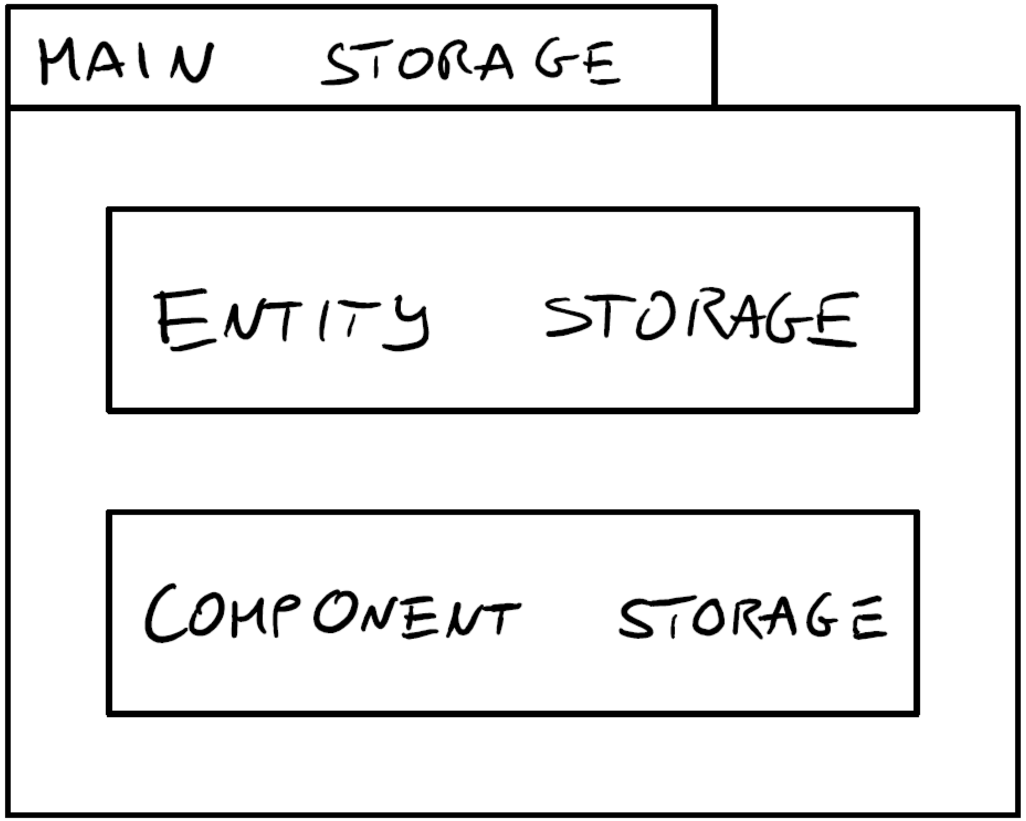
- Movable class that aggregates the main components of **ECST**.




```
constexpr auto component_signatures = setup_components();
constexpr auto system_signatures = setup_systems();
constexpr auto context_settings =
    setup_settings(component_signatures, system_signatures);

auto ctx = ecst::context::make(context_settings);
```

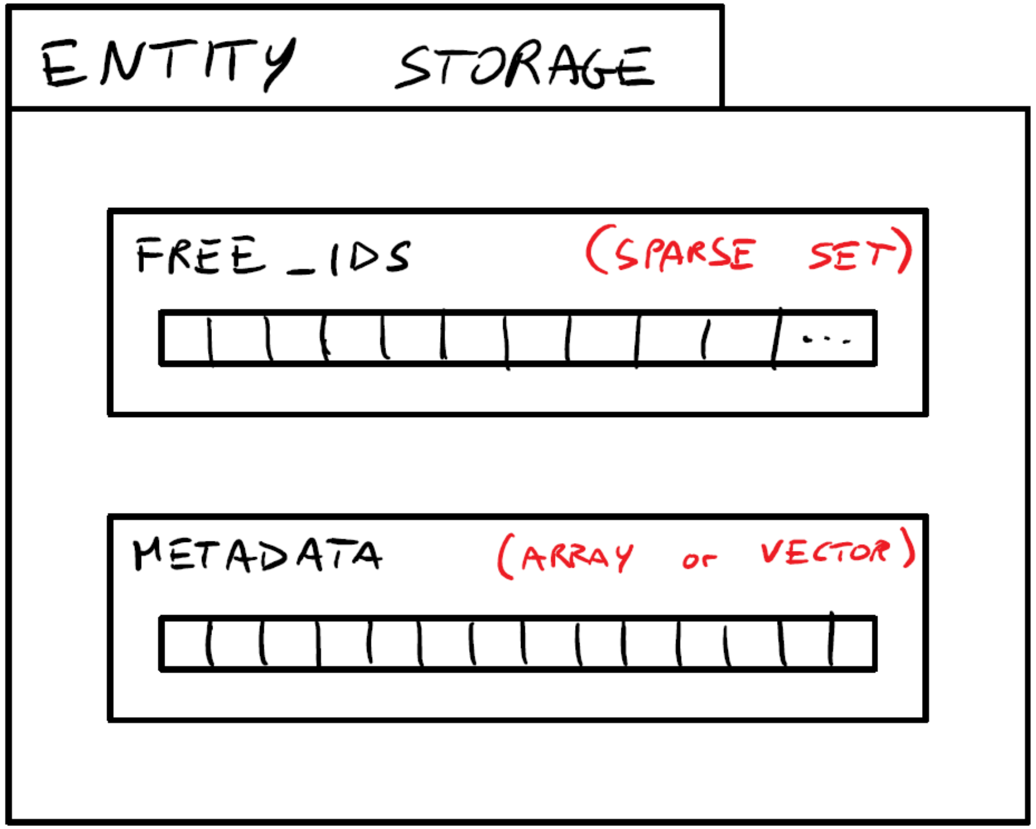
- Movable class that aggregates the main components of **ECST**.

Context -> Main Storage



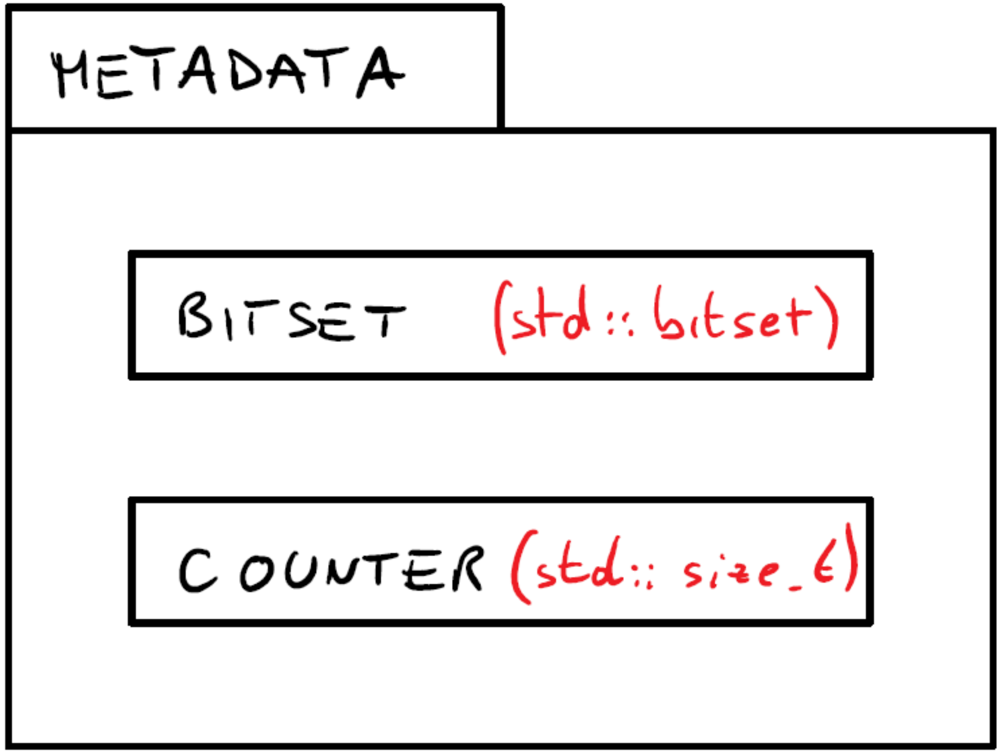
- Encapsulates **entity storage**, which deals with entity IDs and metadata.
- Encapsulates **component storage**, which deals with component data and component storage strategies.

Context -> Main Storage -> Entity Storage



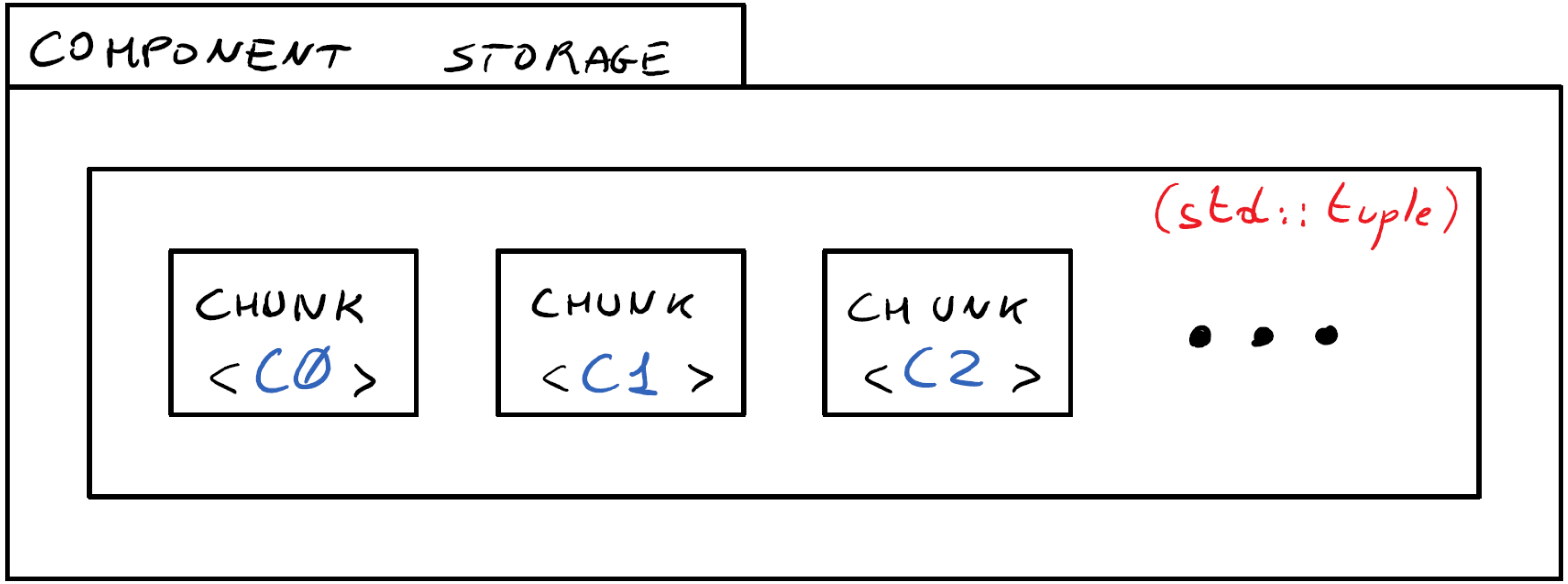
- Keeps track of used and unused **entity IDs**.
- Manages **entity metadata**.
  - Entity metadata may be enriched by component-container-specific metadata at compile-time.

Context -> Main Storage -> Entity Storage -> Metadata



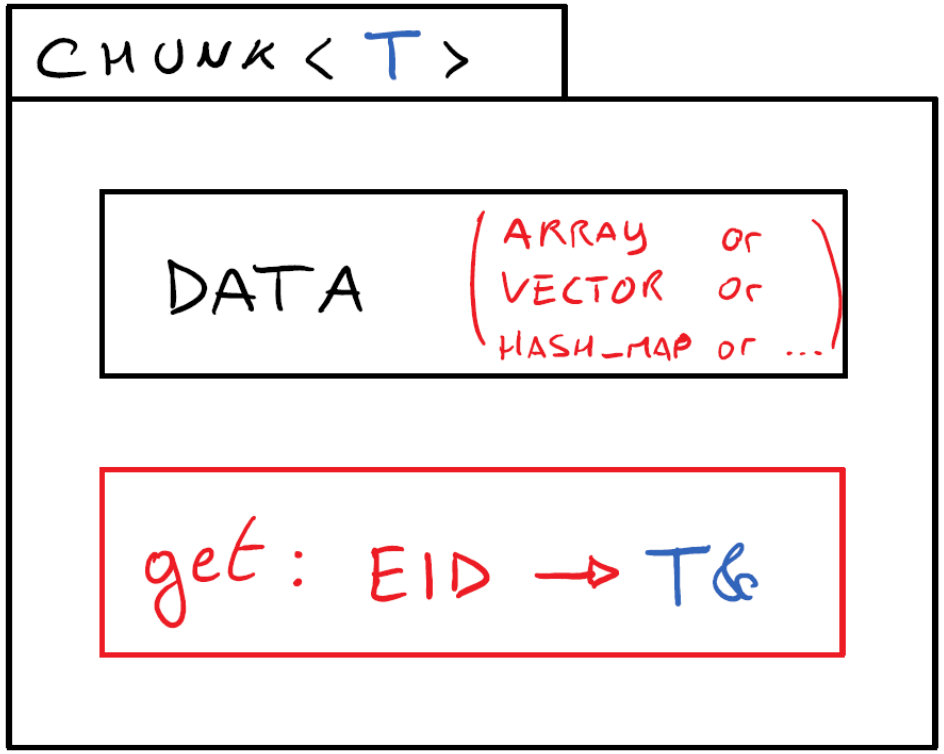
- **Component bitset**: keeps track of current components.
- **Validity counter**: differentiates entity instances which re-use a previous ID. **Handles** check the counter to prevent access to re-used entities.
- Eventual component storage metadata.

Context -> Main Storage -> Component Storage



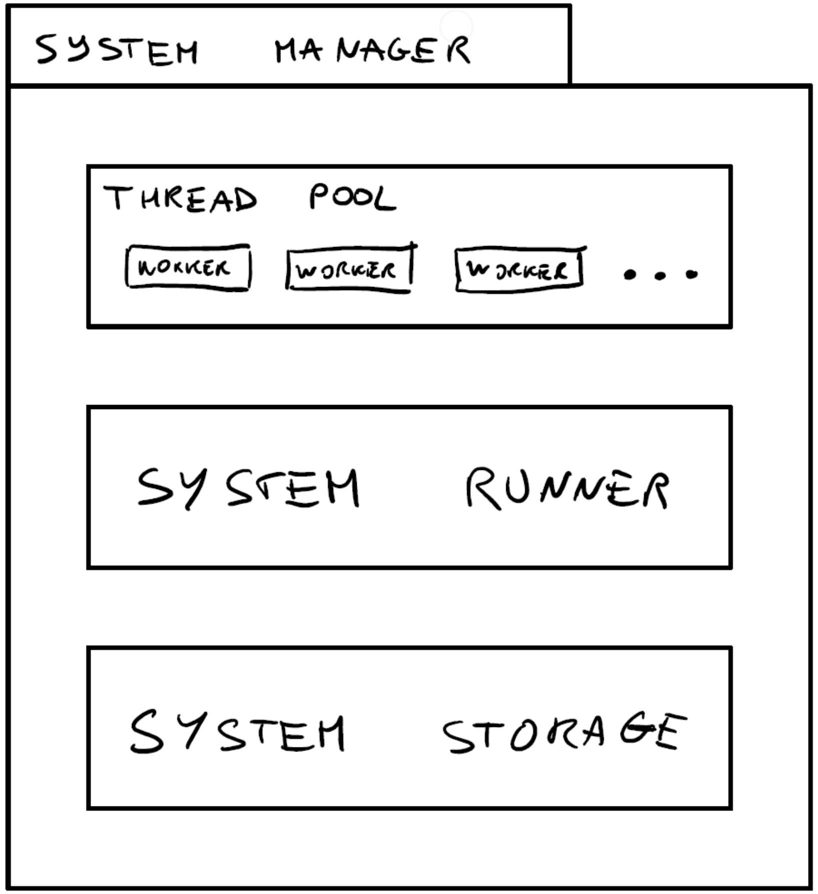
- Stores a **tuple** of component **storage chunks**.
  - Every chunk stores a specific component type in a specific data structure.

Context -> Main Storage -> Component Storage -> Chunk



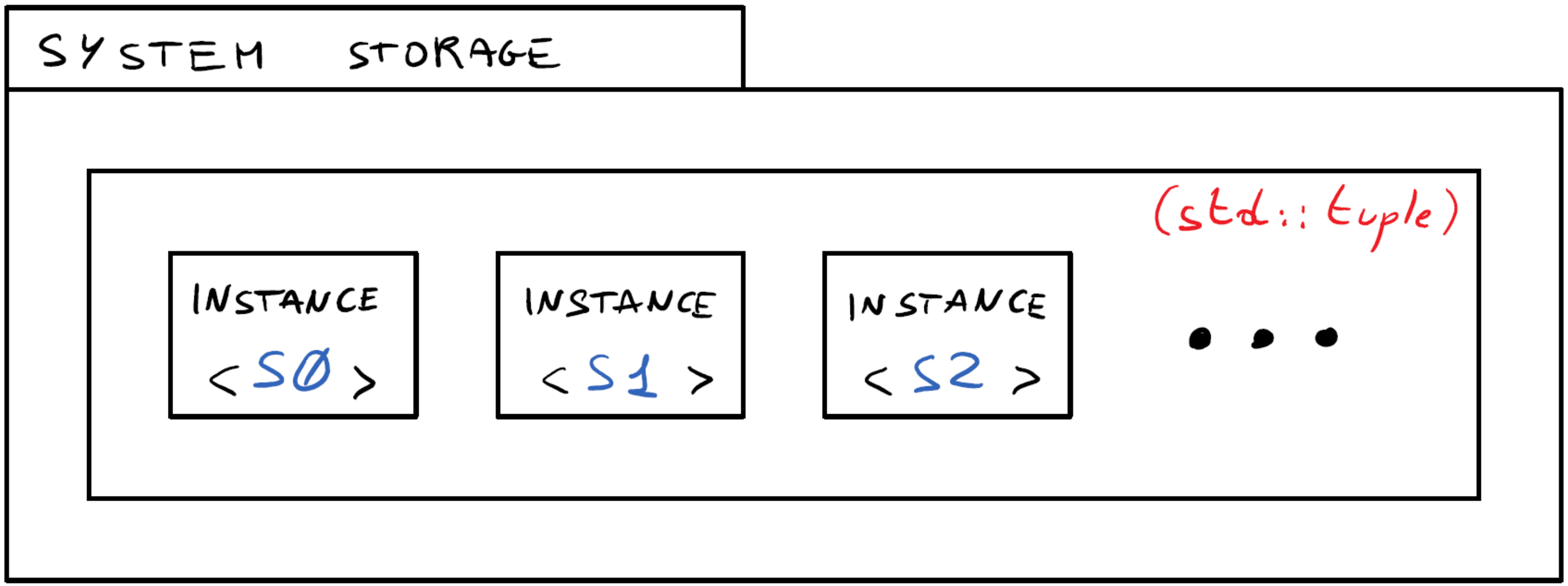
- Manages component data for a particular component type.
- Provides a way to retrieve component data by entity ID.

Context -> System Manager



- Contains a **thread pool**, used to execute system logic.
- Contains a **system runner** which provides a nice interface for a **system scheduler**.
- Stores **system instances** and relative metadata in the **system storage**.

Context -> System Manager -> System Storage

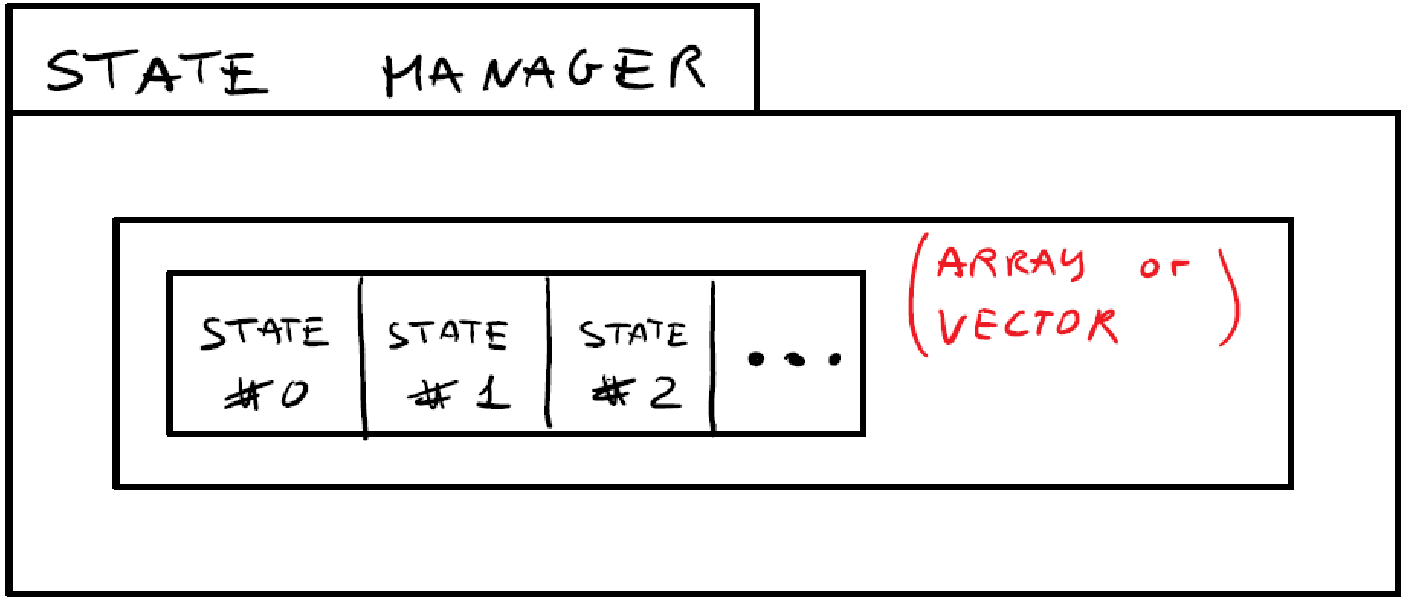


- Stores a **tuple** of **system instances**, one per system type.

Context -> System Manager -> System Storage -> Instance

- Contains an instance of the "real" system type.
- Manages N subtasks states through the **state manager**.
- Keeps track of subscribed entities.
- Contains a **inner parallelism executor** instance.
- Contains a **bitset** representing the required components.

Context -> System Manager -> ... -> Instance -> State Manager



- Manages a number of **states**, that can change at run-time.
  - Every state is connected to a **subtask**.
    - Subtasks are executed in separate threads.

Context -> System Manager -> … -> State Manager -> State

- Contains a set of entity IDs that will be killed during the next **refresh step**.
- Contains a vector of **deferred functions** that will be executed sequentially during the next **refresh step**.
- Optionally contains user-defined **system output data**.

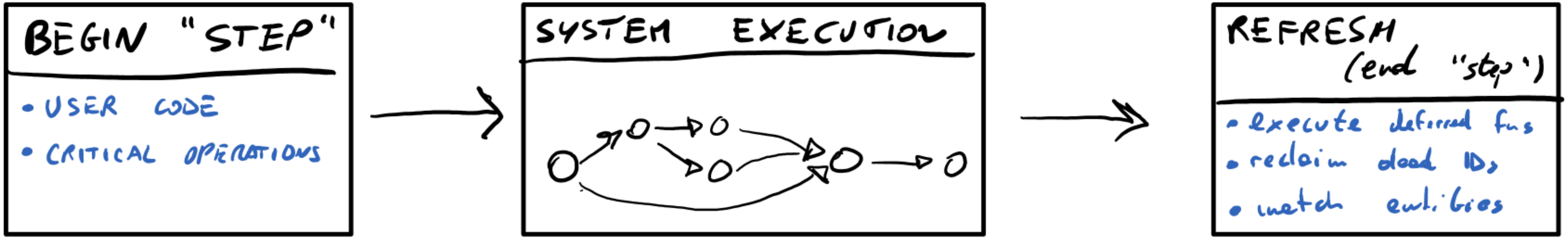
Implementation of **ECST**.

Challenges, execution flow, multithreading, and metaprogramming.

Implementation details overview

- **Execution flow** and **critical operations**.
- Implementation **challenges**.
- **System processing**.
    - Multithreading details.
    - System scheduling *(outer parallelism)*.
    - System inner parallelism.
- Proxies.
- Metaprogramming module examples.

Flow - **overview**



- System execution is preceded by a **step**, where **critical operations** can occur.
- After system execution, a **refresh** takes place, during which:
  - **Deferred critical operations** are executed.
  - **Dead** entity IDs are **reclaimed**.
  - New and modified entities are **matched to systems**.

Flow - **critical operations**

- Operations that need to be executed **sequentially** in a **separate final step** are referred to as **"critical"**.
  - Critical operations can be queued and **deferred** to a later step during system logic execution.
- Critical operations:
  - Creating or destroying an entity.
  - Adding or removing a component to/from an entity.
  - Running the system scheduler.
- Non-critical operations:
  - Analyzing/using a dependency's output system data.
  - Marking an entity as dead.
  - Accessing and mutating existing component data.

```cpp
struct collision_effects
{
    template <typename TData>
    void process(TData& data)
    {
        data.for_entities([&](auto eid)
        {
            const auto& contacts =
                data.get_previous_output(st::collision_detection);

            if(contacts.was_hit(eid))
            {
                auto& h = data.get(ct::health, eid);
                h.damage();

                if(h.dead()) data.kill_entity(eid);

                data.defer([&](auto& proxy)
                {
                    auto e = proxy.create_entity();
                    e.add_component(ct::effect_particle)
                        .set_effect(effects::explosion);
                });
            }
        });
    }
};
```

```
struct collision_effects
{
    template <typename TData>
    void process(TData& data)
    {
        data.for_entities([&](auto eid)
        {
            const auto& contacts =
                data.get_previous_output(st::collision_detection);

            if(contacts.was_hit(eid))
            {
                auto& h = data.get(ct::health, eid);
                h.damage();

                if(h.dead()) data.kill_entity(eid);

                data.defer([&](auto& proxy)
                {
                    auto e = proxy.create_entity();
                    e.add_component(ct::effect_particle)
                        .set_effect(effects::explosion);
                });
            }
        });
    }
};
```

```cpp
struct collision_effects
{
    template <typename TData>
    void process(TData& data)
    {
        data.for_entities([&](auto eid)
        {
            const auto& contacts =
                data.get_previous_output(st::collision_detection);

            if(contacts.was_hit(eid))
            {
                auto& h = data.get(ct::health, eid);
                h.damage();

                if(h.dead()) data.kill_entity(eid);

                data.defer([&](auto& proxy)
                {
                    auto e = proxy.create_entity();
                    e.add_component(ct::effect_particle)
                        .set_effect(effects::explosion);
                });
            }
        });
    }
};
```

```cpp
struct collision_effects
{
    template <typename TData>
    void process(TData& data)
    {
        data.for_entities([&](auto eid)
        {
            const auto& contacts =
                data.get_previous_output(st::collision_detection);

            if(contacts.was_hit(eid))
            {
                auto& h = data.get(ct::health, eid);
                h.damage();

                if(h.dead()) data.kill_entity(eid);

                data.defer([&](auto& proxy)
                {
                    auto e = proxy.create_entity();
                    e.add_component(ct::effect_particle)
                        .set_effect(effects::explosion);
                });
            }
        });
    }
};
```

```cpp
struct collision_effects
{
    template <typename TData>
    void process(TData& data)
    {
        data.for_entities([&](auto eid)
        {
            const auto& contacts =
                data.get_previous_output(st::collision_detection);

            if(contacts.was_hit(eid))
            {
                auto& h = data.get(ct::health, eid);
                h.damage();

                if(h.dead()) data.kill_entity(eid);

                data.defer([&](auto& proxy)
                {
                    auto e = proxy.create_entity();
                    e.add_component(ct::effect_particle)
                        .set_effect(effects::explosion);
                });
            }
        });
    }
};
```

```cpp
struct collision_effects
{
    template <typename TData>
    void process(TData& data)
    {
        data.for_entities([&](auto eid)
        {
            const auto& contacts =
                data.get_previous_output(st::collision_detection);

            if(contacts.was_hit(eid))
            {
                auto& h = data.get(ct::health, eid);
                h.damage();

                if(h.dead()) data.kill_entity(eid);

                data.defer([&](auto& proxy)
                {
                    auto e = proxy.create_entity();
                    e.add_component(ct::effect_particle)
                        .set_effect(effects::explosion);
                });
            }
        });
    }
};
```

Flow – user code

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

Flow – user code

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

Flow – user code

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

Flow – user code

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

Flow – user code

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

Flow – user code

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

Flow – user code

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

Flow – user code

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

Challenges

- Efficient management of **entity IDs**.
- Exploiting **compile-time knowledge** to increase performance and safety.
- Executing systems **respecting dependencies** between them and using **parallelism** when possible.
- Processing entities subsets of the same system in different threads.
- Dealing with entity/component addition/removal during system execution.
- Providing a clean and safe interface to the user.

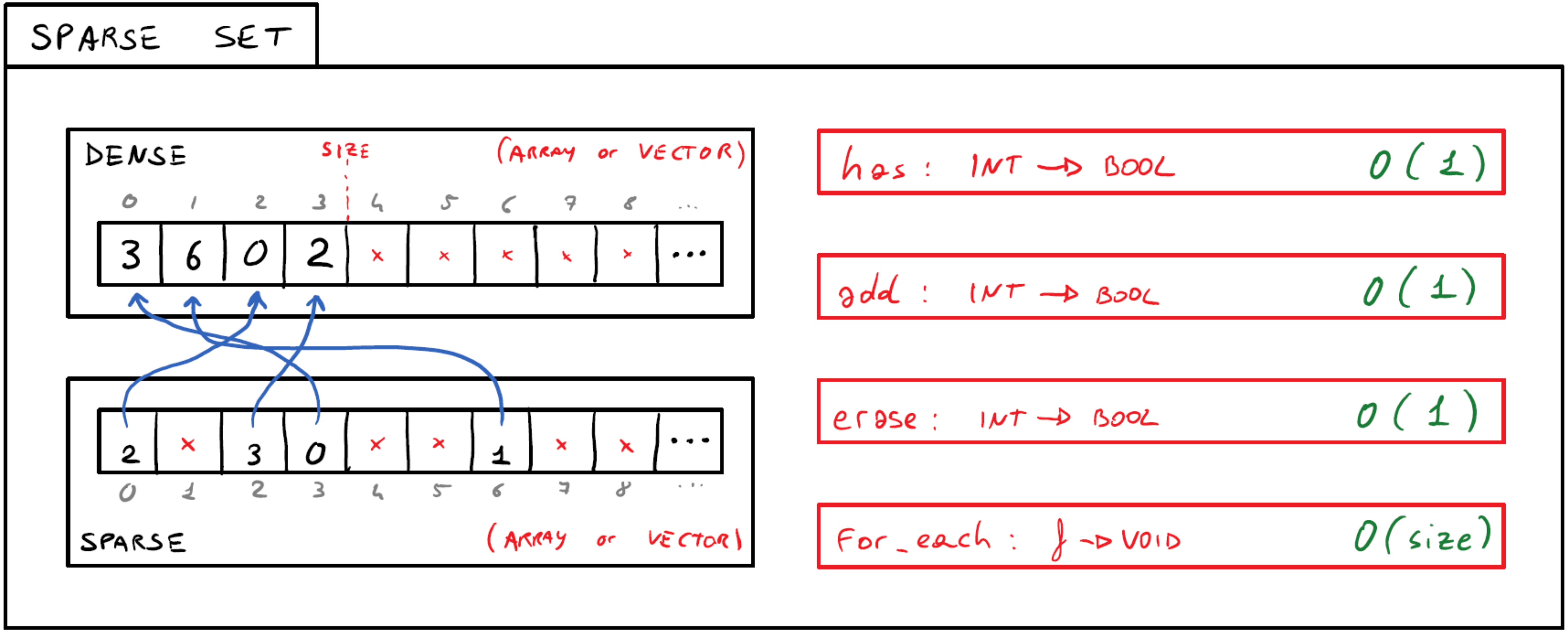
Sparse Integer Sets

- Extremely useful data structure to deal with entity ID management.
- Conceptually represents a set of unsigned integers.
- Allows:
  - `O(1)` test, insertion and removal.
  - `O(k)` iteration, where **k** is the number of integers in the set.

- Used in **system instances** *(subscribed entities)* and in **entity storage** *(available IDs)*.

```cpp
/// @brief Sparse integer set, with fixed array storage.
template <typename T, sz_t TCapacity>
using fixed_array_sparse_set = impl::base_sparse_set<
    impl::sparse_set_settings<
        T,
        impl::sparse_set_storage::fixed_array<T, TCapacity>
    >
>;
```

Sparse Integer Sets



Data structure static dispatching – **user syntax**

- Users can choose settings at compile-time that can restrict the functionality of ECST in order to achieve superior performance.
- Example: **max entity limit**.

```
constexpr auto s =
    ecst::settings::make()
        .allow_inner_parallelism()
        .fixed_entity_limit(sz_v<50000>)
        .component_signatures(csl)
        .system_signatures(ssl)
        .scheduler(st::atomic_counter);
```

```
constexpr auto s =
    ecst::settings::make()
        .allow_inner_parallelism()
        .dynamic_entity_limit()
        .component_signatures(csl)
        .system_signatures(ssl)
        .scheduler(st::atomic_counter);
```

Data structure static dispatching – **user syntax**

- Users can choose settings at compile-time that can restrict the functionality of ECST in order to achieve superior performance.
- Example: **max entity limit**.

```
constexpr auto s =
    ecst::settings::make()
        .allow_inner_parallelism()
        .fixed_entity_limit(sz_v<50000>)
        .component_signatures(csl)
        .system_signatures(ssl)
        .scheduler(st::atomic_counter);
```

```
constexpr auto s =
    ecst::settings::make()
        .allow_inner_parallelism()
        .dynamic_entity_limit()
        .component_signatures(csl)
        .system_signatures(ssl)
        .scheduler(st::atomic_counter);
```

Data structure static dispatching – **option maps**

- The “fluent” syntax is implemented using **option maps**.
- They are implemented on top of `boost::hana::map`.
- They map a *compile-time key* to a *compile-time value* and a *compile-time boolean*.
- The boolean prevents setting the multiple option twice.

Data structure static dispatching – **option maps**

```
namespace keys
{
    constexpr auto threading = sz_v<0>;
    constexpr auto entity_storage = sz_v<1>;
    constexpr auto component_signature_list = sz_v<2>;
    constexpr auto system_signature_list = sz_v<3>;
    constexpr auto scheduler = sz_v<4>;
    constexpr auto refresh_parallelism = sz_v<5>;
}
```

Data structure static dispatching – **option maps**

```
namespace keys
{
    constexpr auto threading = sz_v<0>;
    constexpr auto entity_storage = sz_v<1>;
    constexpr auto component_signature_list = sz_v<2>;
    constexpr auto system_signature_list = sz_v<3>;
    constexpr auto scheduler = sz_v<4>;
    constexpr auto refresh_parallelism = sz_v<5>;
}
```

Every key has a unique type.

Data structure static dispatching – **option maps**

```
template <typename TOptions>
class context_settings
{
    TOptions _map;

    template <typename TKey, typename T>
    constexpr auto change_self(const TKey& key, T x) const noexcept
    {
        auto new_options = _map.set(key, x);
        return data<decltype(new_options)>{};
    }

public:
    template <typename T>
    constexpr auto set_threading(T x) const noexcept
    {
        return change_self(keys::threading, x);
    }
```

Data structure static dispatching – **option maps**

```
template <typename TOptions>
class context_settings
{
    TOptions _map;

    template <typename TKey, typename T>
    constexpr auto change_self(const TKey& key, T x) const noexcept
    {
        auto new_options = _map.set(key, x);
        return data<decltype(new_options)>{};
    }

public:
    template <typename T>
    constexpr auto set_threading(T x) const noexcept
    {
        return change_self(keys::threading, x);
    }
```

Changing an option creates a new type.

Data structure static dispatching – **option maps**

```cpp
template <typename TKey, typename T>
constexpr auto option_map::set(const TKey& key, T x) const noexcept
{
    // Prevent setting same setting twice.
    static_assert(
        bh::second(bh::at_key(_map, key)) == bh::false_c);

    auto new_map = impl::replace(_map,
        bh::make_pair(key, bh::make_pair(x, bh::true_c)));

    return option_map<decltype(new_map)>{};
}
```

Data structure static dispatching – **option maps**

```
template <typename TKey, typename T>
constexpr auto option_map::set(const TKey& key, T x) const noexcept
{
    // Prevent setting same setting twice.
    static_assert(
        bh::second(bh::at_key(_map, key)) == bh::false_c);

    auto new_map = impl::replace(_map,
        bh::make_pair(key, bh::make_pair(x, bh::true_c)));

    return option_map<decltype(new_map)>{};
}
```

Check if the option was previously set.

Data structure static dispatching – **option maps**

```
template <typename TKey, typename T>
constexpr auto option_map::set(const TKey& key, T x) const noexcept
{
    // Prevent setting same setting twice.
    static_assert(
        bh::second(bh::at_key(_map, key)) == bh::false_c);

    auto new_map = impl::replace(_map,
        bh::make_pair(key, bh::make_pair(x, bh::true_c)));

    return option_map<decltype(new_map)>{};
}
```

Data structure static dispatching – **option maps**

```
template <typename TKey, typename T>
constexpr auto option_map::set(const TKey& key, T x) const noexcept
{
    // Prevent setting same setting twice.
    static_assert(
        bh::second(bh::at_key(_map, key)) == bh::false_c);

    auto new_map = impl::replace(_map,
        bh::make_pair(key, bh::make_pair(x, bh::true_c)));

    return option_map<decltype(new_map)>{};
}
```

The boolean value is set to true prevent changing the same option twice accidentally.

Data structure static dispatching – **option maps**

```
template <typename TKey, typename T>
constexpr auto option_map::set(const TKey& key, T x) const noexcept
{
    // Prevent setting same setting twice.
    static_assert(
        bh::second(bh::at_key(_map, key)) == bh::false_c);

    auto new_map = impl::replace(_map,
        bh::make_pair(key, bh::make_pair(x, bh::true_c)));

    return option_map<decltype(new_map)>{};
}
```

Data structure static dispatching – **branching** (1)

```
template <typename TSettings, typename TFFixed, typename TFDynamic>
auto dispatch_on_storage_type(
    TSettings && s, TFFixed && f_fixed, TFDynamic && f_dynamic)
{
    return static_if(s.has_fixed_capacity())
        .then([f_fixed = FWD(f_fixed)](auto xs)
            {
                auto capacity = xs.get_fixed_capacity();
                return f_fixed(capacity);
            })
        .else_([f_dynamic = FWD(f_dynamic)](auto xs)
            {
                auto initial_capacity = xs.get_dynamic_capacity();
                return f_dynamic(initial_capacity);
            })(s);
}
```

Data structure static dispatching – **branching** (2)

```
return static_if(settings.inner_parallelism_allowed())
    .then([&](auto&& xf)
        {
            return this->execute_in_parallel(ctx, FWD(xf));
        })
    .else_([&](auto&& xf)
        {
            return this->execute_single(ctx, FWD(xf));
        })(FWD(f));
```

System settings and dependencies

```cpp
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

```cpp
constexpr auto ssig_velocity =
    ss::make(st::velocity)
        .parallelism(split_evenly_per_core)
        .dependencies(st::acceleration)
        .read(ct::velocity)
        .write(ct::position);
```

```cpp
constexpr auto ssig_render =
    ss::make(st::render)
        .parallelism(none)
        .dependencies(st::velocity)
        .read(ct::position, ct::sprite)
        .output(ss::output<std::vector<Vertex>>);
```

System settings and dependencies

```cpp
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

```cpp
constexpr auto ssig_velocity =
    ss::make(st::velocity)
        .parallelism(split_evenly_per_core)
        .dependencies(st::acceleration)
        .read(ct::velocity)
        .write(ct::position);
```

```cpp
constexpr auto ssig_render =
    ss::make(st::render)
        .parallelism(none)
        .dependencies(st::velocity)
        .read(ct::position, ct::sprite)
        .output(ss::output<std::vector<Vertex>>);
```

System settings and dependencies

```cpp
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

```cpp
constexpr auto ssig_velocity =
    ss::make(st::velocity)
        .parallelism(split_evenly_per_core)
        .dependencies(st::acceleration)
        .read(ct::velocity)
        .write(ct::position);
```

```cpp
constexpr auto ssig_render =
    ss::make(st::render)
        .parallelism(none)
        .dependencies(st::velocity)
        .read(ct::position, ct::sprite)
        .output(ss::output<std::vector<Vertex>>);
```

System settings and dependencies

```
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

```
constexpr auto ssig_velocity =
    ss::make(st::velocity)
        .parallelism(split_evenly_per_core)
        .dependencies(st::acceleration)
        .read(ct::velocity)
        .write(ct::position);
```

```
constexpr auto ssig_render =
    ss::make(st::render)
        .parallelism(none)
        .dependencies(st::velocity)
        .read(ct::position, ct::sprite)
        .output(ss::output<std::vector<Vertex>>);
```

System settings and dependencies

```
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

```
constexpr auto ssig_velocity =
    ss::make(st::velocity)
        .parallelism(split_evenly_per_core)
        .dependencies(st::acceleration)
        .read(ct::velocity)
        .write(ct::position);
```

```
constexpr auto ssig_render =
    ss::make(st::render)
        .parallelism(none)
        .dependencies(st::velocity)
        .read(ct::position, ct::sprite)
        .output(ss::output<std::vector<Vertex>>);
```

System settings and dependencies

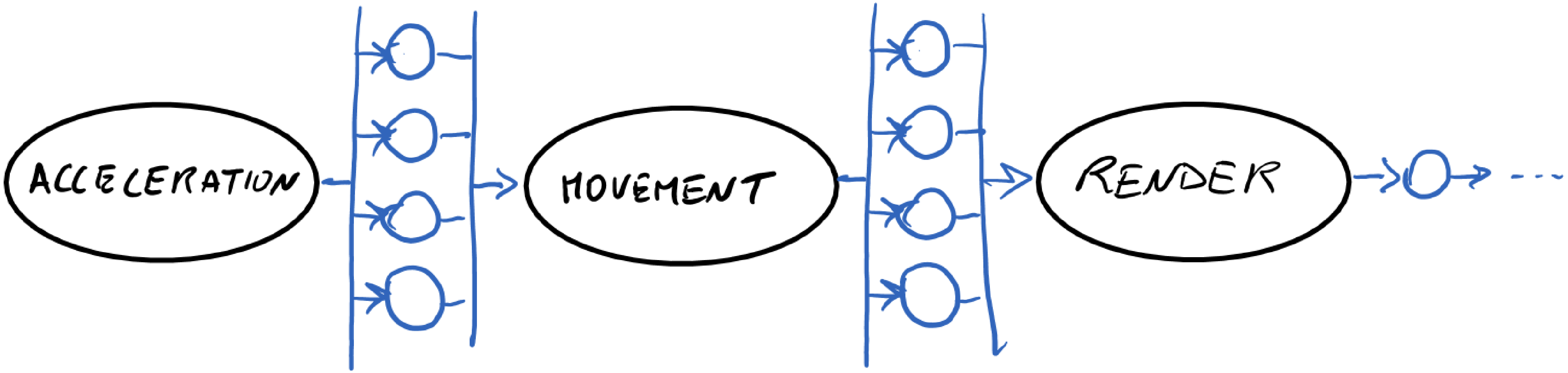
```cpp
constexpr auto ssig_acceleration =
    ss::make(st::acceleration)
        .stateless()
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

```cpp
constexpr auto ssig_velocity =
    ss::make(st::velocity)
        .parallelism(split_evenly_per_core)
        .dependencies(st::acceleration)
        .read(ct::velocity)
        .write(ct::position);
```

```cpp
constexpr auto ssig_render =
    ss::make(st::render)
        .parallelism(none)
        .dependencies(st::velocity)
        .read(ct::position, ct::sprite)
        .output(ss::output<std::vector<Vertex>>);
```

System settings and dependencies

- An implicit **directed acyclic graph** is built thanks to the compile-time system signatures.



System settings and dependencies

```cpp
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

System settings and dependencies

```
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

System settings and dependencies

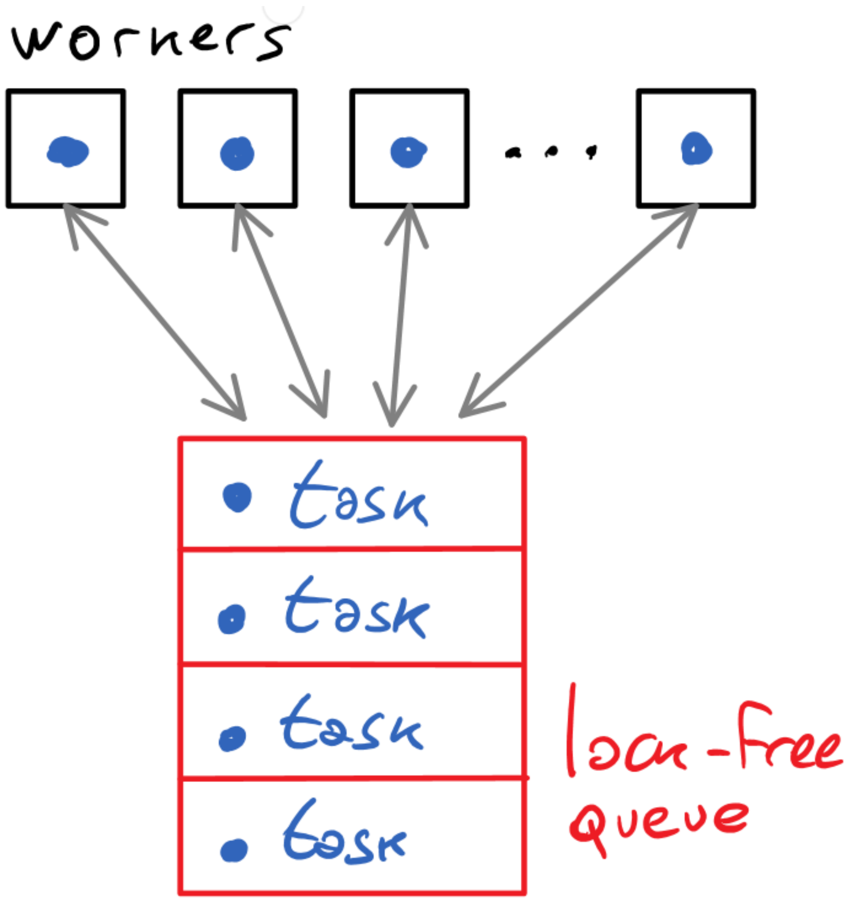
```
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

System settings and dependencies

```
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

Multithreading details

- Avoid **busy waiting**, use **condition variables**.
- Use **thread pooling** with a fast **lock-free concurrent queue**.
- Avoid **futures**, use **atomic counters** with **condition variables** to wait for tasks to complete.



Multithreading details – **lock-free queue**

- `moodycamel::BlockingConcurrentQueue`
  - By **Cameron Desrochers** *(**cameron314** on GitHub).*
    - https://github.com/cameron314/concurrentqueue *(Simplified BSD License)*

```cpp
template <typename TF>
void pool::post(TF&& f)
{
    _queue->enqueue(std::move(f));
}
```

```cpp
void worker::run()
{
    task t;

    while(_running)
    {
        // Dequeue (blocking) and execute.
        _queue->wait_dequeue(_queue.ctok(), t);
        t();
    }

    _finished = true;
}
```

Multithreading details – **synchronization**

- **Condition variables** and **counters** are used to wait for multiple tasks to complete.
  - Used both in **system scheduling** and in **inner parallelism**.
- The `latch` struct aggregates a **mutex**, a **condition variable** and a **counter**.

```
{
    latch l{n};

    l.execute_and_wait_until_zero(cb, []
    {
        run_tasks(n);
    });
}
```

Multithreading details – **synchronization**

- **Condition variables** and **counters** are used to wait for multiple tasks to complete.
  - Used both in **system scheduling** and in **inner parallelism**.
- The `latch` struct aggregates a **mutex**, a **condition variable** and a **counter**.

```
{
    latch l{n};

    l.execute_and_wait_until_zero(cb, []
        {
            run_tasks(n);
        });
}
```

`std::experimental::latch`

Multithreading details – **synchronization**

```cpp
class latch
{
private:
    std::condition_variable _cv;
    std::mutex _mutex;
    std::size_t counter;

public:
    latch(std::size_t initial_count) noexcept;

    void decrement_and_notify_one() noexcept;
    void decrement_and_notify_all() noexcept;

    template <typename TF>
    void execute_and_wait_until_zero(TF&& f);
};
```

- `latch` is essentially a simple abstraction to prevent busy waiting.
- Clean «interface» functions that take it as a parameter are defined for convenience and safety.

Multithreading details – **synchronization**

```
/// @brief Accesses `cv` and `c` through a `lock_guard` on `mutex`, and
/// calls `f(cv, c)`.
template <typename TF>
void access_cv_counter(
    mutex_type& mutex, cv_type& cv, counter_type& c, TF&& f) noexcept
{
    std::lock_guard<std::mutex> l(mutex);
    f(cv, c);

    // Prevent warnings.
    (void)l;
}
```

- Access to the counter is synchronized thanks to a `lock_guard` that locks the `mutex` contained in the counter blocker.
- The passed function will be executed after locking, to prevent mistakes and explicit locking.

Multithreading details – **synchronization**

```
/// @brief Decrements `c` through `mutex`, and calls `cv.notify_one()`.
void decrement_cv_counter_and_notify_one(
    mutex_type& mutex, cv_type& cv, counter_type& c) noexcept
{
    access_cv_counter(mutex, cv, c, [](auto& x_cv, auto& x_c)
        {
            ECST_ASSERT(x_c > 0);
            --x_c;
            x_cv.notify_one();
        });
}
```

- This is how the counter is decremented and the waiting threads are notified.

Multithreading details – **synchronization**

```cpp
/// @brief Decrements `c` through `mutex`, and calls `cv.notify_one()`.
void decrement_cv_counter_and_notify_one(
    mutex_type& mutex, cv_type& cv, counter_type& c) noexcept
{
    access_cv_counter(mutex, cv, c, [](auto& x_cv, auto& x_c)
        {
            ECST_ASSERT(x_c > 0);
            --x_c;
            x_cv.notify_one();
        });
}
```

- This is how the counter is decremented and the waiting threads are notified.

Multithreading details – **synchronization**

```cpp
/// @brief Decrements `c` through `mutex`, and calls `cv.notify_one()`.
void decrement_cv_counter_and_notify_one(
    mutex_type& mutex, cv_type& cv, counter_type& c) noexcept
{
    access_cv_counter(mutex, cv, c, [](auto& x_cv, auto& x_c)
        {
            ECST_ASSERT(x_c > 0);
            --x_c;
            x_cv.notify_one();
        });
}
```

- This is how the counter is decremented and the waiting threads are notified.

Multithreading details – **synchronization**

```
/// @brief Locks `mutex`, executes `f` and waits until `predicate`
/// through `cv`.
template <typename TPredicate, typename TF>
void execute_and_wait_until(mutex_type& mutex, cv_type& cv,
    TPredicate&& predicate, TF&& f) noexcept
{
    std::unique_lock<std::mutex> l(mutex);
    f();
    cv.wait(l, FWD(predicate));
}
```

- The above function is used to wait for a predicate to become true.
- As previously seen, a function is passed to avoid mistakes, explicit locking and explicit waiting.

System scheduling

- Current scheduling policy: **“atomic counter”**.
  - Explored alternative: `when_all(…).then(…)` chain generation.
- The scheduler uses a **remaining systems** `latch` which keeps track of how many systems have been executed.
  - The caller thread blocks until all systems’ executions have been completed.
- Every system instance uses a **remaining dependencies** atomic counter**.**
  - In the beginning, all instances with **zero** dependencies are executed.
  - As soon as an instance is done, all dependents’ counters are decremented.
    - If a dependency counter reaches **zero**, that instance is executed.

System scheduling - **scheduler**

```
template <typename TContext, typename TStartSystemTagList, typename TF>
void atomic_counter::execute(TContext& ctx, TStartSystemTagList sstl, TF& f)
{
    // Number of unique nodes traversed starting from every node in `sstl`.
    constexpr auto chain_size(
        signature_list::system::chain_size(ssl(), sstl));

    // Aggregates the required synchronization objects.
    latch b(chain_size);

    // Starts every independent task and waits until the remaining tasks
    // counter reaches zero.
    b.execute_and_wait_until_zero([&]
        {
            this->start_execution(ctx, sstl, b, f);
        });
}
```

System scheduling - **scheduler**

```
template <typename TContext, typename TStartSystemTagList, typename TF>
void atomic_counter::execute(TContext& ctx, TStartSystemTagList sstl, TF& f)
{
    // Number of unique nodes traversed starting from every node in `sstl`.
    constexpr auto chain_size(
        signature_list::system::chain_size(ssl(), sstl));

    // Aggregates the required synchronization objects.
    latch b(chain_size);

    // Starts every independent task and waits until the remaining tasks
    // counter reaches zero.
    b.execute_and_wait_until_zero([&]
        {
            this->start_execution(ctx, sstl, b, f);
        });
}
```

System scheduling - **scheduler**

```cpp
template <typename TContext, typename TStartSystemTagList, typename TF>
void atomic_counter::execute(TContext& ctx, TStartSystemTagList sstl, TF& f)
{
    // Number of unique nodes traversed starting from every node in `sstl`.
    constexpr auto chain_size(
        signature_list::system::chain_size(ssl(), sstl));

    // Aggregates the required synchronization objects.
    latch b(chain_size);

    // Starts every independent task and waits until the remaining tasks
    // counter reaches zero.
    b.execute_and_wait_until_zero([&]
        {
            this->start_execution(ctx, sstl, b, f);
        });
}
```

```cpp
void task<TDependencyData>::run(TTaskGroup& tg, TLatch& b, TID my_id,
                                TContext& ctx, TF& f)
{
    auto& s_instance(ctx.instance_by_id(my_id));
    s_instance.execute(ctx, f);
    b.decrement_and_notify_one();

    // For every dependent task ID...
    dependency_data().for_dependent_ids([this, &tg, &b, &ctx, &f](auto id) {

        // ...retrieve the corresponding task.
        auto& dt = tg.task_by_id(id);

        // Then, inform the task that one of its dependencies (the
        // current task) has been executed.
        if(dt.dependency_data().decrement_and_check())
        {
            // Recursively run the dependent instance if all dependencies have
            // been processed.
            ctx.post_in_thread_pool([&] { dt.run(tg, b, id, ctx, f); });
        }
    });
}
```

```cpp
void task<TDependencyData>::run(TTaskGroup& tg, TLatch& b, TID my_id,
                                TContext& ctx, TF& f)
{
    auto& s_instance(ctx.instance_by_id(my_id));
    s_instance.execute(ctx, f);
    b.decrement_and_notify_one();

    // For every dependent task ID...
    dependency_data().for_dependent_ids([this, &tg, &b, &ctx, &f](auto id) {

        // ...retrieve the corresponding task.
        auto& dt = tg.task_by_id(id);

        // Then, inform the task that one of its dependencies (the
        // current task) has been executed.
        if(dt.dependency_data().decrement_and_check())
        {
            // Recursively run the dependent instance if all dependencies have
            // been processed.
            ctx.post_in_thread_pool([&] { dt.run(tg, b, id, ctx, f); });
        }
    });
}
```

```cpp
void task<TDependencyData>::run(TTaskGroup& tg, TLatch& b, TID my_id,
                                TContext& ctx, TF& f)
{
    auto& s_instance(ctx.instance_by_id(my_id));
    s_instance.execute(ctx, f);
    b.decrement_and_notify_one();

    // For every dependent task ID...
    dependency_data().for_dependent_ids([this, &tg, &b, &ctx, &f](auto id) {

        // ...retrieve the corresponding task.
        auto& dt = tg.task_by_id(id);

        // Then, inform the task that one of it
        // current task) has been executed.
        if(dt.dependency_data().decrement_and_c
        {
            // Recursively run the dependent instance if all dependencies have
            // been processed.
            ctx.post_in_thread_pool([&] { dt.run(tg, b, id, ctx, f); });
        }
    });
}
```

```cpp
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });
```

System inner parallelism

- Every running system instance can be split in **subtasks**.
  - The instance has a counter of **running subtasks**.
  - As soon as a subtask is finished, the counter is decremented.
  - A **latch** is used to wait for the counter to become **zero**.
- Splitting strategies can be **composed** at compile-time and extended by the users.
  - Strategy executors can be stateful.
    - Motivating example: self-profiling executor that chooses between single-threaded and multi-threaded system execution during application run-time.

System inner parallelism – **parallel execution**

```
template <typename TSettings, typename TSystemSignature>
template <typename TContext, typename TF>
void instance<TSettings, TSystemSignature>::execute_in_parallel(
    TContext & ctx, TF && f
    )
{
    // Aggregates synchronization primitives.
    _parallel_executor.execute(*this, ctx, f);
}
```

- System inner parallel execution is done through a **parallel executor**, which is stored inside the system instance and encapsulates an execution strategy *(or a composition of strategies)*.
- Synchronization is handled inside the executor.

System inner parallelism – **parallel execution**

```
template <typename TSettings, typename TSystemSignature>
template <typename TContext, typename TF>
void instance<TSettings, TSystemSignature>::execute_in_parallel(
    TContext & ctx, TF && f
    )
{
    // Aggregates synchronization primitives.
    _parallel_executor.execute(*this, ctx, f);
}
```

- System inner parallel execution is done through a **parallel executor**, which is stored inside the system instance and encapsulates an execution strategy *(or a composition of strategies)*.
- Synchronization is handled inside the executor.

System inner parallelism – **executor example**

```
template <typename TInstance, typename TContext, typename TF>
void split_evenly::execute(TInstance& inst, TContext& ctx, TF&& f)
{
    constexpr auto split_count = sz_v<parameters::subtask_count()>;

    auto per_split = inst.subscribed_count() / split_count;
    inst.prepare_and_wait_n_subtasks(split_count, [&](auto& b)
        {
            auto run_subtask = [&inst, &b, &ctx, &f](
                auto split_idx, auto xi_begin, auto xi_end)
            {
                // Create looping execution function.
                auto bse = inst.make_bound_slice_executor(
                    b, ctx, split_idx, xi_begin, xi_end, f);

                inst.run_subtask_in_thread_pool(ctx, std::move(bse));
            };

            // Builds and runs the subtasks.
            utils::execute_split_compile_time(inst.subscribed_count(),
                per_split, split_count, run_subtask);
        });
}
```

System inner parallelism – **executor example**

```
template <typename TInstance, typename TContext, typename TF>
void split_evenly::execute(TInstance& inst, TContext& ctx, TF&& f)
{
    constexpr auto split_count = sz_v<parameters::subtask_count()>;

    auto per_split = inst.subscribed_count() / split_count;
    inst.prepare_and_wait_n_subtasks(split_count, [&](auto& b)
        {
            auto run_subtask = [&inst, &b, &ctx, &f](
                auto split_idx, auto xi_begin, auto xi_end)
            {
                // Create looping execution function.
                auto bse = inst.make_bound_slice_executor(
                    b, ctx, split_idx, xi_begin, xi_end, f);

                inst.run_subtask_in_thread_pool(ctx, std::move(bse));
            };

            // Builds and runs the subtasks.
            utils::execute_split_compile_time(inst.subscribed_count(),
                per_split, split_count, run_subtask);
        });
}
```

System inner parallelism – **executor example**

```cpp
template <typename TInstance, typename TContext, typename TF>
void split_evenly::execute(TInstance& inst, TContext& ctx, TF&& f)
{
    constexpr auto split_count = sz_v<parameters::subtask_count()>;

    auto per_split = inst.subscribed_count() / split_count;
    inst.prepare_and_wait_n_subtasks(split_count, [&](auto& b)
        {
            auto run_subtask = [&inst, &b, &ctx, &f](
                auto split_idx, auto xi_begin, auto xi_end)
            {
                // Create looping execution function.
                auto bse = inst.make_bound_slice_executor(
                    b, ctx, split_idx, xi_begin, xi_end, f);

                inst.run_subtask_in_thread_pool(ctx, std::move(bse));
            };

            // Builds and runs the subtasks.
            utils::execute_split_compile_time(inst.subscribed_count(),
                per_split, split_count, run_subtask);
        });
}
```

System inner parallelism – **executor example**

```
template <typename TInstance, typename TContext, typename TF>
void split_evenly::execute(TInstance& inst, TContext& ctx, TF&& f)
{
    constexpr auto split_count = sz_v<parameters::subtask_count()>;

    auto per_split = inst.subscribed_count() / split_count;
    inst.prepare_and_wait_n_subtasks(split_count, [&](auto& b)
        {
            auto run_sub
                auto spl
            {
                // Create
                auto bse
                    b, c

                inst.run
            };

            // Builds and
            utils::execu
                per_spli
        });
}
```

```
template <typename TSettings, typename TSystemSignature>
template <typename TF>
void instance<TSettings, TSystemSignature>::prepare_and_wait_n_subtasks(
    sz_t n, TF && f)
{
    _sm.clear_and_prepare(n);
    counter_blocker b{n};

    // Runs the parallel executor and waits until the remaining subtasks
    // counter is zero.
    execute_and_wait_until_counter_zero(b, [this, &b, &f]
        {
            f(b);
        });
}
```

System inner parallelism – **executor example**

```cpp
template <typename TInstance, typename TContext, typename TF>
void split_evenly::execute(TInstance& inst, TContext& ctx, TF&& f)
{
    constexpr auto split_count = sz_v<parameters::subtask_count()>;

    auto per_split = inst.subscribed_count() / split_count;
    inst.prepare_and_wait_n_subtasks(split_count, [&](auto& b)
        {
            auto run_subtask = [&inst, &b, &ctx, &f](
                auto split_idx, auto xi_begin, auto xi_end)
            {
                // Create looping execution function.
                auto bse = inst.make_bound_slice_executor(
                    b, ctx, split_idx, xi_begin, xi_end, f);

                inst.run_subtask_in_thread_pool(ctx, std::move(bse));
            };

            // Builds and runs the subtasks.
            utils::execute_split_compile_time(inst.subscribed_count(),
                per_split, split_count, run_subtask);
        });
}
```

System inner parallelism – **executor example**

```cpp
template <typename TInstance, typename TContext, typename TF>
void split_evenly::execute(TInstance& inst, TContext& ctx, TF&& f)
{
    constexpr auto split_count = sz_v<parameters::subtask_count()>;

    auto per_split = inst.subscribed_count() / split_count;
    inst.prepare_and_wait_n_subtasks(split_count, [&](auto& b)
        {
            auto run_subtask = [&inst, &b, &ctx, &f](
                auto split_idx, auto xi_begin, auto xi_end)
            {
                // Create looping execution function.
                auto bse = inst.make_bound_slice_executor(
                    b, ctx, split_idx, xi_begin, xi_end, f);

                inst.run_subtask_in_thread_pool(ctx, std::move(bse));
            };

            // Builds and runs the subtasks.
            utils::execute_split_compile_time(inst.subscribed_count(),
                per_split, split_count, run_subtask);
        });
}
```

System inner parallelism – **executor example**

```cpp
template <typename TInstance, typename TContext, typename TF>
void split_evenly::execute(TInstance& inst, TContext& ctx, TF&& f)
{
    constexpr auto split_count = sz_v<parameters::subtask_count()>;

    auto per_split = inst.subscribed_count() / split_count;
    inst.prepare_and_wait_n_subtasks(split_count, [&](auto& b)
        {
            auto run_subtask = [&inst, &b, &ctx, &f](
                auto split_idx, auto xi_begin, auto xi_end)
            {
                // Create looping execution function.
                auto bse = inst.make_bound_slice_executor(
                    b, ctx, split_idx, xi_begin, xi_end, f);

                inst.run_subtask_in_thread_pool(ctx, std::move(bse));
            };

            // Builds and runs the subtasks.
            utils::execute_split_compile_time(inst.subscribed_count(),
                per_split, split_count, run_subtask);
        });
}
```

System inner parallelism – **executor example**

```cpp
template <typename TInstance, typename TContext, typename TF>
void split_evenly::execute(TInstance& inst, TContext& ctx, TF&& f)
{
    constexpr auto split_count = sz_v<parameters::subtask_count()>;

    auto per_split = inst.subscribed_count() / split_count;
    inst.prepare_and_wait_n_subtasks(split_count, [&](auto& b)
        {
            auto run_subtask = [&inst, &b, &ctx, &f](
                auto split_idx, auto xi_begin, auto xi_end)
            {
                // Create looping execution function.
                auto bse = inst.make_bound_slice_executor(
                    b, ctx, split_idx, xi_begin, xi_end, f);

                inst.run_subtask_in_thread_pool(ctx, std::move(bse));
            };

            // Builds and runs the subtasks.
            utils::execute_split_compile_time(inst.subscribed_count(),
                per_split, split_count, run_subtask);
        });
}
```

System inner parallelism – **executor example**

```
template <typename TInstance, typename TContext, typename TF>
void split_evenly::execute(TInstance& inst, TContext& ctx, TF&& f)
{
    constexpr auto split_count = sz_v<parameters::subtask_count()>;

    auto per_split = inst.subscribed_count() / split_count;
    inst.prepare_and_wait_n_subtasks(split_count, [&](auto& b)
        {
            auto run_subtask = [&inst, &b, &ctx, &f](
                auto split_idx, auto xi_begin, auto xi_end)
            {
                // Create looping execution function.
                auto bse = inst.make_bound_slice_executor(
                    b, ctx, split_idx, xi_begin, xi_end, f);

                inst.run_subtask_in_thread_pool(ctx, std::move(bse));
            };

            // Builds and runs the subtasks.
            utils::execute_split_compile_time(inst.subscribed_count(),
                per_split, split_count, run_subtask);
        });
}
```

System inner parallelism - **slicing**

```
template <typename TSettings, typename TSystemSignature>
template <typename TCounterBlocker, typename TContext, typename TF>
auto instance<TSettings, TSystemSignature>::make_bound_slice_executor(
    TCounterBlocker & cb, TContext & ctx, sz_t state_idx, sz_t i_begin,
    sz_t i_end, TF && f) noexcept
{
    return [this, &cb, &ctx, &f, state_idx, i_begin, i_end]
    {
        this->make_slice_executor(cb, ctx, state_idx, i_begin, i_end)(f);
    };
}
```

- The above function binds a range of entities *(subset of subscribed entities)* to an execution callable object.

System inner parallelism - **slicing**

```
template <typename TSettings, typename TSystemSignature>
template <typename TCounterBlocker, typename TContext, typename TF>
auto instance<TSettings, TSystemSignature>::make_bound_slice_executor(
    TCounterBlocker & cb, TContext & ctx, sz_t state_idx, sz_t i_begin,
    sz_t i_end, TF && f) noexcept
{
    return [this, &cb, &ctx, &f, state_idx, i_begin, i_end]
    {
        this->make_slice_executor(cb, ctx, state_idx, i_begin, i_end)(f);
    };
}
```

- The above function binds a range of entities *(subset of subscribed entities)* to an execution callable object.

System inner parallelism - **slicing**

```
template <typename TSettings, typename TSystemSignature>
template <typename TCounterBlocker, typename TContext>
auto ECST_PURE_FN instance<TSettings,
    TSystemSignature>::make_slice_executor(TCounterBlocker & cb,
    TContext & ctx, sz_t state_idx, sz_t i_begin, sz_t i_end) noexcept
{
    return [this, &cb, &ctx, state_idx, i_begin, i_end](auto&& f)
    {
        auto data =
            this->make_entity_range_data(ctx, state_idx, i_begin, i_end);

        // Executes the processing function over the slice of entities.
        this->execute_subtask_and_decrement_counter(cb, data, f);
    };
}
```

- The above function creates a **data proxy** and runs the parallel subtask function passing it as one of the parameters.

System inner parallelism - **slicing**

```
template <typename TSettings, typename TSystemSignature>
template <typename TCounterBlocker, typename TContext>
auto ECST_PURE_FN instance<TSettings,
    TSystemSignature>::make_slice_executor(TCounterBlocker & cb,
    TContext & ctx, sz_t state_idx, sz_t i_begin, sz_t i_end) noexcept
{
    return [this, &cb, &ctx, state_idx, i_begin, i_end](auto&& f)
    {
        auto data =
            this->make_entity_range_data(ctx, state_idx, i_begin, i_end);

        // Executes the processing function over the slice of entities.
        this->execute_subtask_and_decrement_counter(cb, data, f);
    };
}
```

- The above function creates a **data proxy** and runs the parallel subtask function passing it as one of the parameters.

System inner parallelism - **slicing**

```
template <typename TSettings, typename TSystemSignature>
template <typename T
auto ECST_PURE_FN in
    TSystemSignature
    TContext & ctx,
{
    return [this, &c
    {
        auto data =
            this->ma

        // Executes the processing function over the slice of entities.
        this->execute_subtask_and_decrement_counter(cb, data, f);
    };
}
```

```
template <typename TSettings, typename TSystemSignature>
template <typename TCounterBlocker, typename TData, typename TF>
void instance<TSettings,
    TSystemSignature>::execute_subtask_and_decrement_counter( // .
    TCounterBlocker & cb, TData & data, TF && f               // .
    )
{
    f(_system, data);
    decrement_cv_counter_and_notify_all(cb);
}
```

- The above function creates a **data proxy** and runs the parallel subtask function passing it as one of the parameters.

System inner parallelism - **slicing**

```cpp
template <typename TSettings, typename TSystemSignature>
template <typename TCounterBlocker, typename TContext>
auto ECST_PURE_FN instance<TSettings,
    TSystemSignature>::make_slice_executor(TCounterBlocker & cb,
    TContext & ctx, sz_t state_idx, sz_t i_begin, sz_t i_end) noexcept
{
    return [this, &cb, &ctx, state_idx, i_begin, i_end](auto&& f)
    {
        auto data =
            this->make_entity_range_data(ctx, state_idx, i_begin, i_end);

        // Executes the processing function over the slice of entities.
        this->execute_subtask_and_decrement_counter(cb, data, f);
    };
}
```

- The above function creates a **data proxy** and runs the parallel subtask function passing it as one of the parameters.

Proxies

- **Proxies** are used to limit the scope in which **critical operations** can be executed.
- **Data proxies** are used during system execution to provide component data access and system output access.
- **Defer proxies** are used in `defer(…)` calls to allow the user to queue up critical operations that will be executed in the **refresh step**. **Step proxies** derive from defer proxies, and allow the execution of system schedulers.
- **Executor proxies** allow fine-grained execution of system instances.
  - *(e.g. calling a function per subtask, or calling a function pre/post system execution)*

Proxies – **data proxy**

```cpp
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

Proxies – **data proxy**

```
struct acceleration
{
    template <typename TData>
    void process(ft dt, TData& data)
    {
        data.for_entities([&](auto eid)
            {
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;

                v += a * dt;
            });
    }
};
```

Proxies – **defer proxy**

```
data.for_entities([&](auto eid)
    {
        const auto& contacts =
            data.get_previous_output<s::collision_detection>();

        if(contacts.was_hit(eid))
        {
            auto& h = ecst::get<c::health>(data, eid);
            h.damage();

            if(h.dead()) data.kill_entity(eid);

            data.defer([&](auto& proxy)
            {
                auto e = proxy.create_entity();
                auto& ep = e.add_component(ct::effect_particle);
                ep.set_effect(effects::explosion);
            });
        }
    });
```

Proxies – **defer proxy**

```
data.for_entities([&](auto eid)
    {
        const auto& contacts =
            data.get_previous_output<s::collision_detection>();

        if(contacts.was_hit(eid))
        {
            auto& h = ecst::get<c::health>(data, eid);
            h.damage();

            if(h.dead()) data.kill_entity(eid);

            data.defer([&](auto& proxy)
            {
                auto e = proxy.create_entity();
                auto& ep = e.add_component(ct::effect_particle);
                ep.set_effect(effects::explosion);
            });
        }
    });
```

Proxies – **defer proxy**

```
data.for_entities([&](auto eid)
    {
        const auto& contacts =
            data.get_previous_output<s::collision_detection>();

        if(contacts.was_hit(eid))
        {
            auto& h = ecst::get<c::health>(data, eid);
            h.damage();

            if(h.dead()) data.kill_entity(eid);

            data.defer([&](auto& proxy)
            {
                auto e = proxy.create_entity();
                auto& ep = e.add_component(ct::effect_particle);
                ep.set_effect(effects::explosion);
            });
        }
    });
```

Proxies – **step proxy**

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

Proxies – **step proxy**

```
context.step([&](auto& proxy)
{
    proxy.system(st::render).prepare();

    proxy.execute_systems_overload(
        [dt](s::physics& s, auto& data){ s.process(dt, data); },
        [](s::render& s, auto& data){ s.process(data); });

    proxy.for_system_outputs(st::render,
        [&window](auto& s, auto& va)
        {
            window.draw(va.data(), va.size(),
                PrimitiveType::Triangles, RenderStates::Default);
        });
});
```

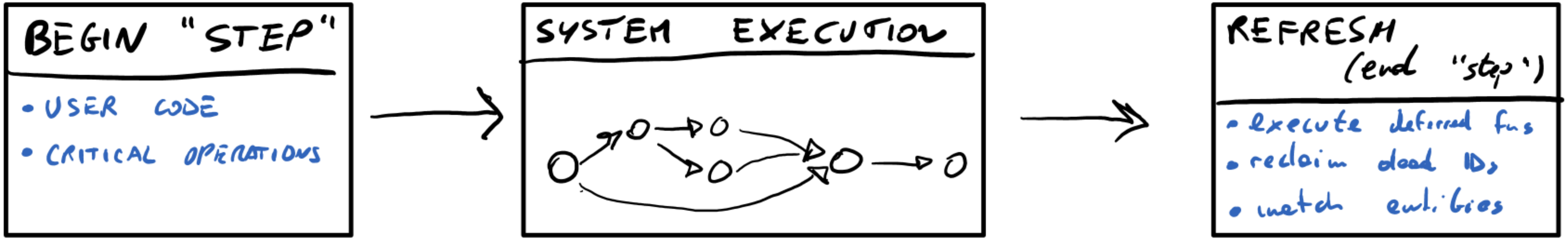
Proxies – **executor proxy**

```
sea::t(st::spatial_partition)
    .detailed_instance([&](auto& i, auto& executor)
        {
            auto& s(i.system());
            s.clear_cells();

            executor.for_subtasks([&s](auto& data){ s.process(data); });

            i.for_outputs([](auto& xs, auto& sp_vector)
                {
                    for(const auto& x : sp_vector) xs.add_sp(x);
                });
        }))
```

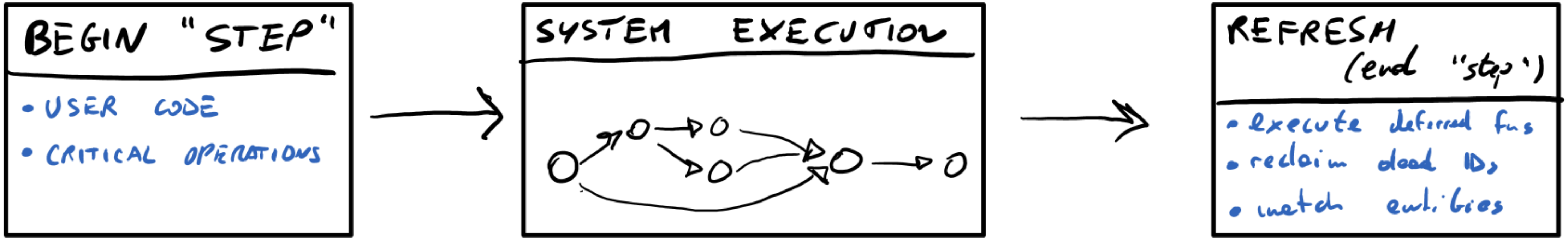
Proxies – executor proxy

```
sea::t(st::spatial_partition)
    .detailed_instance([&](auto& i, auto& executor)
        {
            auto& s(i.system());
            s.clear_cells();

            executor.for_subtasks([&s](auto& data){ s.process(data); });

            i.for_outputs([](auto& xs, auto& sp_vector)
                {
                    for(const auto& x : sp_vector) xs.add_sp(x);
                });
        }))
```

Flow – implementation overview (2)

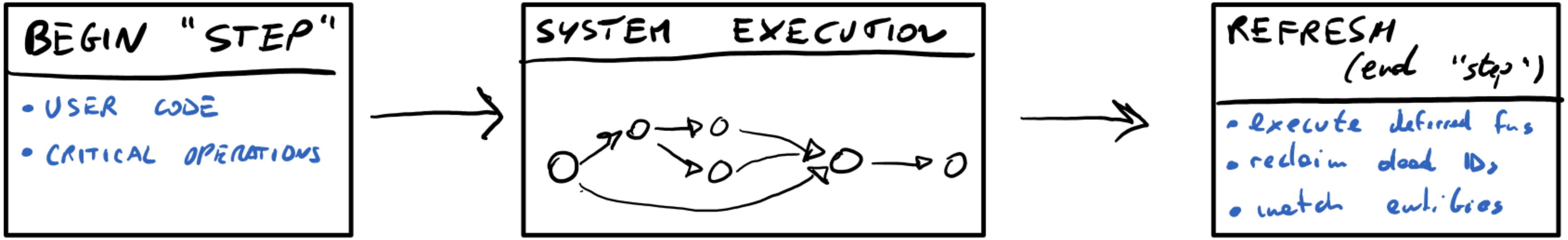


Flow – implementation overview (2)




```
_ctx.step([this, dt](auto& proxy)
    {
        /* ... */
        proxy.execute_systems(/* ... */);
        /* ... */
    };
```

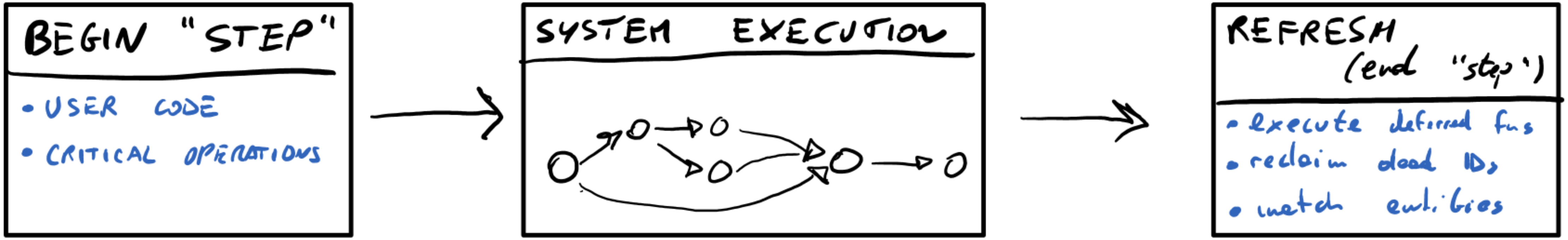
Flow – implementation overview (2)




```cpp
_ctx.step([this, dt](auto& proxy)
    {
        /* ... */
        proxy.execute_systems(/
        /* ... */
    };
```

```cpp
template <typename TSettings>
template <typename TContext, typename TF>
void system_manager<TSettings>::execute_systems(
    TContext& context, TF&& f)
{
    _system_runner.execute(context, FWD(f));
}
```

Flow – implementation overview (2)




```
_ctx.step([this, dt](auto& proxy)
    {
        /* ... */
        proxy.execute_systems(/
        /* ... */
    };
```

```
template <typename TSettings>
template <typename TContext,
void system_manager<TSettings
    TContext& context, TF&& f
{
    _system_runner.execute(co
}
```

```
template <typename TSettings>
void data<TSettings>::refresh_impl()
{
    // Execute deferred functions, filling up the refresh state and
    // allocating memory if necessary.
    refresh_impl_execute_deferred(_refresh_state);

    // Unsubscribe all killed entities from systems.
    refresh_impl_kill_entities(_refresh_state);

    // Match all modified and new entities to systems.
    refresh_impl_match_entities(_refresh_state);
}
```

Flow – “begin” step

```cpp
template <typename TSettings>
template <typename TF>
auto context::data<TSettings>::step(TF&& f)
{
    // Clear refresh state.
    _refresh_state.clear();

    // Build context step proxy.
    step_proxy_type step_proxy{*this, _refresh_state};

    // Execute user-defined step.
    f(step_proxy);

    // Refresh context.
    refresh();
}
```

Flow – “begin” step

```
template <typename TSettings>
template <typename TF>
auto context::data<TSettings>::step(TF&& f)
{
    // Clear refresh state.
    _refresh_state.clear();

    // Build context step proxy.
    step_proxy_type step_proxy{*t

    // Execute user-defined step.
    f(step_proxy);

    // Refresh context.
    refresh();
}
```

```
template <typename TSettings>
class refresh_state
{
private:
    using set_type = dispatch_set<TSettings>;
    set_type _to_match_ids, _to_kill_ids;

public:
    // ...
};
```

Flow – “begin” step

```
template <typename TSettings>
template <typename TF>
auto context::data<TSettings>::step(TF&& f)
{
    // Clear refresh state.
    _refresh_state.clear();

    // Build context step proxy.
    step_proxy_type step_proxy{*this, _refresh_state};

    // Execute user-defined step.
    f(step_proxy);

    // Refresh context.
    refresh();
}
```

Flow – “begin” step

```cpp
template <typename TSettings>
template <typename TF>
auto context::data<TSettings>::step(TF&& f)
{
    // Clear refresh state.
    _refresh_state.clear();

    // Build context step proxy.
    step_proxy_type step_proxy{*this, _refresh_state};

    // Execute user-defined step.
    f(step_proxy);

    // Refresh context.
    refresh();
}
```

Flow – refresh step

```cpp
template <typename TSettings>
void context::data<TSettings>::refresh()
{
    // Execute deferred functions, filling up the refresh state and
    // allocating memory if necessary.
    refresh_impl_execute_deferred(_refresh_state);

    // Unsubscribe all killed entities from systems.
    refresh_impl_kill_entities(_refresh_state);

    // Match all modified and new entities to systems.
    refresh_impl_match_entities(_refresh_state);
}
```

Flow – refresh step – execute deferred functions

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_execute_deferred(TRefreshState& rs)
{
    defer_proxy_type defer_proxy{*this, rs};

    for_systems_sequential([&defer_proxy](auto& system)
        {
            system.for_states([&defer_proxy](auto& state)
                {
                    // The execution of deferred functions fills the
                    // refresh state and alters the context state.
                    state.execute_deferred_fns(defer_proxy);
                });
        });
}
```

Flow – refresh step – execute deferred functions

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_execute_deferred(TRefreshState& rs)
{
    defer_proxy_type defer_proxy{*this, rs};

    for_systems_sequential([&defer_proxy](auto& system)
        {
            system.for_states([&defer_proxy](auto& state)
                {
                    // The execution of deferred functions fills the
                    // refresh state and alters the context state.
                    state.execute_deferred_fns(defer_proxy);
                });
        });
}
```

Flow – refresh step – execute deferred functions

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_execute_deferred(TRefreshState& rs)
{
    defer_proxy_type defer_proxy{*this, rs};

    for_systems_sequential([&defer_proxy](auto& system)
        {
            system.for_states([&defer_proxy](auto& state)
                {
                    // The execution of deferred functions fills the
                    // refresh state and alters the context state.
                    state.execute_deferred_fns(defer_proxy);
                });
        });
}
```

Flow – refresh step – reclaim dead entities

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_kill_entities(TRefreshState& rs)
{
    for_systems_sequential([&rs](auto& system)
        {
            system.for_states([&rs](auto& state)
                {
                    state.for_to_kill([&](auto eid){ rs.add_to_kill(eid); });
                });
        });

    for_systems _parallel ([this, &rs](auto& system)
        {
            rs.for_to_kill([&system](auto eid){ system.unsubscribe(eid); });
        });

    rs.for_to_kill([this](auto eid){ this->reclaim(eid); });
}
```

Flow – refresh step – reclaim dead entities

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_kill_entities(TRefreshState& rs)
{
    for_systems_sequential([&rs](auto& system)
        {
            system.for_states([&rs](auto& state)
                {
                    state.for_to_kill([&](auto eid){ rs.add_to_kill(eid); });
                });
        });

    for_systems _parallel ([this, &rs](auto& system)
        {
            rs.for_to_kill([&system](auto eid){ system.unsubscribe(eid); });
        });

    rs.for_to_kill([this](auto eid){ this->reclaim(eid); });
}
```

Flow – **refresh step – reclaim dead entities**

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_kill_entities(TRefreshState& rs)
{
    for_systems_sequential([&rs](auto& system)
        {
            system.for_states([&rs](auto& state)
                {
                    state.for_to_kill([&](auto eid){ rs.add_to_kill(eid); });
                });
        });

    for_systems _parallel ([this, &rs](auto& system)
        {
            rs.for_to_kill([&system](auto eid){ system.unsubscribe(eid); });
        });

    rs.for_to_kill([this](auto eid){ this->reclaim(eid); });
}
```

Flow – refresh step – reclaim dead entities

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_kill_entities(TRefreshState& rs)
{
    for_systems_sequential([&rs](auto& system)
        {
            system.for_states([&rs](auto& state)
                {
                    state.for_to_kill([&](auto eid){ rs.add_to_kill(eid); });
                });
        });

    for_systems _parallel ([this, &rs](auto& system)
        {
            rs.for_to_kill([&system](auto eid){ system.unsubscribe(eid); });
        });

    rs.for_to_kill([this](auto eid){ this->reclaim(eid); });
}
```

Flow – refresh step – match entities to systems

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_match_entities(TRefreshState& rs)
{
    for_systems _parallel ([this, &rs](auto& system)
        {
            rs.for_to_match([this, &system](auto eid)
                {
                    // Get entity metadata.
                    auto& em(this->metadata(eid));

                    // Check if the bitset matches the system.
                    if(system.matches_bitset(em.bitset()))
                        system.subscribe(eid);
                    else
                        system.unsubscribe(eid);
                });
        });
}
```

Flow – refresh step – match entities to systems

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_match_entities(TRefreshState& rs)
{
    for_systems _parallel ([this, &rs](auto& system)
    {
        rs.for_to_match([this, &system](auto eid)
        {
            // Get entity metadata.
            auto& em(this->metadata(eid));

            // Check if the bitset matches the system.
            if(system.matches_bitset(em.bitset()))
                system.subscribe(eid);
            else
                system.unsubscribe(eid);
        });
    });
}
```

Flow – refresh step – match entities to systems

```
template <typename TSettings>
template <typename TRefreshState>
void context::data<TSettings>::refresh_impl_match_entities(TRefreshState& rs)
{
    for_systems _parallel ([this, &rs](auto& system)
        {
            rs.for_to_match([this, &system](auto eid)
                {
                    // Get entity metadata.
                    auto& em(this->metadata(eid));

                    // Check if the bitset matches the system.
                    if(system.matches_bitset(em.bitset()))
                        system.subscribe(eid);
                    else
                        system.unsubscribe(eid);
                });
        });
}
```

```
(s_bs & e_bs) == s_bs
```

Metaprogramming – forcing constexpr

```cpp
namespace impl
{
    template <typename TList, typename TComparer>
    auto find_if_impl(TList l, TComparer c)
    {
        auto res = find_first_index_of_matching(l, c);
        return static_if(is_null(res))
            .then([](auto){ return null_v; })
            .else_([=](auto y){ return at(l, y); })(res);
    }
}

template <typename TList, typename TComparer>
constexpr auto find_if(TList l, TComparer c)
{
    return decltype(impl::find_if_impl(l, c)){};
}
```

Metaprogramming – **forcing constexpr**

```
namespace impl
{
    template <typename TList, typename TComparer>
    auto find_if_impl(TList l, TComparer c)
    {
        auto res = find_first_index_of_matching(l, c);
        return static_if(is_null(res))
            .then([](auto){ return null_v; })
            .else_([=](auto y){ return at(l, y); })(res);
    }
}

template <typename TList, typename TComparer>
constexpr auto find_if(TList l, TComparer c)
{
    return decltype(impl::find_if_impl(l, c)){};
}
```

Metaprogramming – **forcing constexpr**

```
namespace impl
{
    template <typename TList, typename TComparer>
    auto find_if_impl(TList l, TComparer c)
    {
        auto res = find_first_index_of_matching(l, c);
        return static_if(is_null(res))
            .then([](auto){ return null_v; })
            .else_([=](auto y){ return at(l, y); })(res);
    }
}

template <typename TList, typename TComparer>
constexpr auto find_if(TList l, TComparer c)
{
    return decltype(impl::find_if_impl(l, c)){};
}
```

Metaprogramming – **forcing constexpr**

```
namespace impl
{
    template <typename TList, typename TComparer>
    auto find_if_impl(TList l, TComparer c)
    {
        auto res = find_first_index_of_matching(l, c);
        return static_if(is_null(res))
            .then([](auto){ return null_v; })
            .else_([=](auto y){ return at(l, y); })(res);
    }
}

template <typename TList, typename TComparer>
constexpr auto find_if(TList l, TComparer c)
{
    return decltype(impl::find_if_impl(l, c)){};
}
```

Conclusion

Resources and future ideas.

Resources – (1)

- https://www.researchgate.net/publication/305730566
  - My thesis!
- http://t-machine.org
  - Articles on **data structures**, **multithreading** and **networking**.
  - Wiki with **ES approaches** and existing implementations.
- http://stackoverflow.com/questions/1901251
  - In-depth analysis of **component-based engine design**.
- http://bitsquid.blogpost.it
  - Articles on **contiguous component data** allocation strategies.
- http://gameprogrammingpatterns.com/component
  - Covers **component-based design** and **entity communication** techniques.
- http://randygaul.net
  - Articles on **component-based design**, covering **communication** and **allocation**.

Resources – (2)

- Existing C++11/14 **ECS** libraries:
  - https://github.com/grandstack/Pronto *(uses compile-time entities).*
  - https://github.com/discoloda/Diana *(uses sparse sets heavily).*
  - https://github.com/alecthomas/entityx *(uses compile-time components and events).*
- https://maikklein.github.io/post/2016-01-14-Entity-Component-System/
  - «The general design of my flawed compile-time entity component system in C++14.»
- https://www.reddit.com/r/gamedev/comments/3nv8uz
  - Discussion on my previous CppCon 2015 ECS talk.

Resources – (3)

- http://stackoverflow.com/questions/35778864/building-asynchronous-future-callback-chain-from-compile-time-dependency-graph
  - Building asynchronous `future` callback chain from compile-time dependency graph (DAG)
  - StackOverflow question I asked regarding the generation of a callback chain for system execution.
- http://cs.stackexchange.com/questions/2524/getting-parallel-items-in-dependency-resolution
  - Executing tasks with dependencies in parallel.

Future ideas – (1)

- Create system hierarchy. *(WIP)*
  - Some systems can act directly on component data *(ignoring entities)*, allowing SIMD operations and in-order component data traversal.
- Match system subtasks to next system subtasks to use partial outputs without merging.

Future ideas – (2)

- Find an alternative to deferred functions.
  - Avoid `std::function` overhead.
  - Proxy object + command queues?
  - Move-only SBO functions?
- Generic "fluent settings" definition library.
  - Define "settings" schemas at compile-time.
  - Focus on constraints and composability.
  - Statically verify settings with nice error messages.
- Avoid latches, use *no-allocation* asynchronous computation chains.
- Allow *"streaming"* system connections.

Questions?

http://vittorioromeo.info

vittorio.romeo@outlook.com

vromeo5@bloomberg.net

http://github.com/SuperV1234

Thank you for attending!