BehaviorTree.CPP:

Task Planning for Robots and Virtual Agents

DAVIDE FACONTI

What you will learn today

1. What Behavior Trees are
2. About the C++ library **BehaviorTree.CPP**

About me

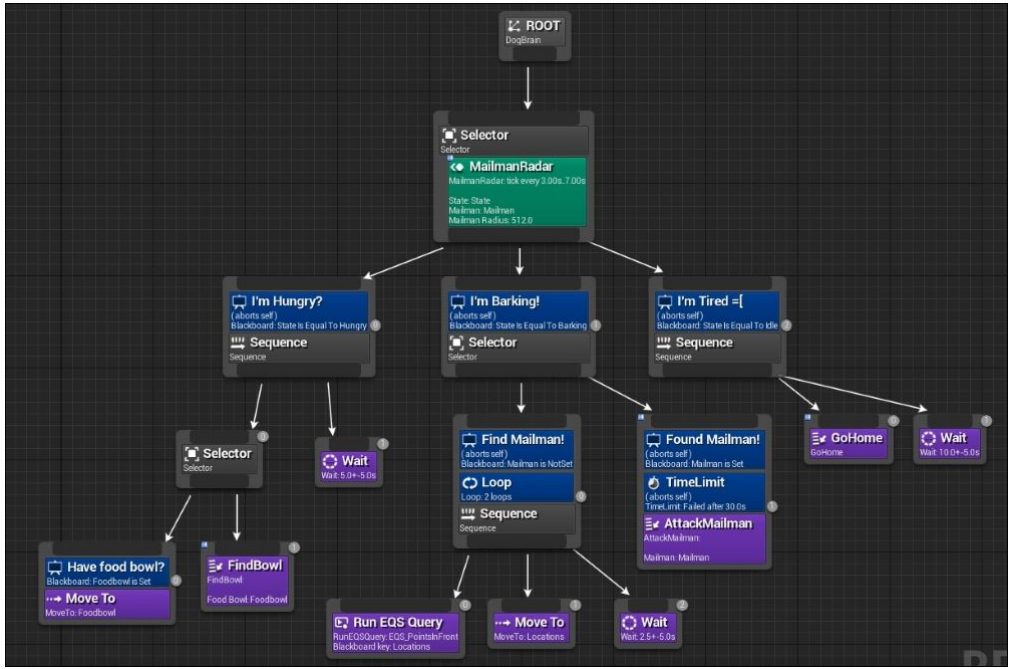
- **Davide Faconti**, nice to meet you :)
- I have been doing robotics for 20 years:
  - **Humanoid robots**: locomotion algorithms, hardware design, simulation
  - **Wheeled robots**: navigation, localization, 3D perception
  - **Robotic arms**: control, motion planning
  - **Tooling**: logging, data visualization
- I love C++, and I learnt a lot thanks to CPPCon videos!

How did I end up developing a Behavior Tree Library?


eurecat
eurecat
Emergency stop

Behavior Trees in the Game Industry

Used for instance in the Unreal Engine



The role of Behavior Trees in robotics

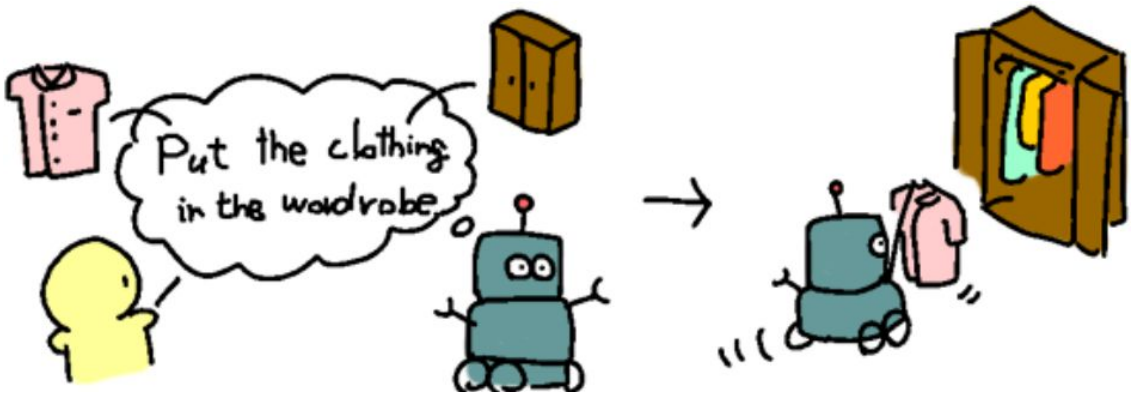
In the last 20 years, the de-facto standard in robotics has been (informally) **Component Based Software Engineering**

- Multi-process and multi-nodes, distributed systems.
- Lot of inter-process communication.
- Publish-Subscribe, Request-Reply.
- Each node of the system has one functionality (or should).

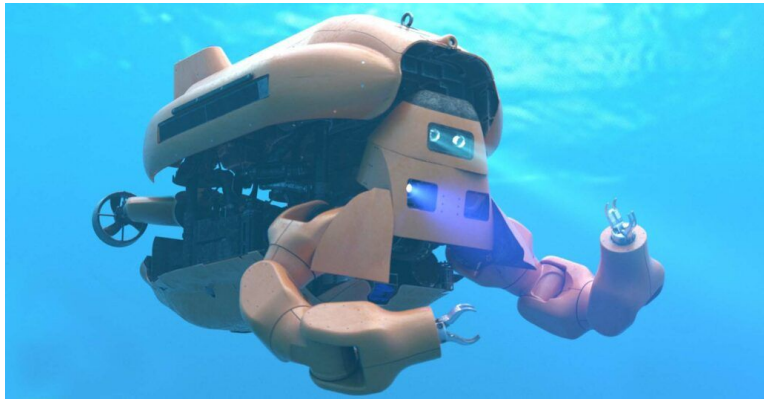
We need a system **Orchestrator** to implement the robot behavior

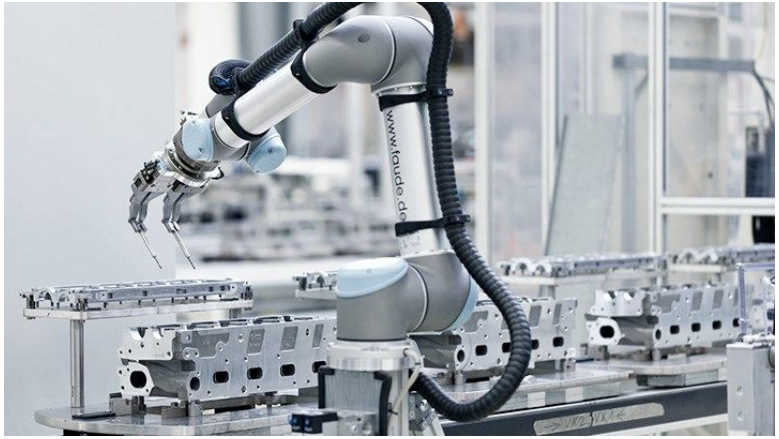
Robot Behaviors

This **Coordinator or Task Planner** is also the highest level of abstraction of our system, where we focus on **WHAT** the robot does (or should do) and not on **HOW** it does it (Skills and Services).

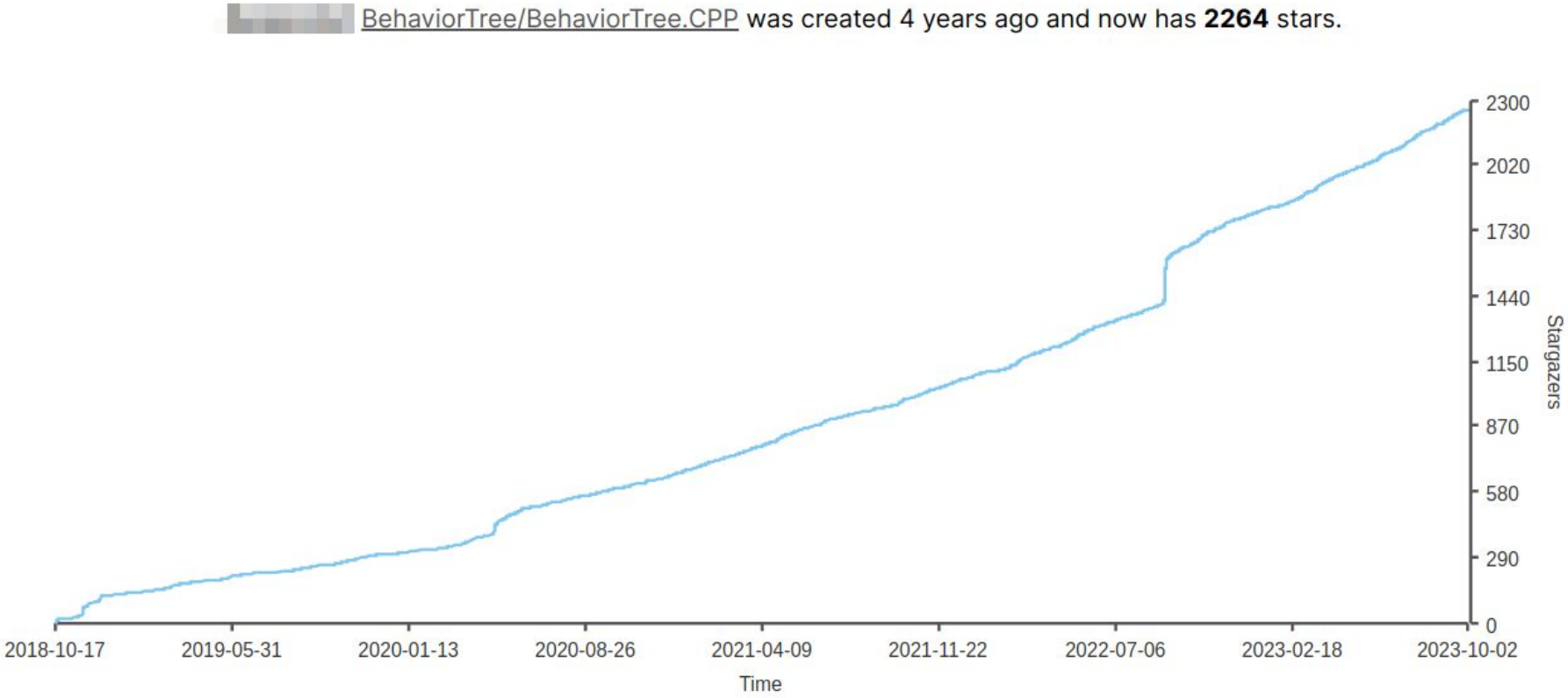


Where BehaviorTree.CPP is used



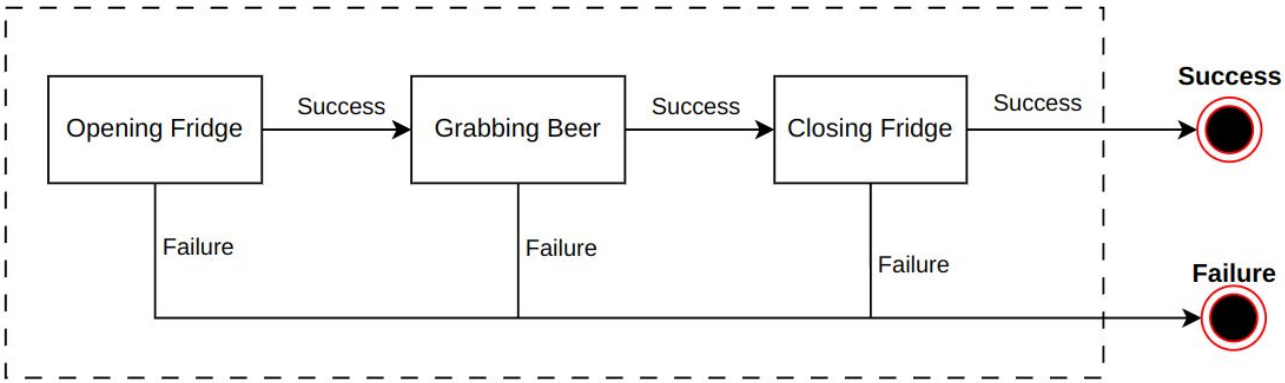


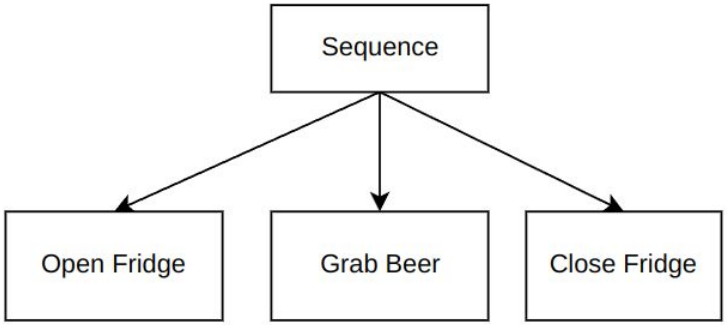
Even if Github Stars are a vanity metric, this gives you an idea of the increasing adoption.



Part 1:
BTs crash-course

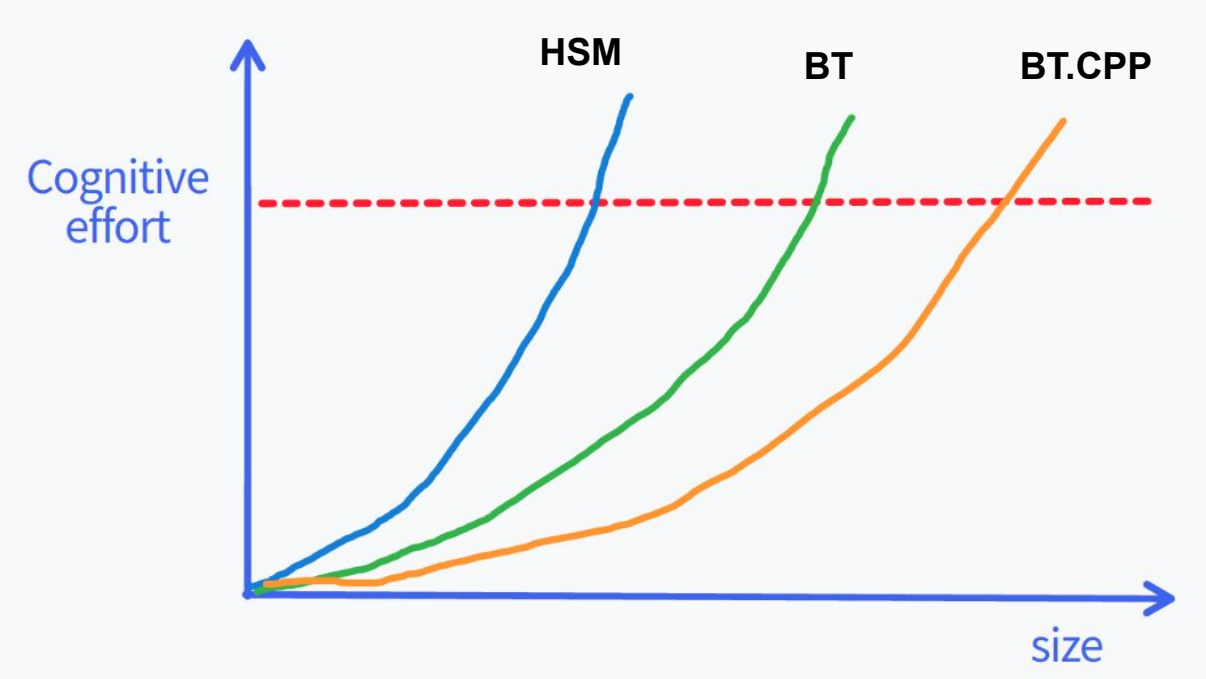
Behavior Trees VS Hierarchical State Machines





The graphical representation of the nodes in a BT has a meaning. Children are ordered from left to right.

Are Behavior Trees better than **Hierarchical State Machines**?

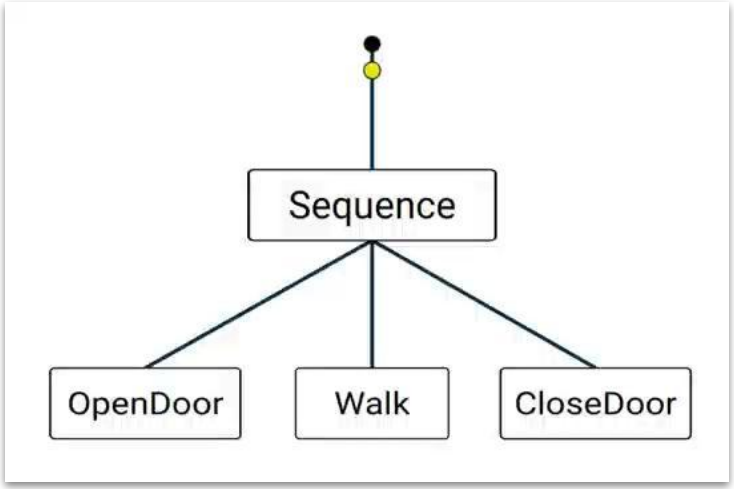


Behavior Trees VS Hierarchical State Machines

	State Machines	**Behavior Trees**
Learning curve	**more familiar**	it needs some practice
Concurrency	no	**yes**
Scalability	poor	**better**
Readability	scales very poorly	**better**

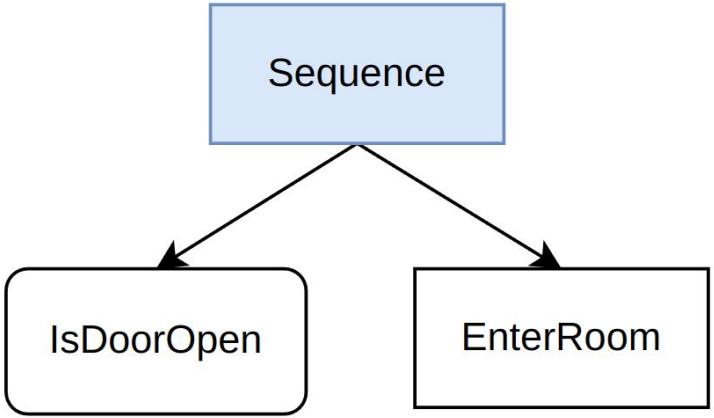
"Tick" and Node types

- The “**tick**” is the signal that we propagate from the root to the leaves.
- The leaves of the tree are the **Actions**. I.e. the user-defined code that should be executed.
- **Conditions** are Actions that are considered atomic and don’t have side effects.
- Nodes can only return SUCCESS, FAILURE or RUNNING.
- **Control Nodes** can have multiple children. **Decorators** can have only one child.
- Control Nodes and Decorators “dispatch” the tick following a certain logic.



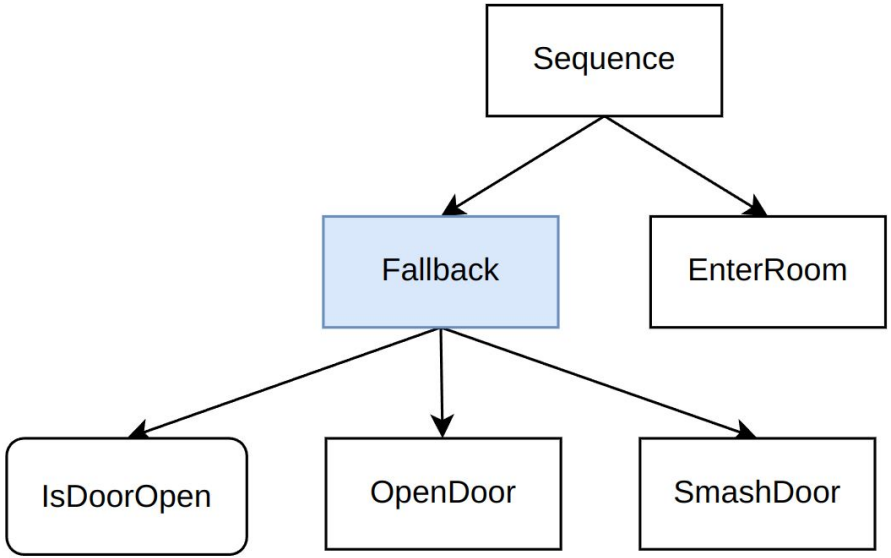
Sequence

- Tick children from left to right, as long as they return SUCCESS
- If all children are successful, Sequence returns SUCCESS
- Stop if a child return FAILURE; Sequence returns FAILURE.



Fallback

- Return SUCCESS as soon as a child returns SUCCESS.
- Otherwise (FAILURE), execute the next child.
- If all children FAIL, Fallback returns FAILURE.

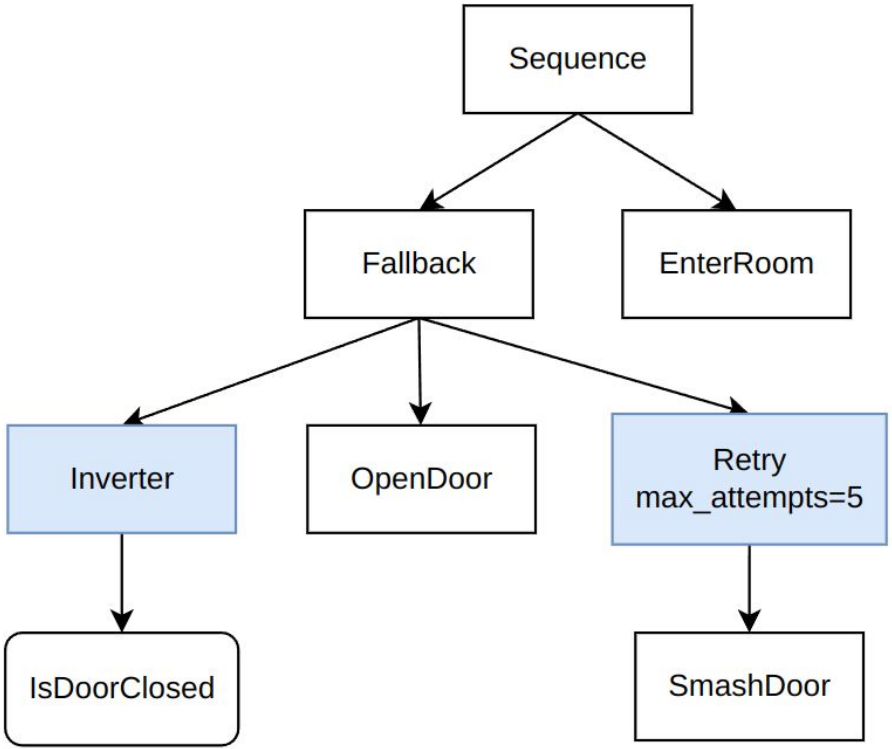


Decorators

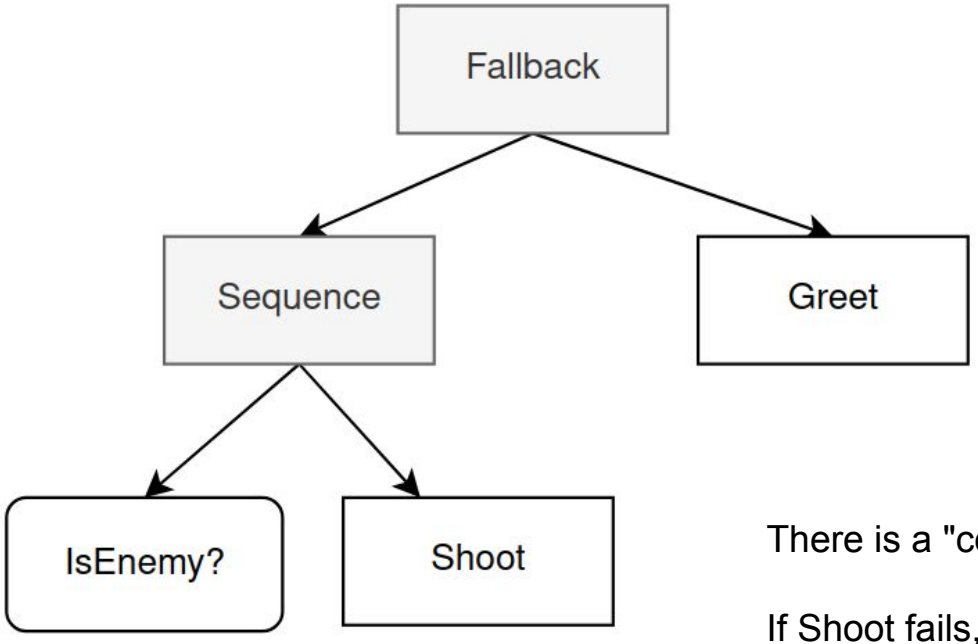
Take a moment to appreciate the power of Decorators, and the complexity of implementing the same in a State Machine!

More examples:

- **ForceFailure / ForceSuccess**
- **Repeat**
- **Timeout**
- **RunOnce**



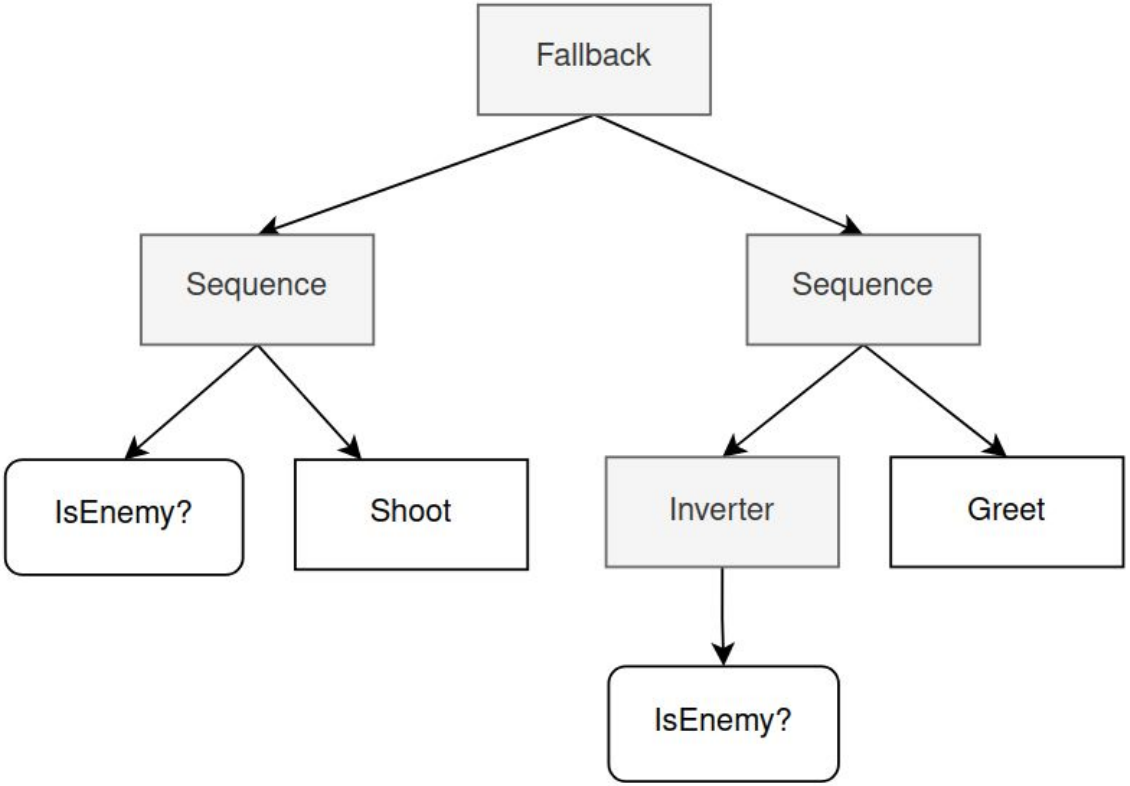
Example: if-then-else



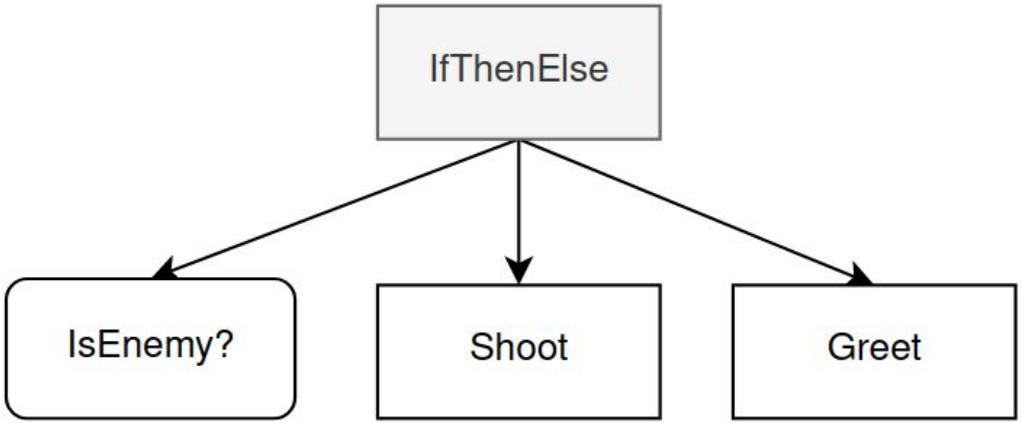
There is a "conceptual" bug here:

If Shoot fails, we will Greet the Enemy

"if-then-else" corrected



if-then-else !



We can extend the number of Control Nodes and Decorators as needed.

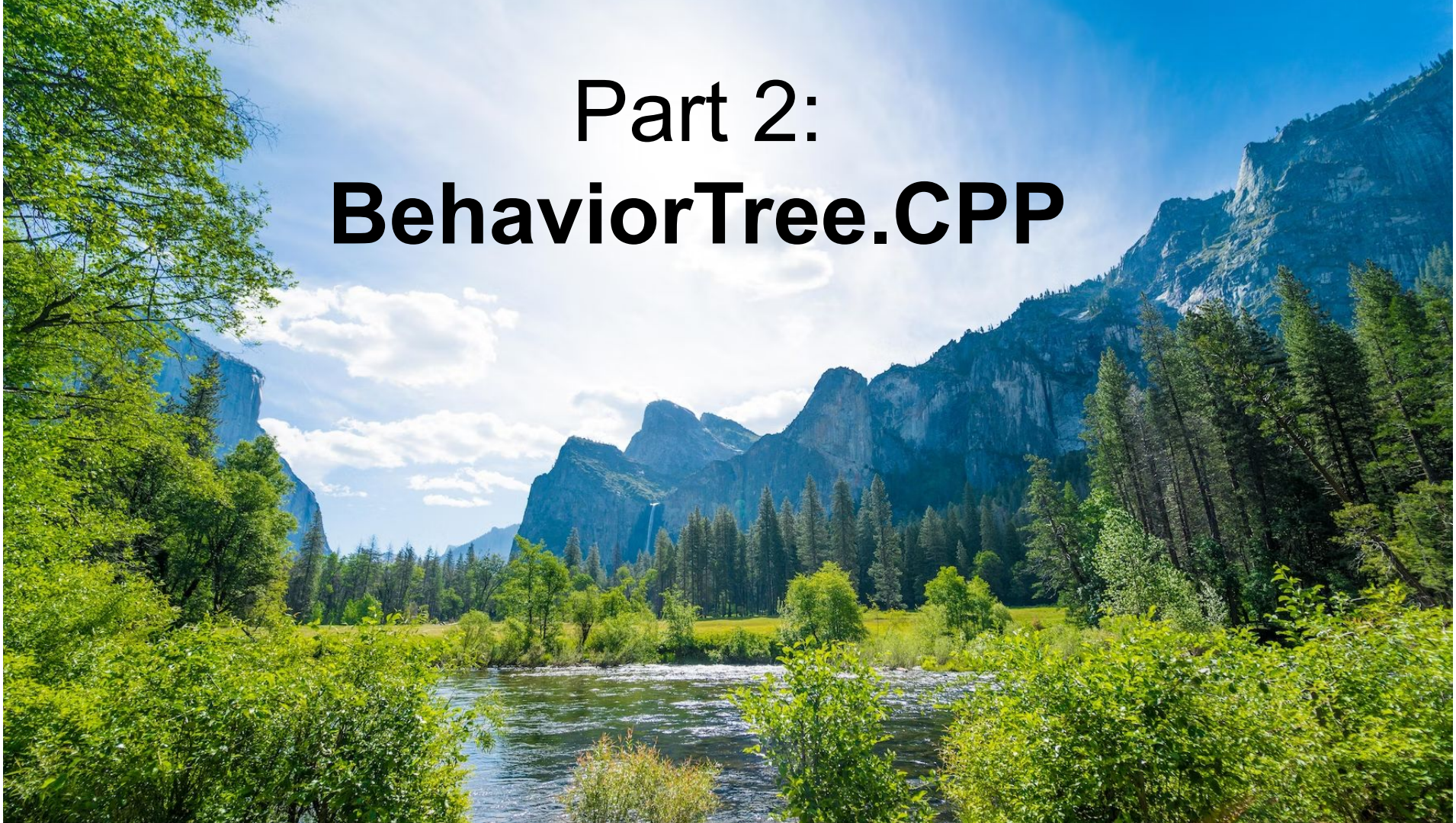
In other words, we need to consider BTs an extensible **Domain Specific Language**

Behavior Trees as a DSL

```
bool FetchBeer()
{
  if( GoTo("kitchen") &&
      OpenFridge() &&
      Grasp("beer") &&
      CloseFridge() )
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

When we start thinking about Behavior Trees as a **domain specific language**, we realize how we need:

- "Functions" and reusable routines, i.e. **Subtrees**
- **Arguments** to be passed to these functions
- **Return values**


Part 2:
BehaviorTree.CPP

I wanted a C++ implementation!

Before starting this project, I found a Python implementation.

- No need to explain what happened next!
- I am not spending my day implementing Python / C++ binders
- But I still wanted the flexibility of a scripting language…

Features of BehaviorTree.CPP

- It mixes the efficiency of C++ with the flexibility of a **scripting language**.
- Considers **asynchronous** actions and **reactive behaviors** as first class citizens.
- **Tons of tools** to debug your behaviors offline (logging) or at run-time (monitoring).
- **Advanced debugging** tools such as fault injection, interactive debugging and more.
- Implements a **novel pre-post condition extension,** that reduces complexity, when dealing with states.

Main Design Principles

1. Productivity
2. Composability
3. Reusability

Nodes and Trees

These are your reusable Tree Nodes, written in **C++**

This is your specific Behavior Tree, written in **XML**

C++ Nodes implementation

```cpp
class ApproachEnemy : public SyncActionNode
{
public:
  ApproachEnemy(const std::string& name, const NodeConfig& config) :
    SyncActionNode(name, config) {}

  // You must override this virtual function
  NodeStatus tick() override
  {
    std::cout << "Type: [ApproachEnemy]. Instance:" << this->name() << std::endl;
    return NodeStatus::SUCCESS;
  }
};

bool ShootAction()
{
  std::cout << "Shoot!" << std::endl;
}
```

C++ Nodes Registration

```
BehaviorTreeFactory factory;

factory.registerNodeType<ApproachEnemy>("ApproachEnemy");

factory.registerSimpleAction("Shoot", [&](TreeNode& wrapping_node)
    {
      return ShootAction() ? NodeStatus::SUCCESS : NodeStatus::FAILURE;
    } );
);
```

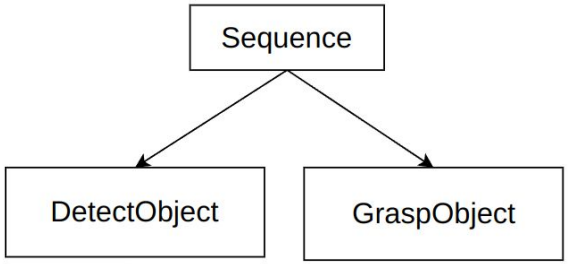
Inheritance is probably the best option, most of the time

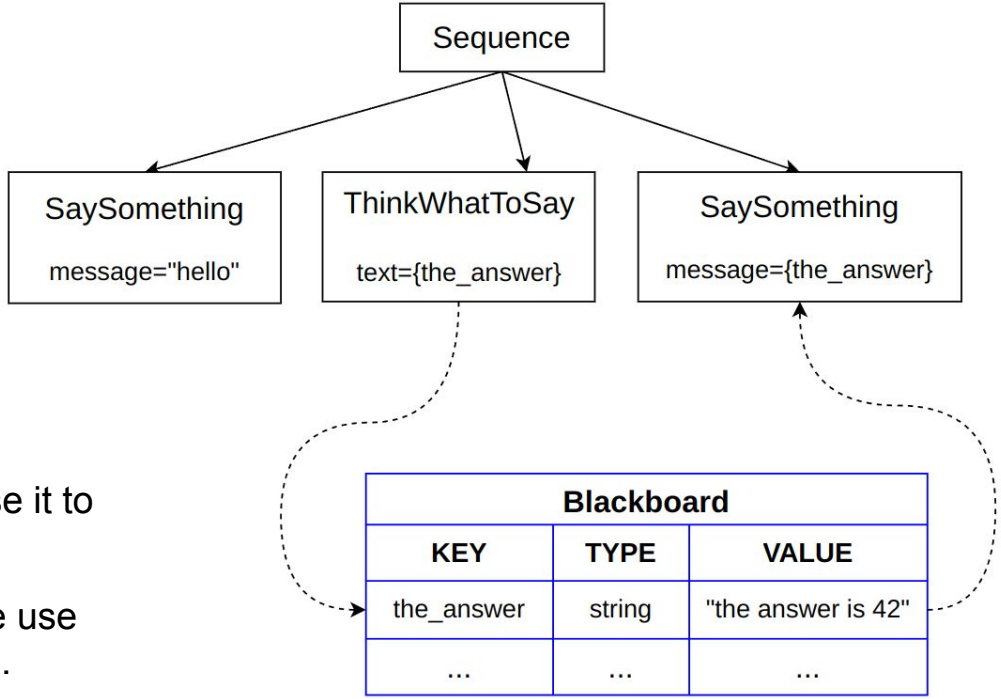
C++ Nodes Registration and XML

```xml
<BehaviorTree ID="ShootEnemy">
  <Sequence>
    <ApproachEnemy/>
    <Repeat num_cycles="3"/>
      <Shoot/>
    </Repeat>
  </Sequence>
</BehaviorTree>
```

```
auto tree = factory.createTreeFromText(xml_text);
NodeStatus status = tree.tickWhileRunning();
```

Dataflow between Nodes





- We need to share data between Nodes.
- We want to make this explicit and expose it to the user.
- The “blackboard” is commonly used. We use a more explicit Input/Output ports model.

```
class Multiply : public SyncActionNode
{
  public:
    Multiply(const std::string& name, const NodeConfig& config):
      SyncActionNode(name, config) {}

    // It is mandatory to define this static method.
    static PortsList providedPorts()
    {
      return { InputPort<int>("value_in"),
               OutputPort<int>("value_out") };
    }

    NodeStatus tick()
    {
      if (auto value = getInput<int>("value_in"))  {
        setOutput<std::string>("value_out") = (*value) * 2;
        return NodeStatus::SUCCESS;
      }
      else {
        std::cout << "missing [message]: ", value.error() << "\n";
        return NodeStatus::FAILURE;
      }
    }
};
```

Using ports

- Ports are an attribute of the class, not the instance. Therefore, they are declared in a **static method.**
- Ports are strongly typed
- When reading a value from an input port, "expected" is returned.

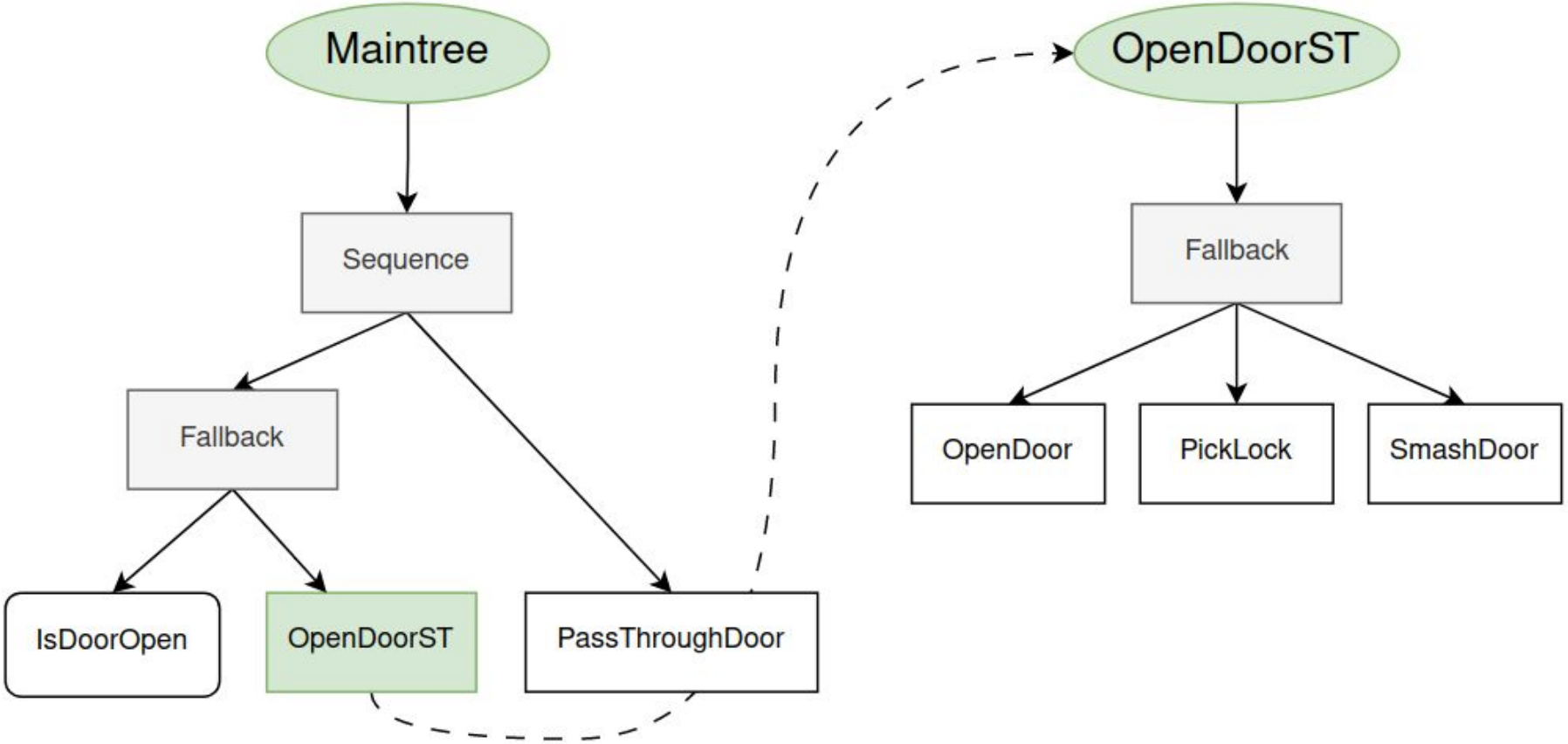
```cpp
template <typename T, typename... ExtraArgs>
void BehaviorTreeFactory::registerNodeType(const std::string& ID, ExtraArgs... args)
{
  static_assert(!std::is_abstract_v<T>, "The given type can't be abstract");

  constexpr bool is_param_constructable =
    std::is_constructible<T, const std::string&, const NodeConfig&, ExtraArgs...>::value;

  constexpr bool has_static_ports_list =
    has_static_method_providedPorts<T>::value;

  static_assert(!(is_param_constructable && !has_static_ports_list),
        "[registerNode]: you MUST implement the static method:\n"
        " PortsList providedPorts();\n");

  static_assert(!(has_static_ports_list && !is_param_constructable),
        "[registerNode]: since you have a static method providedPorts(),\n"
        "you MUST add a constructor with signature:\n"
        "(const std::string&, const NodeParameters&)\n");

  if constexpr (has_static_ports_list)
    registerNodeType<T>(ID, T::providedPorts(), args...);
  else {
    registerNodeType<T>(ID, {}, args...);
  }
}
```

```
class TypeInfo
{
  // omitting other stuff and methods

private:
  // retains information about the type
  std::type_index type_info_;
  // function pointer. Implements "from_string()"
  StringConverter converter_;
};

struct Entry
{
  Any value;  // the type erased value
  TypeInfo info;
  mutable std::mutex entry_mutex;
};

class BlackBoard
{
  // omitting  other stuff and methods

  // This entry might be shared by multiple Blackboards
  std::unordered_map<std::string, std::shared_ptr<Entry>>
     storage_;
};
```

Blackboard

- Using an equivalent to `std::any` for type erasure (16 bytes SVO).
- Only ports with the same type can be connected as "output -> input".
- Exception to type safety: if an input port with a specific type is trying to read from a string, we try converting from string.
- User should specialize `convertFromString<>` for their custom types.

Hierarchical composability and Subtrees

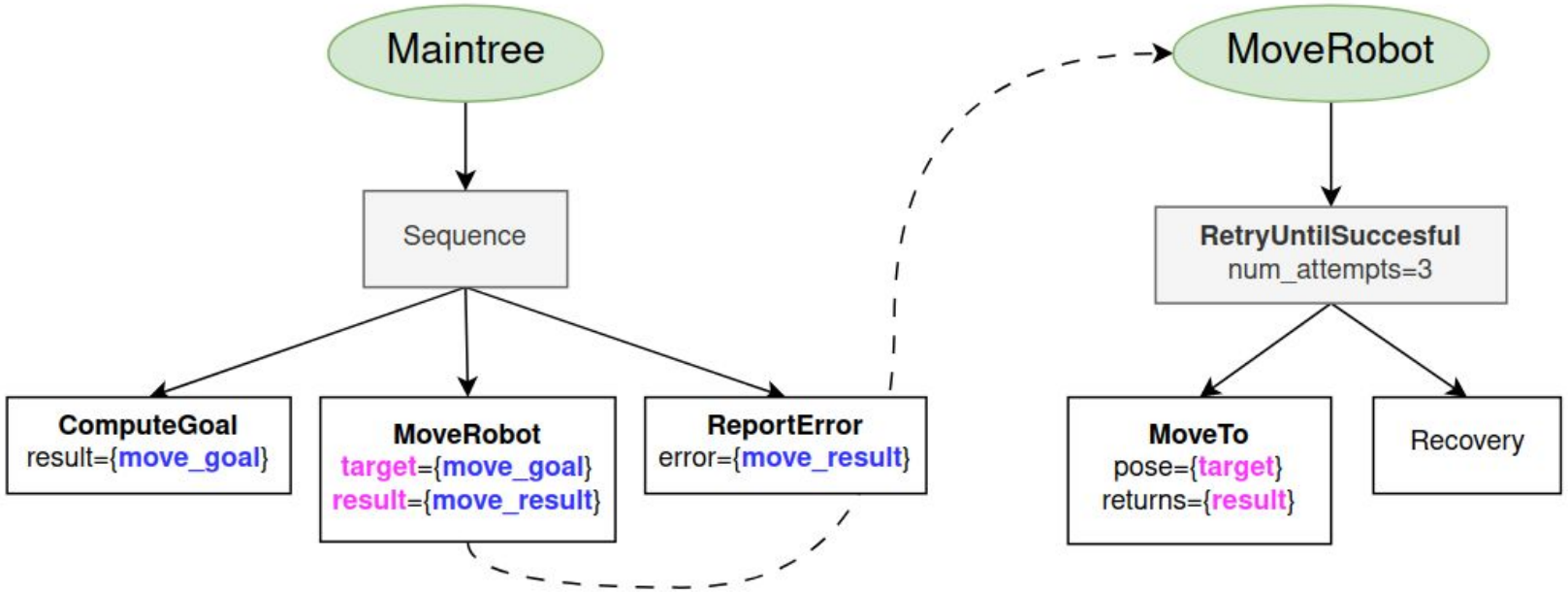

Maintree
Sequence
Fallback
IsDoorOpen
OpenDoorST
PassThroughDoor
OpenDoorST
Fallback
OpenDoor
PickLock
SmashDoor

```
<BehaviorTree ID="MainTree">
    <Sequence>
        <Fallback>
            <IsDoorOpen/>
            <SubTree ID="OpenDoorST"/>
        </Fallback>
        <PassThroughDoor/>
    </Sequence>
</BehaviorTree>

<BehaviorTree ID="OpenDoorST">
    <Fallback>
        <OpenDoor/>
        <PickLock/>
        <SmashDoor/>
    </Fallback>
</BehaviorTree>
```

Subtrees

- We can include a Subtree inside another, as it was a "function".
- Each subtree has its own instance (not a reference).
- Each Subtree has its own Blackboard (scope)
- To make a scoped Blackboard entry visible outside a Subtree, we need to explicitly map it to the parent Tree

Port Remapping




```
<SubTree ID="MoveRobot" target="{move_goal}" result="{move_result}" />
```


BUT WAIT
SANDSTRÖM
ONTV COMPANY
THERE'S MORE
quickmeme.com

Debuggability and Observability

Logging and Remote Monitoring

The library implements an **Observer Pattern**

- Multiple Observers can be "attached"
- The typical applications are **Loggers**:
  - ConsoleLogger, FileLogger, SQLiteLogger.
- You can **publish** the current state of the tree in real-time to an external process (GUI).
- Collect **statistics** about the execution of the tree and the actions in a non-intrusive way.

Code injection and interactive debugging

Code injection and interactive debugging

User can inject at deployment-time or run-time **functions** to be execute before or after the **tick()**. This allow us to:

- Implement "GDB like" breakpoints in the code.
- Substitute entire nodes with "dummy ones".
- Do fault injection in the tree, to test how the robot behave when other branches are taken.
- This functionality is integrated with our remote GUI.

A script language on top of Behavior Trees

A script language on top of Behavior Trees

We added a simple scripting language to manipulate data in the blackboard:

- Arithmetic and Bitwise operators
- Logic and Comparison operators
- Support for ENUMS
- The variables are the Entries in the blackboard

Using
foonathan::Lexy

Jonathan Müller
foonathan

Example

```
<Sequence>
    <Script code=" msg:='hello' " />
    <Script code=" msg+=' world' " />
    <Script code=" A:=42; B:=3.14 + A; color:=RED " />
    <Precondition if="A>B && color!=BLUE" else="FAILURE">
        <Sequence>
            <SaySomething message="{A}"/>     // prints "42"
            <SaySomething message="{B}"/>     // prints "45.14"
            <SaySomething message="{msg}"/>   // prints "hello world"
            <SaySomething message="{color}"/> // printf the number associated with RED
        </Sequence>
    </Precondition>
</Sequence>
```

Pre and Post conditions in Nodes

Preconditions:

- **skipIf**
- **successIf**
- **failureIf**
- **while**

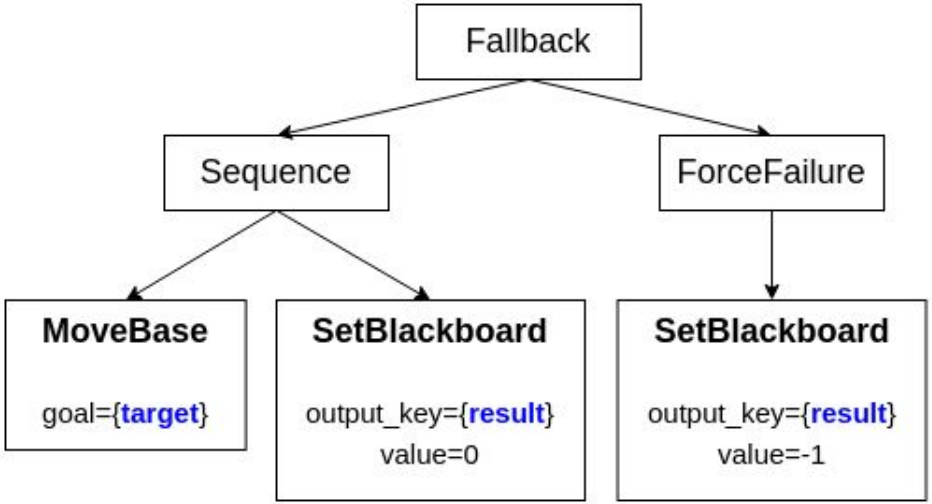
Scripts executed before the **tick()**

Postconditions:

- **onSuccess**
- **onFailure**
- **onHalted**
- **post**

Scripts executed after the **tick()**

Pre and Post conditions in Nodes




```
<MoveBase goal="{target}" _onSuccess="result:=OK" _onFailure="result:=ERROR"/>
```

Reactive Behaviors and Concurrency

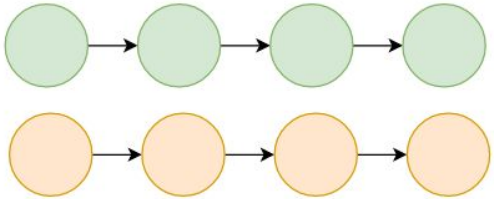
Asynchronous Actions

Actions may require a long time to be executed:

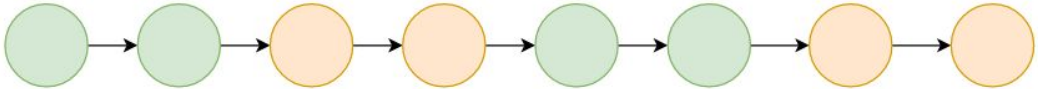
- Nodes may return **RUNNING** to communicate that they haven't finished, and that you should ask later.
- We want to be able to stop a long-running action, if something happens (reactive behaviors)
- We want to execute multiple actions at the same time.

Concurrency VS Parallelism

Parallelism



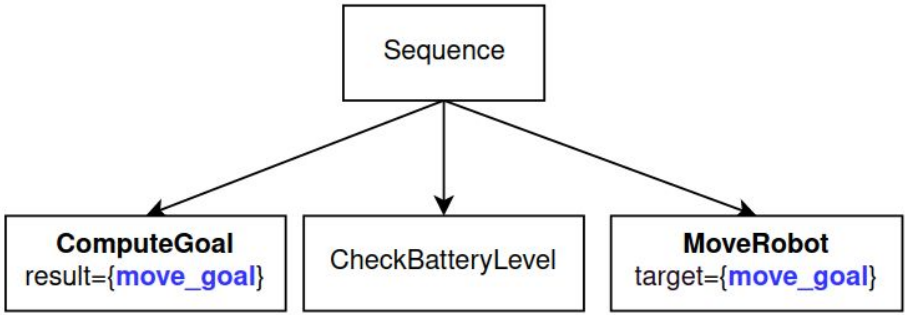
Concurrency

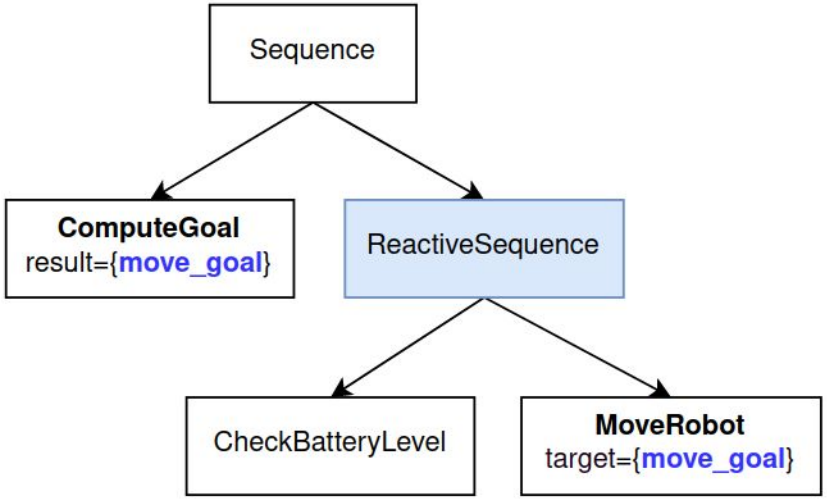


BT.CPP uses a Concurrency model to execute Asynchronous Actions

Examples: Reactive Nodes

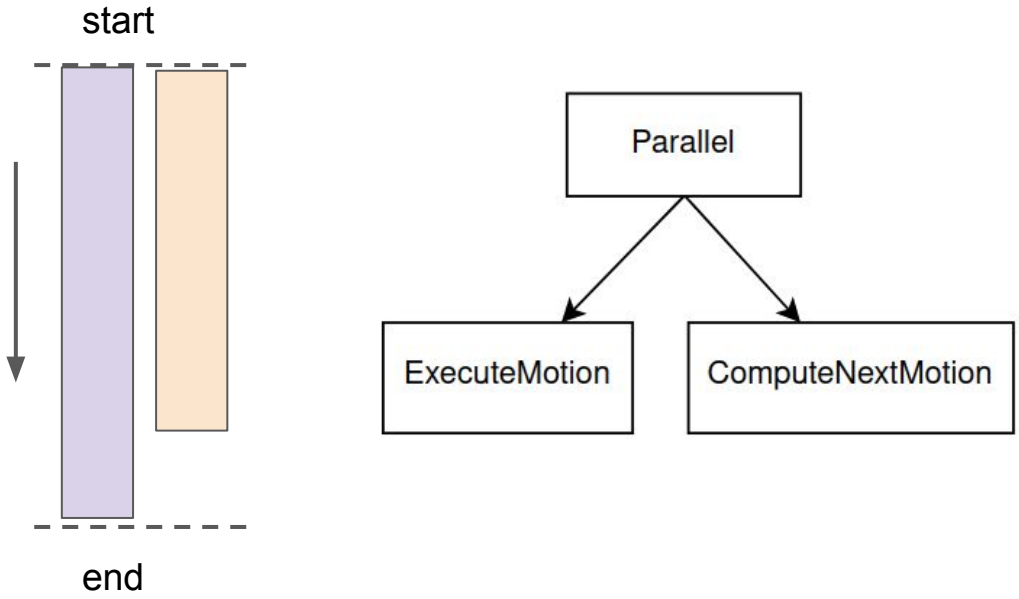
The tree on the right will call **CheckBatteryLevel** at each tick, and interrupt **MoveRobot** if FAILS





Examples: Parallel Nodes

While executing an arm motion, compute the next motion to be executed



About Multi-threading

I can't do multi-threading for you!

It is YOUR responsibility, when implementing an Async Node, to do it "right":

- There is a (now deprecated) implementation with its own thread. But people expect that to work auto-magically.
- There is no magic way I can stop your thread, it is better that you comply to my interfaces (there is a **halt()** callback).
- You will need some cleanup, when your Action is interrupted.
- You must not keep the thread calling **tick()** hostage (avoid starvation).

```
class LongAction : public ActionNodeBase
{
public:
  // omitting the constructor for brevity

  NodeStatus tick() override
  {
    // simulating a very long action
    LongOperation();
    return NodeStatus::SUCCESS;
  }

  void halt() override
  {
    // nothing I can do here :(
  }
};
```

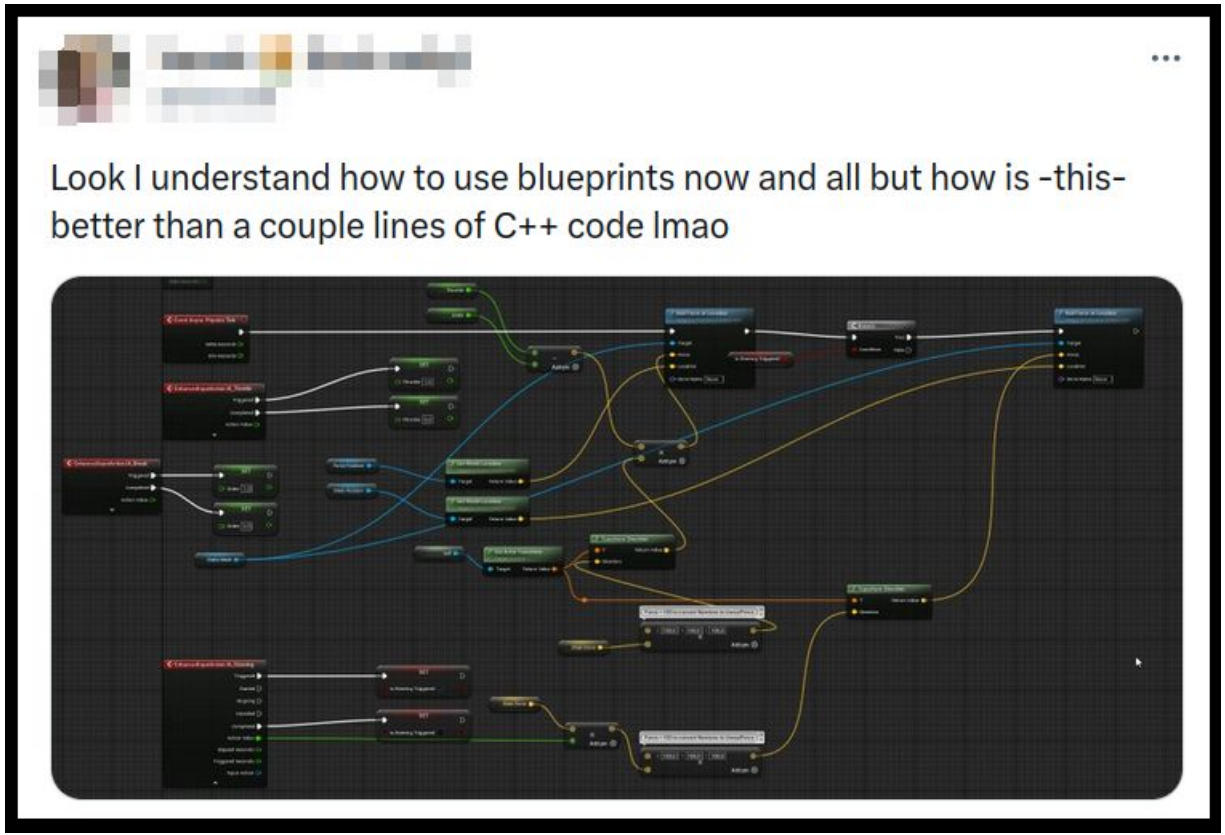
Async Actions

Here you can notice how the **tick()** will block the main thread executing the tree.

This prevent us from implementing a reactive behavior.

```cpp
class LongAction : public StatefulActionNode
{
public:
  LongAction(const std::string& name, const NodeConfig& config) :
    StatefulActionNode(name, config) {}

  NodeStatus onStart() override {
    fut_ = std::async(std::launch::async, []() { LongOperation(); });
    return NodeStatus::RUNNING;
  }

  NodeStatus onRunning() override {
    const std::chrono::milliseconds no_wait(0);
    if(fut_.wait_for(no_wait) == std::future_status::timeout) {
      return NodeStatus::RUNNING;
    }
    return NodeStatus::SUCCESS;
  }

  void halt() override { /* nothing I can do here :( */ }

private:
  std::future<void> fut_;
};
```

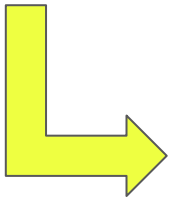
Final Remarks



Final Remarks

The **true value** of Behavior Trees (and State Machines, to be fair) is that:

1. They allow us to reason at a higher level of abstraction.
2. It allows people that are not domain experts or C++ developers to design or understand the robot behavior
3. They oblige developers to encapsulate their code.



A "side effect" of BTs is actually its real advantage:

better **separation of concerns** and **modularity**

https://github.com/BehaviorTree/BehaviorTree.CPP

Thank you!