Obervation and Logging
What's going on

- **Agents** move on the field (World)
- At the end of a turn, the **scheduler** informs observers to do their work
- Observers look at the current situation (snapshot)
- They see what the world exposes (WorldInterface) and what the agents expose (AgentInterface)
- They don't see, what's not exposed
- They don't see, how the agents came to their final decisions

**Issue Description**

- **AgentInterface**
- **WorldInterface**
- **Snapshot**
- **Observer**
- **Observable Variable**
An Agent consists of several Models, each with access to the AgentInterface. Within a simulation step, Models use Signals for communication. At the end, writes are synchronized currently, the last write wins. Then, Observers to do their work.

Unobservable Entities:
- Multiple writes to the AgentInterface within the current turn
- Signals... (although more or less public)
- Internal variables
- Evolution of internal variables
- Internal Events (e.g., State Transition)

Note, some Models do not have access to the AgentInterface.
Observation vs. Logging

**Observer**

**Logger**

**SCOPE**

- public
- private

**Notes**

- Good for generic use, such as CollisionDetection
- Currently, use runResult for data exchange

**Control Flow**

- The core is in control of what data is available and when it is analyzed.

**What’s the goal of openPASS?**

- Generate meaningful data for analysis.
- Don’t reinvent the wheel
- Compatibility to standard analytics

**Control Flow**

- The models know what to write and when to write it. Consumers can access data as soon as it is written.

**Notes**

- Currently only used to write runtime application information into the log files (aka CallbackInterface)

**What are the real requirements?**
THE BIGH MESH UP
Combine Observation and Logging in the ObservationInterface

What to do

✔ Resuscitate ObservationInterface
   - PCM Use isolated the Models from the ObservationNetwork
   The ObservationInterface does not offer model specific methods anyhow
   - No configurable assignment of observations to specific modules
   The ModelLibrary can simply forward all ObservationModules

✔ Extend ObservationInterface
   - At least: Insert-Method, e.g.
      Insert(time, agentId, topic, key, value)

Pro
✔ Almost works out of the box

Con
✔ Unclear, to which time-step reported value belongs (out of sync)
✔ Definitely, only a workaround:
   No future-proof architectural strategy and no separation of concerns

Option 1
Minimally invasive
**PUBLISH/SUBSCRIBE PATTERN**

**Introducing PublishInterface**

**What to do**
- ✓ Replace ObservationInterface
- ✓ Extend SimulationCore
- ✓ Ideally, hide logic from Model (see next Slides)

**Pro**
- ✓ Seperated concerns
- ✓ Everyone can publish private data
- ✓ Decoupled producer and consumer

**Con**
- ✓ Decoupled producer and consumer ;)
- ✓ **Definitely, only a workaround:** No future-proof architectural strategy and no seperation of concerns

**Option 2**

Update UnrestrictedModelInterface
Example

Pseudo Code (Auto-Publishing)

```cpp
template<typename T>
class Observable {
  std::list<T> _values;

public:
  explicit Observable(T initialValue) {
    set(initialValue);
  }

  const T get() const {
    return _values.back();
  }

  void set(T value) {
    _values.push_back(value);
  }

  const std::list<T>& values() const {
    return _values;
  }

  Observable& operator=(T value) {
    set(value);
    return *this;
  }

  operator T() const {
    return get();
  }

  bool operator==(const T& lhs) {
    return lhs == get();
  }

  bool operator==(const Observable& lhs) {
    return lhs.get() == get();
  }
};

int main() {
  Observable x{0.0}, y{0.0};

  // assign to local variable
  double a = x;
  // store update (publish)
  x = 12.0;

  if (x == 12.0) {
    // compare to base type
  }

  if (x == y) {
    // compare to observable
  }

  for (auto value : x.values()) {
    // loop history
  }
}
```

Managed by Broker

Might be a request to the Broker

payload <double>
Apache Kafka
- Publish and subscribe to streams of records, similar to a message queue
- Store streams of records in a fault-tolerant durable way
- Process streams of records as they occur

MQTT
- Easy information organization through hierarchical topics, e.g. Deathstar/Laser/Temperature or Deathstar/Laser/*
- OASIS accepted ISO Standard
- Quality of Service implementation

What we should aim for
- Publish/Subscribe System (could also replace Signals)
- A lightweight interface or decorator for publishing
- A new ModelInterface
- Independent logger (consumers)
- Compatibility to persisting (streaming) systems, opening support for consumers with different processing speed (hot/cold paths)

Published Data

Cold Path: e.g. Compare to other runs

Published Data

Hot Path: e.g. Stop because of collision

Option 3
The bigger picture