UNIVERSITÀ DEGLI STUDI
DI MILANO

Behaviour Trees

A.I. for Video Games

Davie Gadia (gadia@di.unimi.it) and Dario Maggiorini (dario@di.unimi.it)

Playlab fOr inNovation in Games

What is a Behavior Tree?

- A convenient way to create a character AI

- A convergence of several techniques:
  - Hierarchical State Machines
  - Scheduling
  - Planning
  - Action Execution

- They are originally a software engineering solution
  - Friendly to non-programmers
  - Easy to read and understand

What is a Behaviour Tree For?

- We already decided what to do
  - So, it is NOT for decision making
- We already planned ahead about actions
  - So, it is NOT for planning
- We have to perform an action following a believable behavior
  - How many ways do you have to “open the door”?
  - How many ways do you have to “cross the street”?

Edges are Blurry ... Once Again

- Behaviour Trees are NOT a decision-making strategy
  - ... and the book confirms that!

- Nevertheless, while you are walking a tree, you end up taking small decisions and planning in order to take out the action
  - If the door is locked, then bash it
  - If there is a guard, then take a detour
  - To fire; load, lock, and pull trigger

- The distinction is sometime a matter of personal taste
  - A personal interpretation: since everything is hardcoded in the tree structure, then we are not actually planning or deciding much
  - You are free to have your idea, just be ready to argument it during the exam 😈

Building a Behavior Tree

- The basic building block is a “task”
  - A definition about a simple “*something to do*”
  - (mostly) self-contained
  - With a standard interface
- Tasks can be combined (in a small tree) to describe complex actions
- Actions can be combined (in a larger tree) to describe a behavior
- To create a tree, use a top-down approach
  - You keep decomposing your NPC behavior until you have tasks that are easy enough for your NPC to do individually

Anatomy of a Task

- A task will “do something” and report success/failure
  - Tempted to implement using a Boolean function?
    - How about returning error codes
    - How about integrating with the scheduling system?

- There are three types of tasks:
  1. Condition
     - To test a property
       (from the internal or external knowledge)
  2. Action
     - To alter the state of the game
       (change something in the internal or external knowledge)
  3. Composite
     - The behavior is based on the behavior of its children
       (organized as a small tree)

Composites

- We can create complex behaviours using a limited numbers of composites

- Selector
  -A set of things to do as alternatives to perform an action; such as open, or push, or bash, or destroy a door to enter a room
  - Will run its children in order until a successful one is found
  - Return value will be “fail” only if all children fail

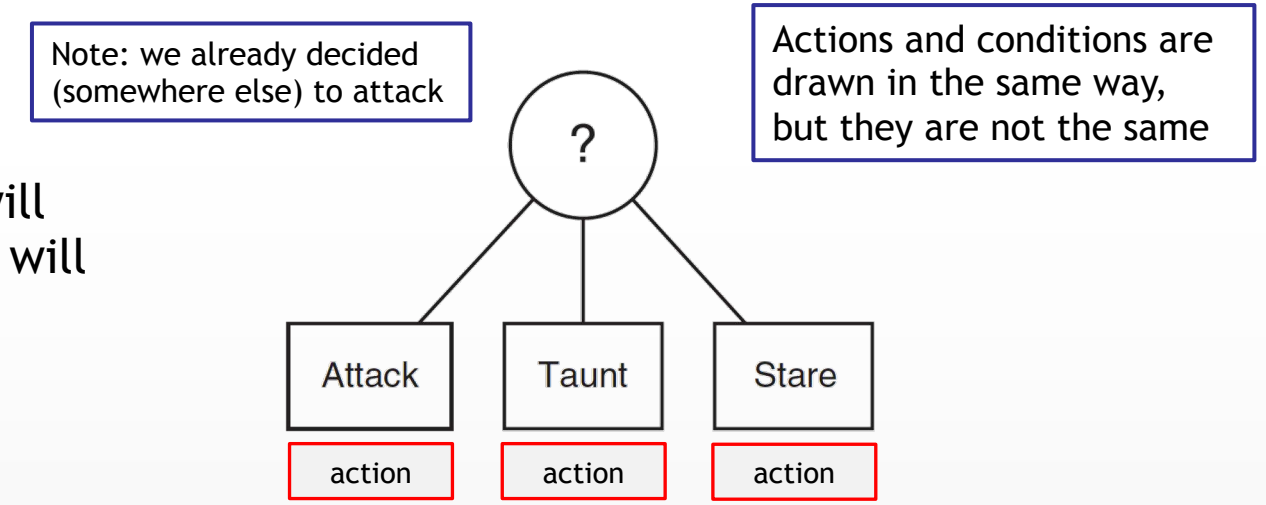
- Sequence
  -A sequence of things to do to perform an action; such as load, lock and pull trigger to fire with a gun
  - Will run all its children until a failing one is found
  - Return value will be “success” only if all children succeed

Composites

A selector for a fighting behaviour.

The NPC will attack, if that is not possible, will taunt. If taunt will not be possible as well, it will just stare to the enemy.

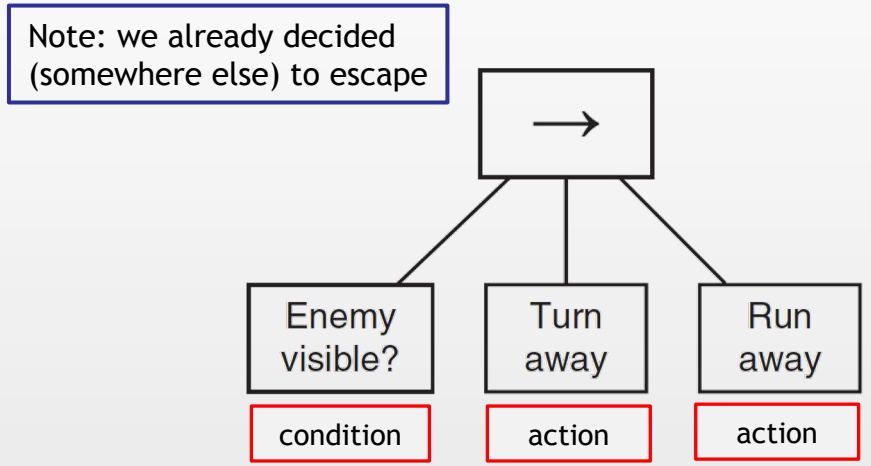
If all the actions fail, the fight will fail



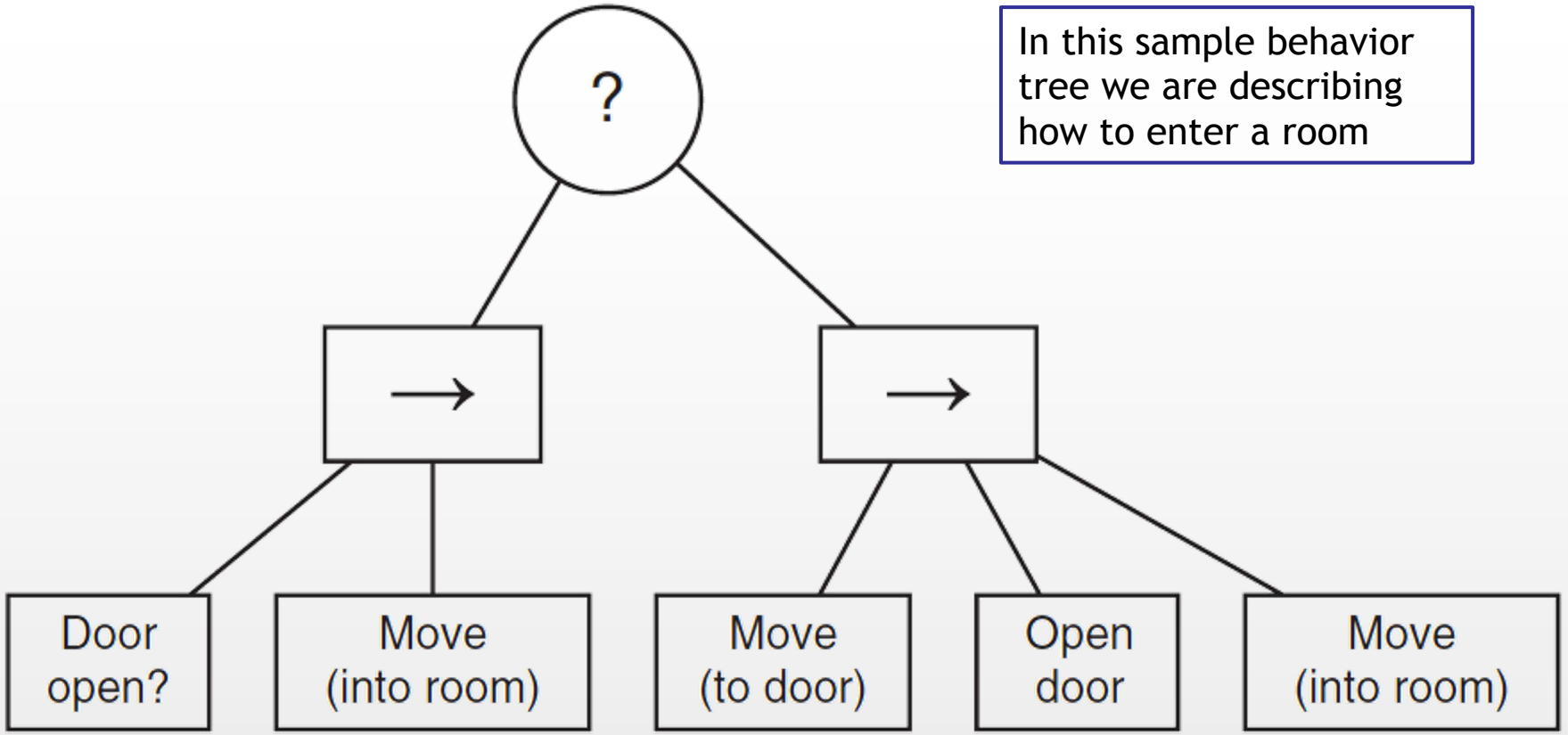
A sequence for an escape behaviour.

The NPC will check if the enemy is visible.
If so, it will turn around and then run away.

If one of the actions fails, the escape will fail

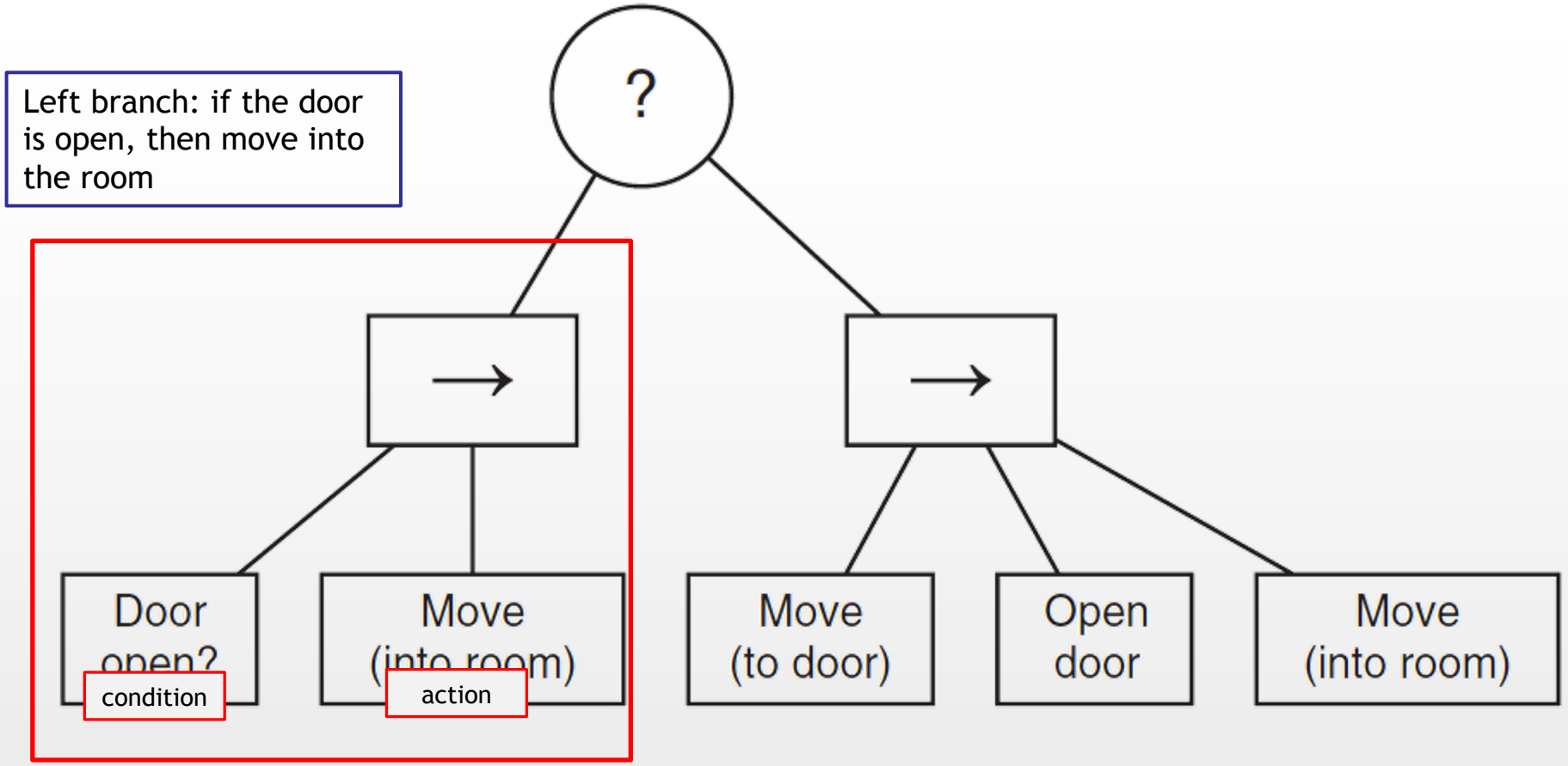


Sample BT



In this sample behavior tree we are describing how to enter a room

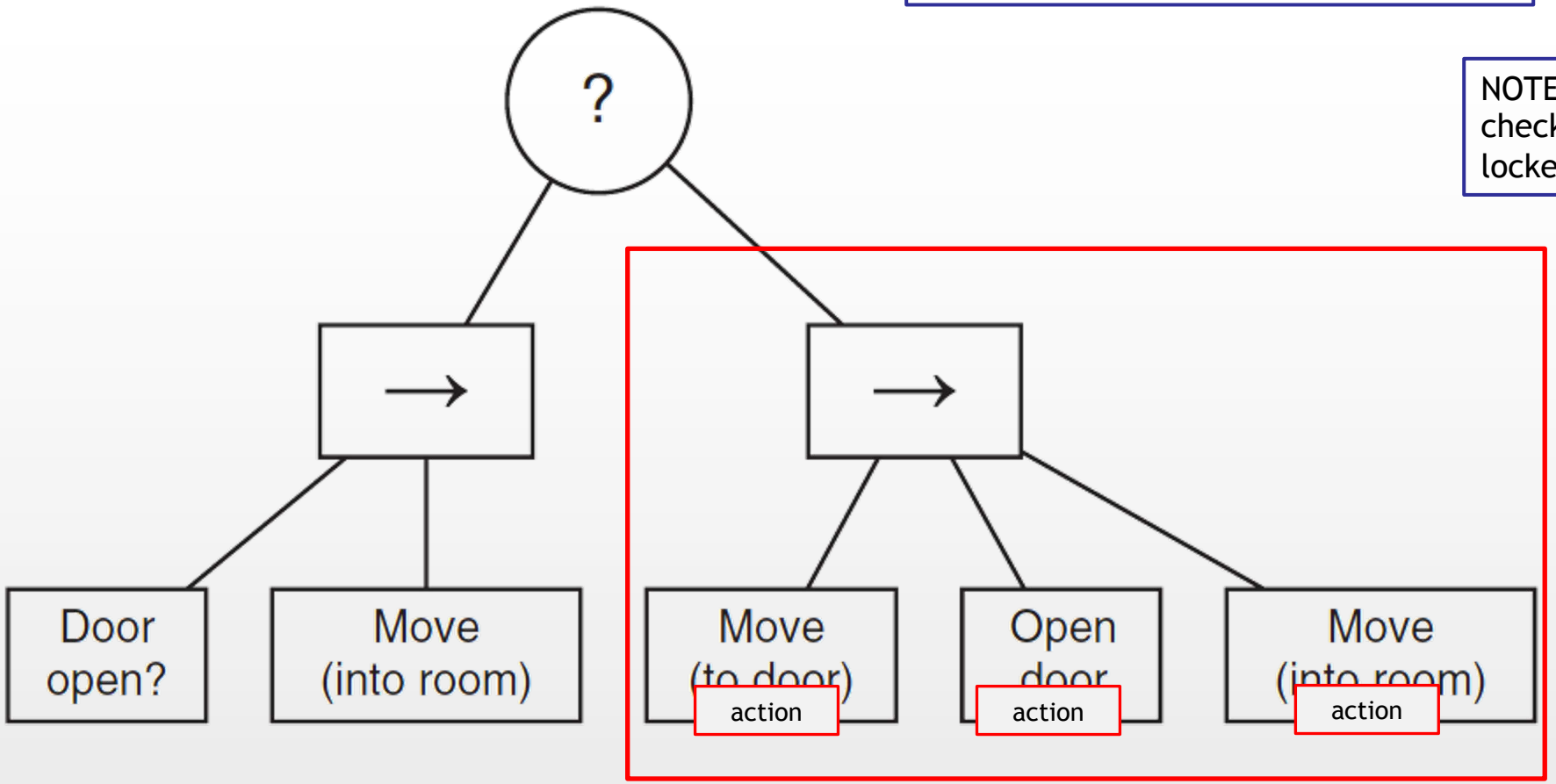
Sample BT


Left branch: if the door is open, then move into the room
?
Door open?
Move (into room)
Move (to door)
Open door
Move (into room)
condition
action

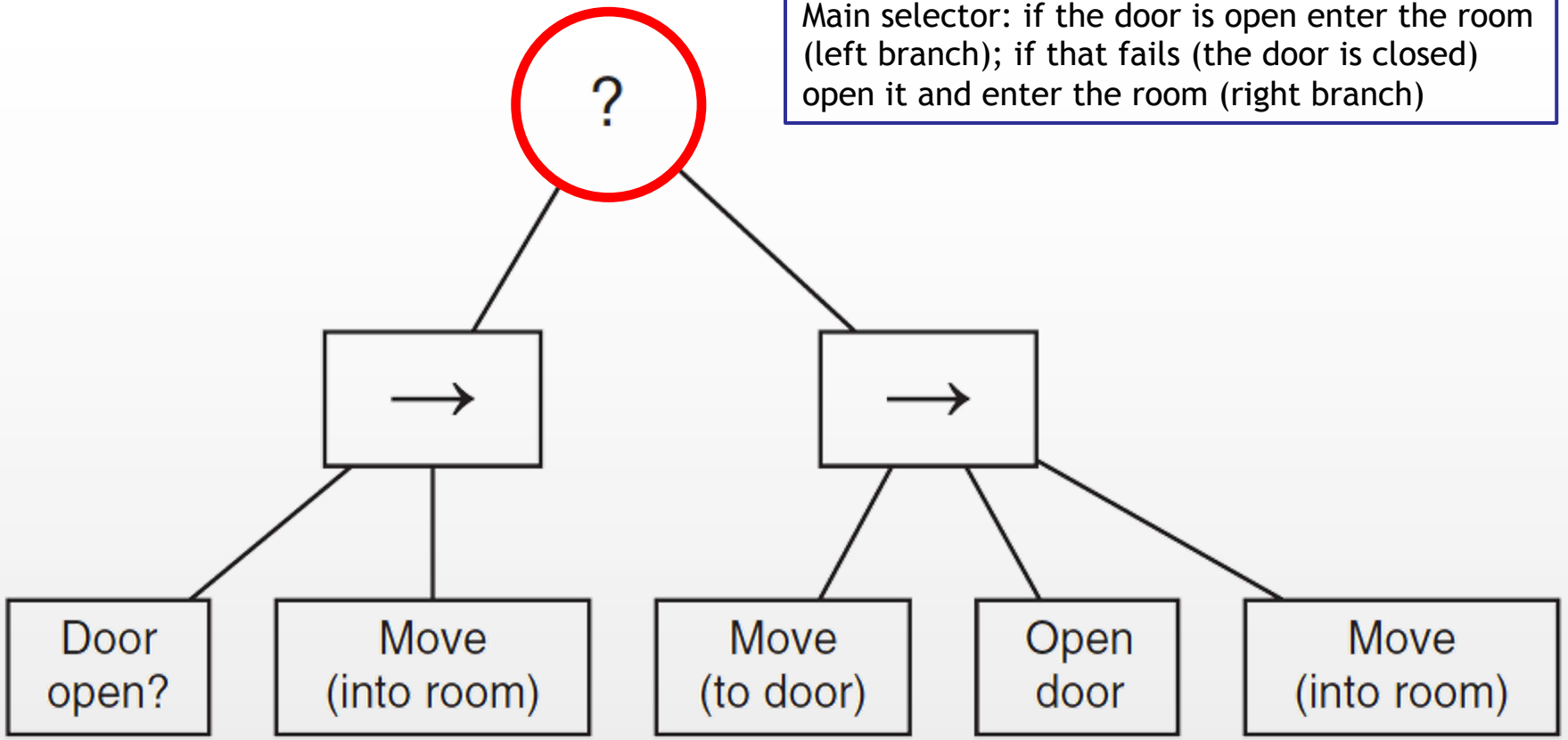
Sample BT

Right branch: move closer to the door, open the door, and move into the room

NOTE: we are NOT checking if the door is locked or not



Sample BT

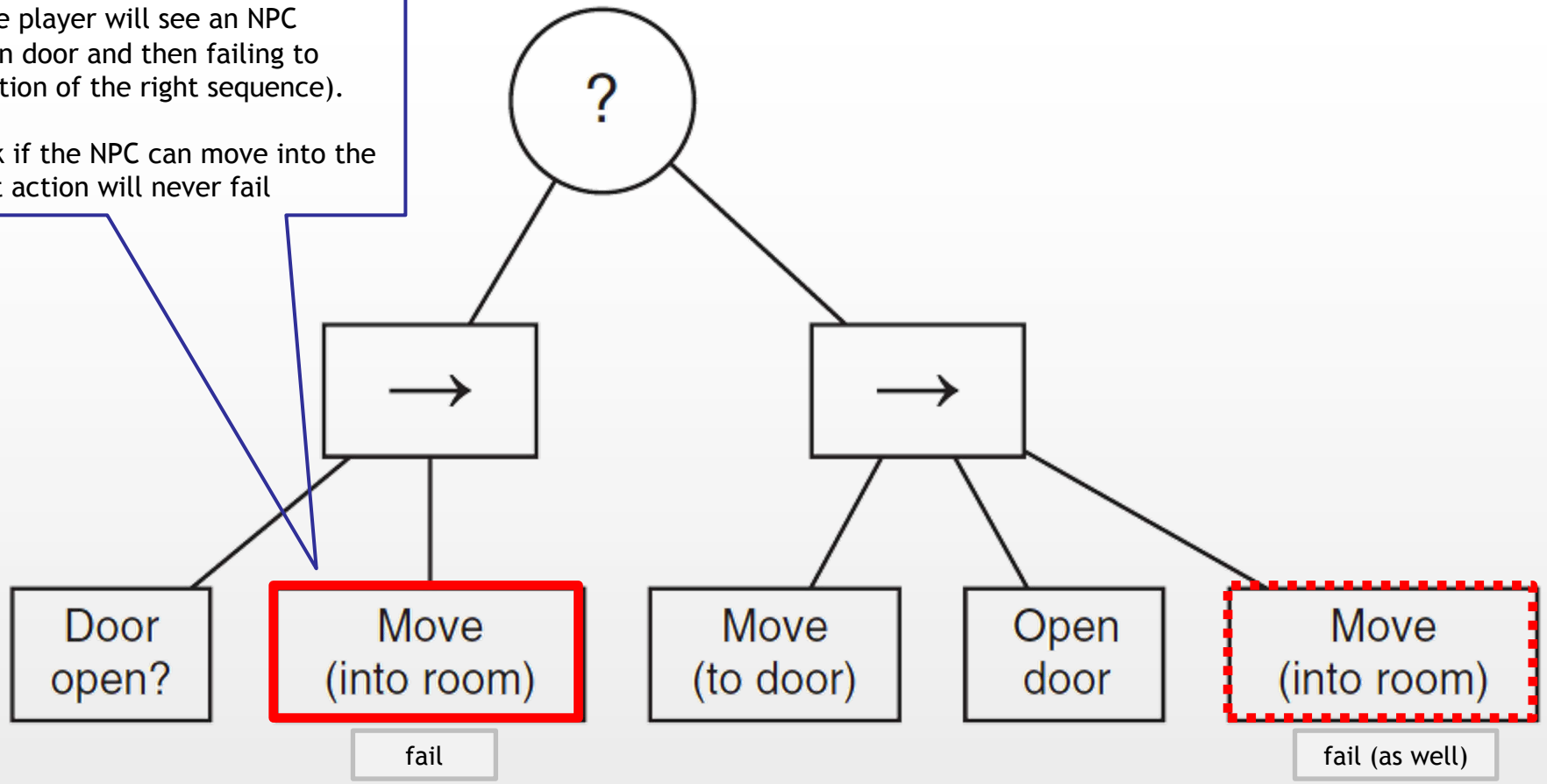


Main selector: if the door is open enter the room (left branch); if that fails (the door is closed) open it and enter the room (right branch)

Sample BT (Nobody is Perfect)

What if the door is open but this action fails?
The left sequence will fail and the selector will run the right one. Then, the player will see an NPC opening an already open door and then failing to enter the room (last action of the right sequence).

The solution is to check if the NPC can move into the room or make sure that action will never fail

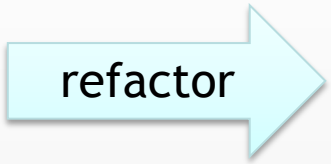


Pseudo-code Equivalent

```
if is_locked(door):
    move_to(door)
    open(door)
    move_to(room)
else:
    move_to(room)
```

Pseudo-code Equivalent

```
if is_locked(door):
    move_to(door)
    open(door)
    move_to(room)
else:
    move_to(room)
```




```
if is_locked(door):
    move_to(door)
    open(door)
move_to(room)
```

Refactoring is possible, just apply some Boolean algebra rules.

Then, we discover that our tree was sub-optimal and can be reduced

Nice Try, Millington!

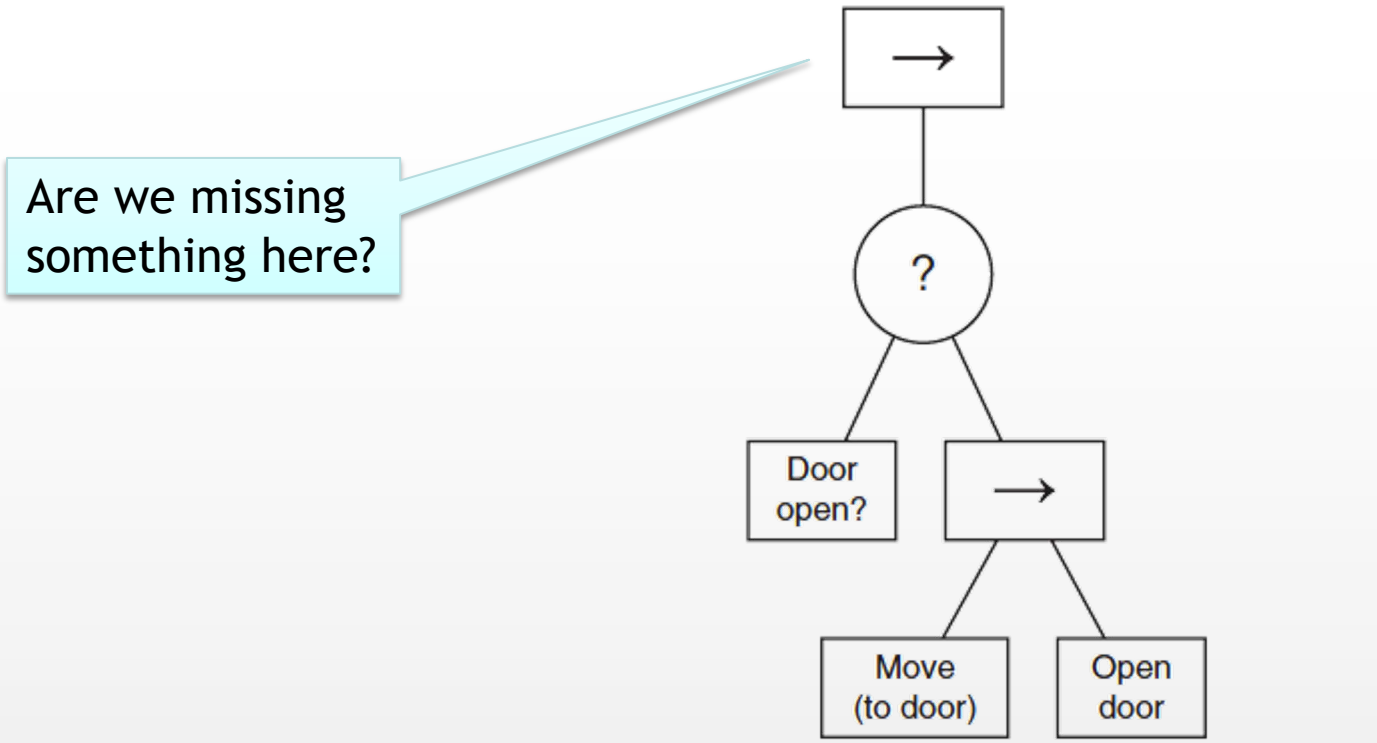
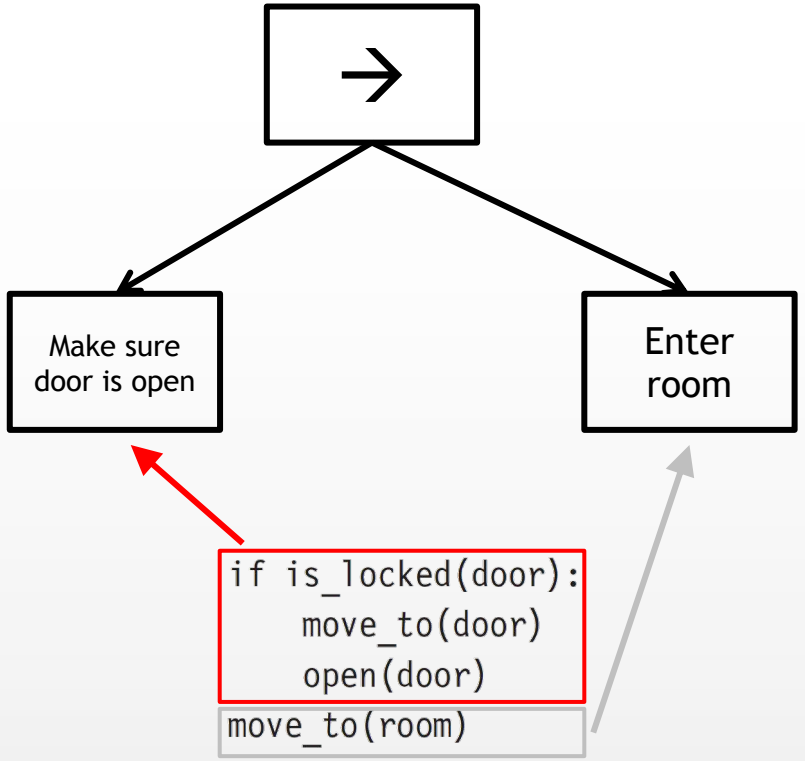


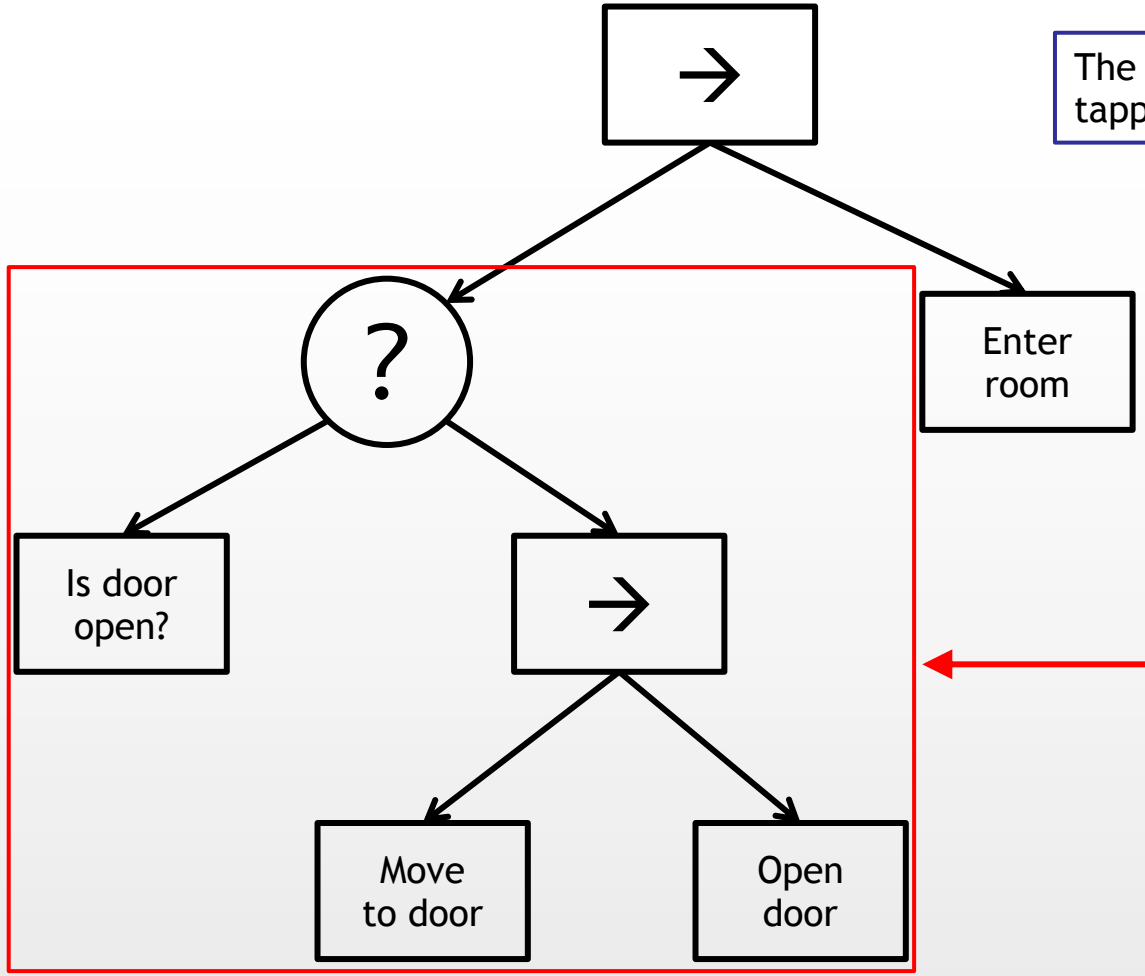
Figure 5.26 A more complicated refactored tree

Let’s Try by Ourselves



In the code, we have an if to make sure the door is open, and then we enter the room (independently by the original status of the door)
NOTE: there is no “else” branch

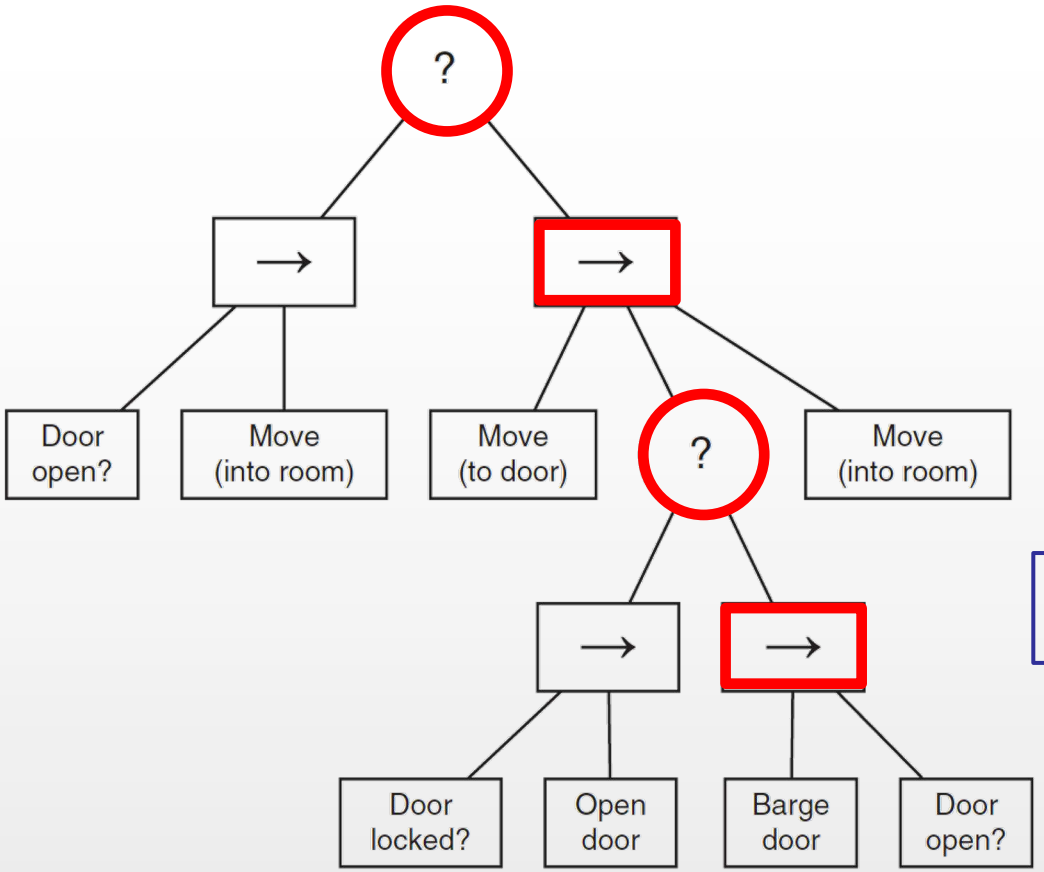
Let’s Try by Ourselves



The left branch can be filled up tapping from the original tree

```
if is_locked(door):
    move_to(door)
    open(door)
```

A More Complex Example



- If the door is open, we enter the room
- Otherwise, we move close to the door and
  - If it is locked, then we just open it
  - Otherwise, (e.g., it is stuck) we barge into it and check if it is now open
- Then, we can enter the room

If, after barging, the door is not open, this sequence will fail

And the sequence failing will make the selector failing

And the selector failing will make another sequence failing

And that will cause the whole tree to fail.

And the AI will know the NPC was not able to carry on the action

Implementing Behaviour Trees

Source: MonoBehaviourTree
Folder: Behaviour Trees

```
// Defer function to actions and conditions
public delegate bool BTCall();

// Uniform interface for all tasks
public interface IBTTask {
    bool Run();
}

// A task with a condition to be verified
public class BTCondition : IBTTask {

    protected BTCall Condition;

    public BTCondition(BTCall call) { Condition = call; }

    public bool Run() { return Condition(); }
}

// A task with an action to perform
// NOTE: the code here is the same as BTCondition
//       we just assume a different semantic in the delegate
public class BTAction : IBTTask {

    protected BTCall Action;

    public BTAction(BTCall call) { Action = call; }

    public bool Run() { return Action(); }
}
```

Implementation follows a similar strategy to decision trees

We define a common interface to run (evaluate or execute) a node, and from there two classes, one for action and one for condition, encapsulating a delegate

Implementing Behavior Trees

```
// Abstract class holding tasks and compose subtasks
public abstract class BTComposite : IBTTask {

    // The list of attached children
    protected IBTTask[] children;

    public BTComposite(IBTTask[] tasks) {
        children = tasks;
    }

    // as for IBTTask interface
    public abstract bool Run();
}
```

Each composite is implementing the same interface (because it is part of the tree as well)

Composite is defined as an abstract class (must be extended by other composites) and adds a list of children to the interface

Implementing Behavior Trees

```
// A task implementing a selector
public class BTSelector : BTComposite {

    public BTSelector(IBTTask[] tasks) : base(tasks) { ; }

    override public bool Run() {
        // All children are run until one succeeds
        foreach (IBTTask t in children) {
            if (t.Run()) return true;
        }
        // Otherwise the selector fails
        return false;
    }
}

// A task implementing a sequence
public class BTSequence : BTComposite {

    public BTSequence(IBTTask[] tasks) : base(tasks) { ; }

    override public bool Run() {
        // All children are run until one fails
        foreach (IBTTask t in children) {
            if (!t.Run()) return false;
        }
        // otherwise the sequence succeeds
        return true;
    }
}
```

All composites need to extend the base abstract class and just fill in the Run method required by the interface

Selector iterates on children and returns true if one is successful

Sequence iterates on children and returns false if one fails

Implementing Behavior Trees

```
// This class will hold the decision structure
public class BehaviorTree : IBTTask {

    protected IBTTask root;

    public BehaviorTree(IBTTask task) { root = task; }

    public bool Run() { return root.Run(); }
}
```

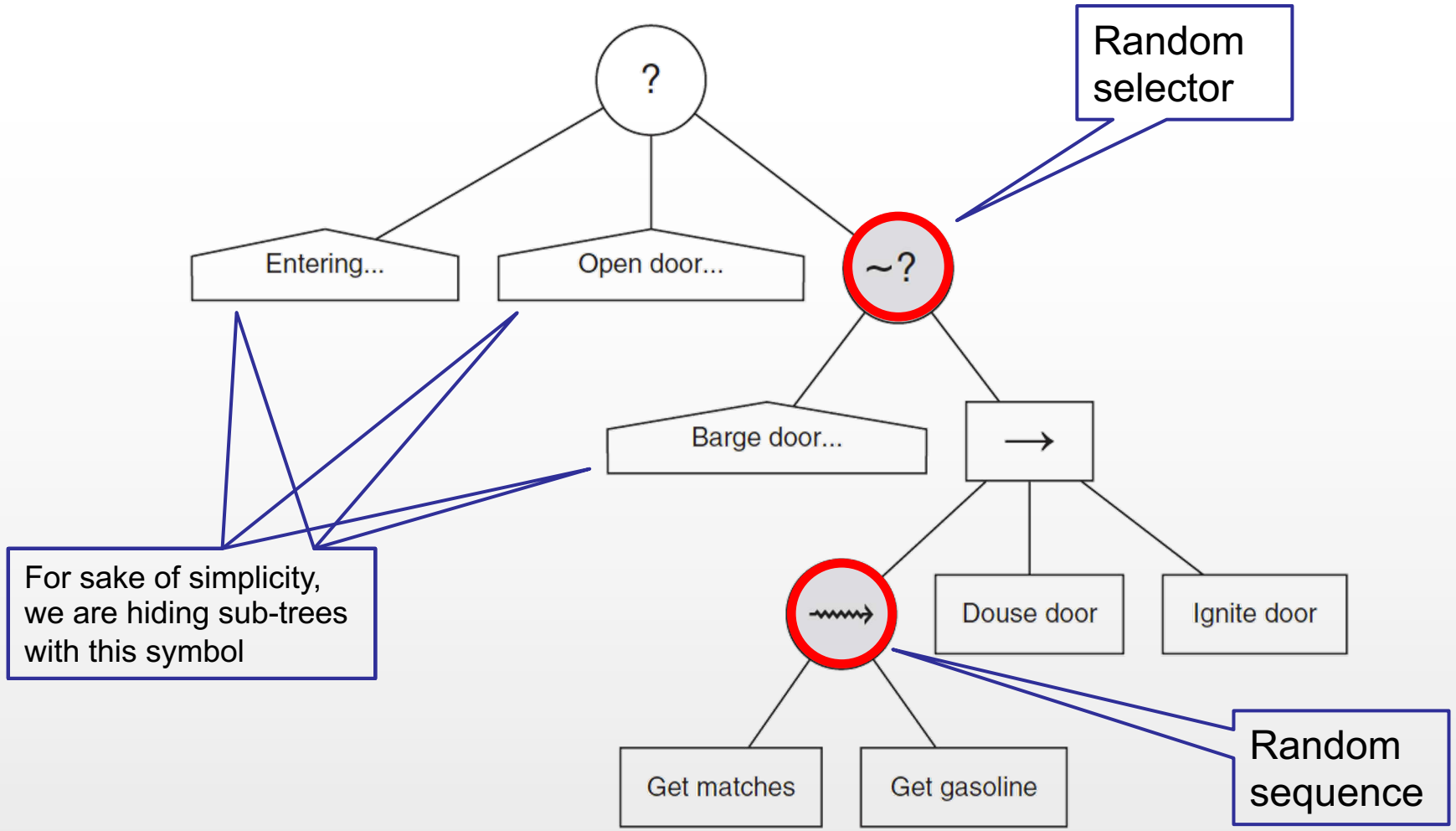
The Behaviour Tree is a data structure with a reference to the root node.

Evaluating the Behaviour Tree is the same as evaluating its root node

Random Composite Tasks

- Performing actions and evaluating conditions always in the same order leads to a predictable behavior
- To spice up player experience we may want to randomize execution order
  - Granted that order is not important for correctness
- Randomizing only a subset may also be convenient and will generate the required effect
  - Always remember, complexity is in the mind of the player; you do not need to be complex in order to have the player perceive complexity (or chaos)

Randomizing Example



Implementing Randomness

```
// Abstract class holding tasks to be run in random order
public abstract class BTRandomComposite : BTComposite /* , IBTTask */ {

    public BTRandomComposite(IBTTask[] tasks) : base(tasks) { ; }

    // Randomize the tasks vector
    public void Shuffle() {
        Random rnd = new Random ();
        children = children.OrderBy (x => rnd.Next ()).ToArray ();
    }
}
```

This shuffle function is exploiting the linq library

```
include System.Linq;
```

We extend the abstract composite class with another abstract class for random composites.

This new class is adding a shuffle function

Implementing Randomness

```
// A task implementing a random selector
public class BTRandomSelector : BTRandomComposite {

    public BTRandomSelector(IBTTask[] tasks) : base(tasks) { ; }

    override public bool Run() {
        Shuffle();
        foreach (IBTTask t in children) {
            if (t.Run()) return true;
        }
        return false;
    }
}

// A task implementing a random sequence
public class BTRandomSequence : BTRandomComposite {

    public BTRandomSequence(IBTTask[] tasks) : base(tasks) { ; }

    override public bool Run() {
        Shuffle();
        foreach (IBTTask t in children) {
            if (!t.Run()) return false;
        }
        return true;
    }
}
```

Same code as before but for a call to the shuffle function

There are many other ways to achieve this, such as overloading or extending the constructor. All are valid, this way is just more readable in class

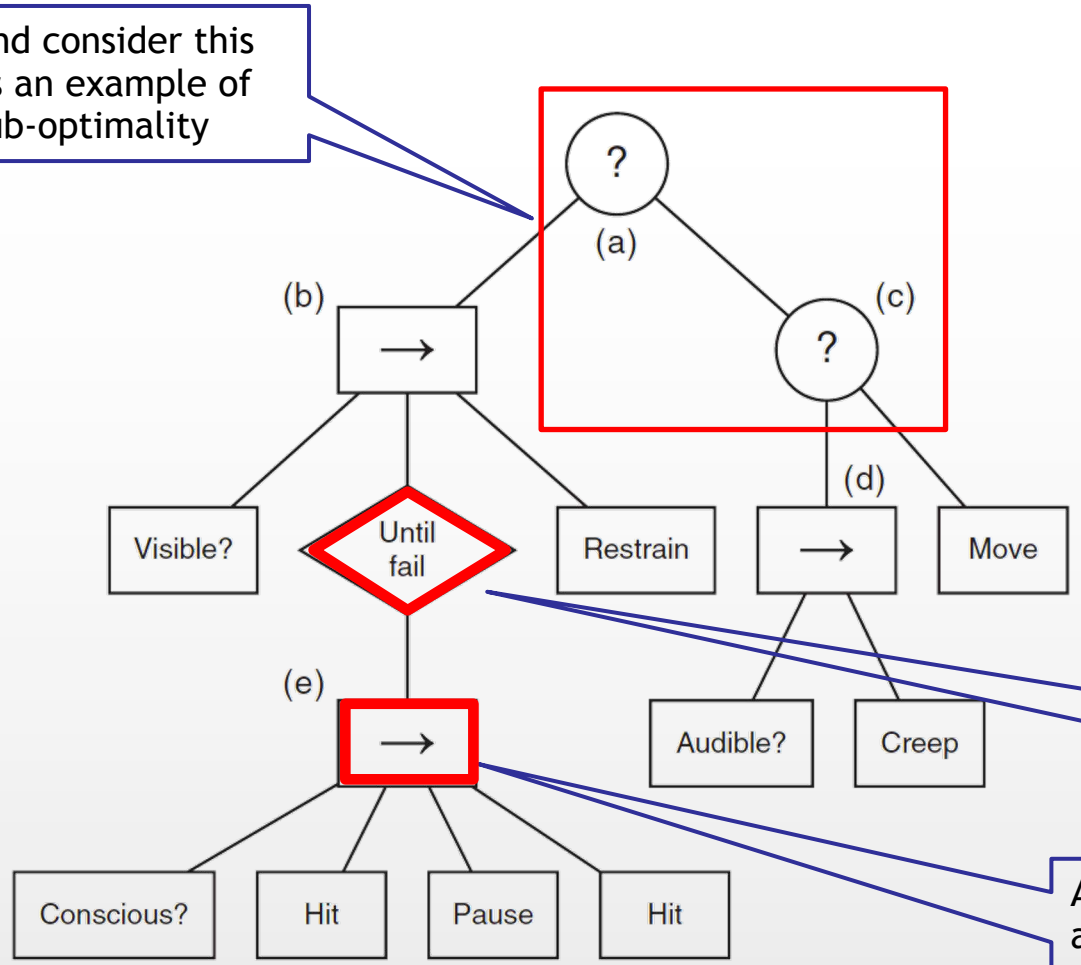
Decorators

- As in software engineering, decorators *wrap* other tasks and modify their behavior
  - A decorator task has always only one child

- We have some standard decorators
  1. Filter
     - Execute the task if and only if a condition is verified
  2. Limit
     - Execute a task at most a limited number of times
  3. Until Fail
     - Keeps executing the task until it fails
  4. Inverter
     - Returns success when the child fails and vice-versa

- Of course, decorators can be nested

Example With Decorators



- In this example we describe the behavior of a guard
- If an intruder is visible, it is beaten until he falls unconscious. Otherwise, if the NPC can just hear it, we start creeping around.
  If the intruder cannot be perceived in any way, just keep moving

Implementing Decorators

```
// Abstract class holding a decorator
public abstract class BTDecorator : IBTTask {

    // Only one child is required
    protected IBTTask child;

    public BTDecorator(IBTTask task) { child = task; }

    public abstract bool Run();
}
```

In the same vein as the composite, the decorator is an abstract class implementing the task interface and allowing a (single) child

Implementing Decorators

A filter extends the decorator abstract class and provides a run method where the Run method of the child is run only if the condition delegate returns true

```
// A task implementing a filter decorator
public class BTDecoratorFilter : BTDecorator {

    // Condition to trigger the filter
    private BTCall Condition;

    public BTDecoratorFilter(BTCall condition, IBTTask task) : base(task) {
        Condition = condition;
    }

    override public bool Run() { return Condition() && child.Run(); }
}
```

I am just too lazy to write an if

Implementing Decorators

```
// A task implementing a limit decorator
public class BTDecoratorLimit : BTDecorator {

    private int maxRepetitions;
    private int count;

    public BTDecoratorLimit(int max, IBTTask task) : base(task) {
        maxRepetitions = max;
        count = 0;
    }

    override public bool Run() {
        if (count < maxRepetitions) {
            count += 1;
            return child.Run();
        }
        return true;
    }
}
```

In the limit decorator we keep a counter to limit repetitions

Implementing Decorators

```
// A task implementing an until fail decorator
public class BTDecoratorUntilFail : BTDecorator {

    public BTDecoratorUntilFail(IBTTask task) : base(task) { ; }

    override public bool Run() { while(child.Run()); return true; } // mind the semicolon
}

// A task implementing an inverter decorator
public class BTDecoratorInverter : BTDecorator {

    public BTDecoratorInverter(IBTTask task) : base(task) { ; }

    override public bool Run() { return !child.Run(); }
}
```

The semicolon here is creating an empty while “doing nothing”.
This is correct because the activity has already been performed while evaluating the while clause

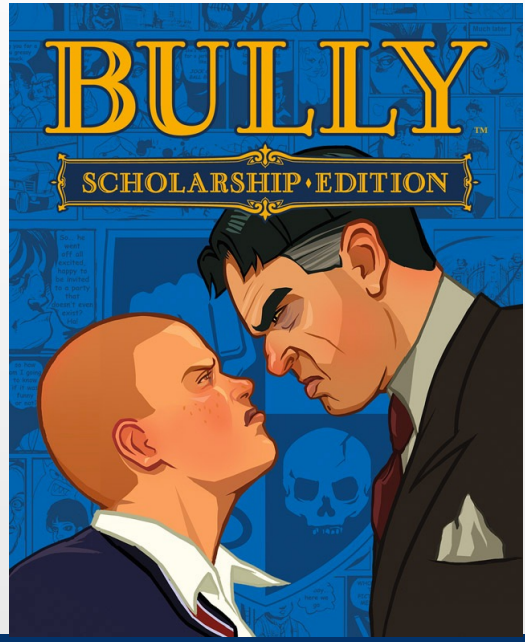
Decorators as Semaphores (in O.S. Terms)

- Some tasks may use limited and/or mutually exclusive resources
  - Running a task unconditionally might jeopardize data consistency
- Possible solutions:
  - Hard-code the test in the behavior
    - This will make your implementation much less portable
  - Create a *condition* task to perform the test and use a *sequence*
    - But then, how do you ensure data consistency while performing a long sequence?
  - Use a *decorator* to guard the resource
    - Make sure the resource is accessed in mutual exclusion only when you need it
- From a C# standpoint, use the System.Threading.Semaphore class inside the decorator and you will be fine

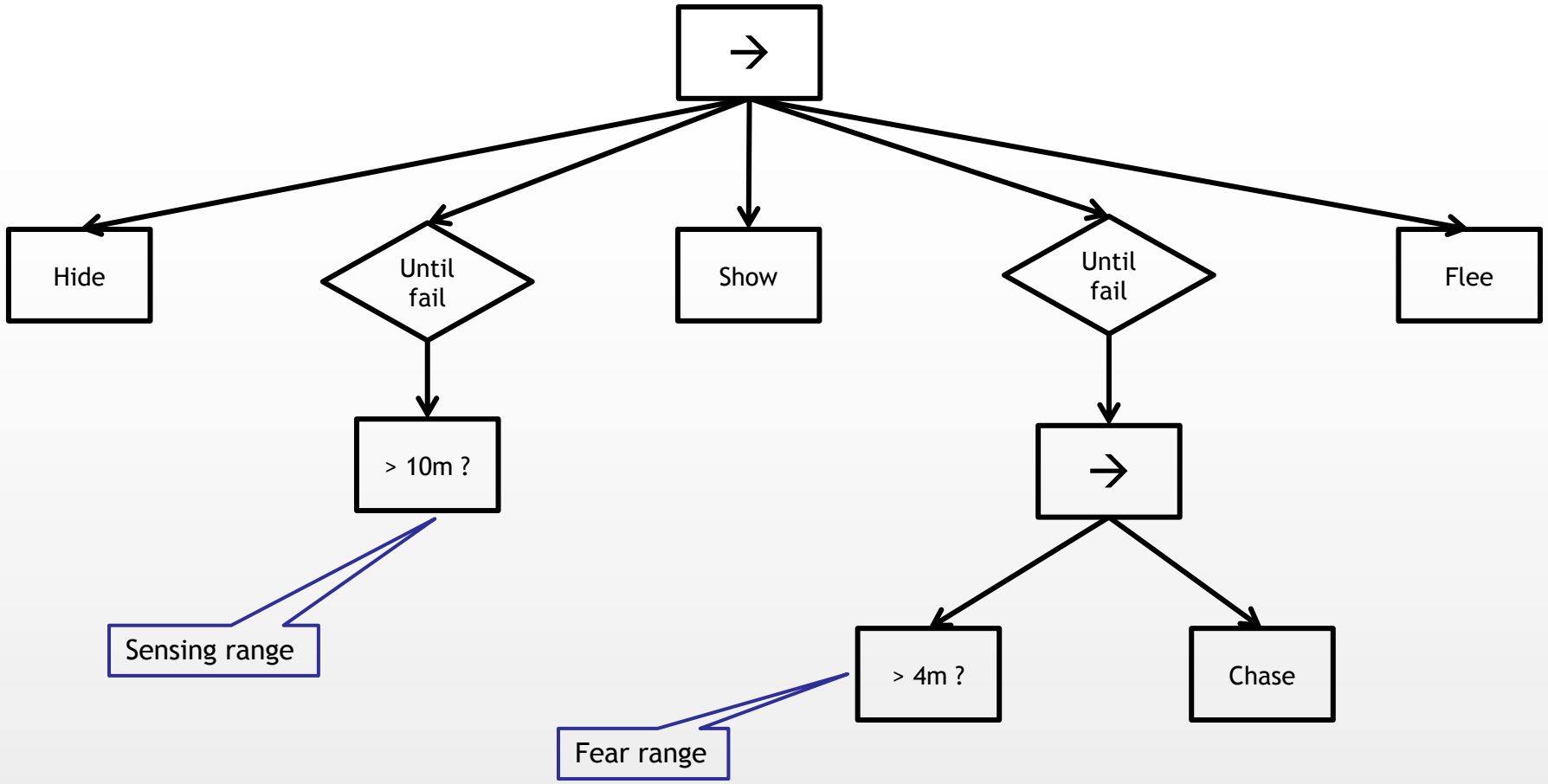
THE ABOVE CONSIDERATIONS DO NOT HOLD IN UNITY!

A Bully Sentinel

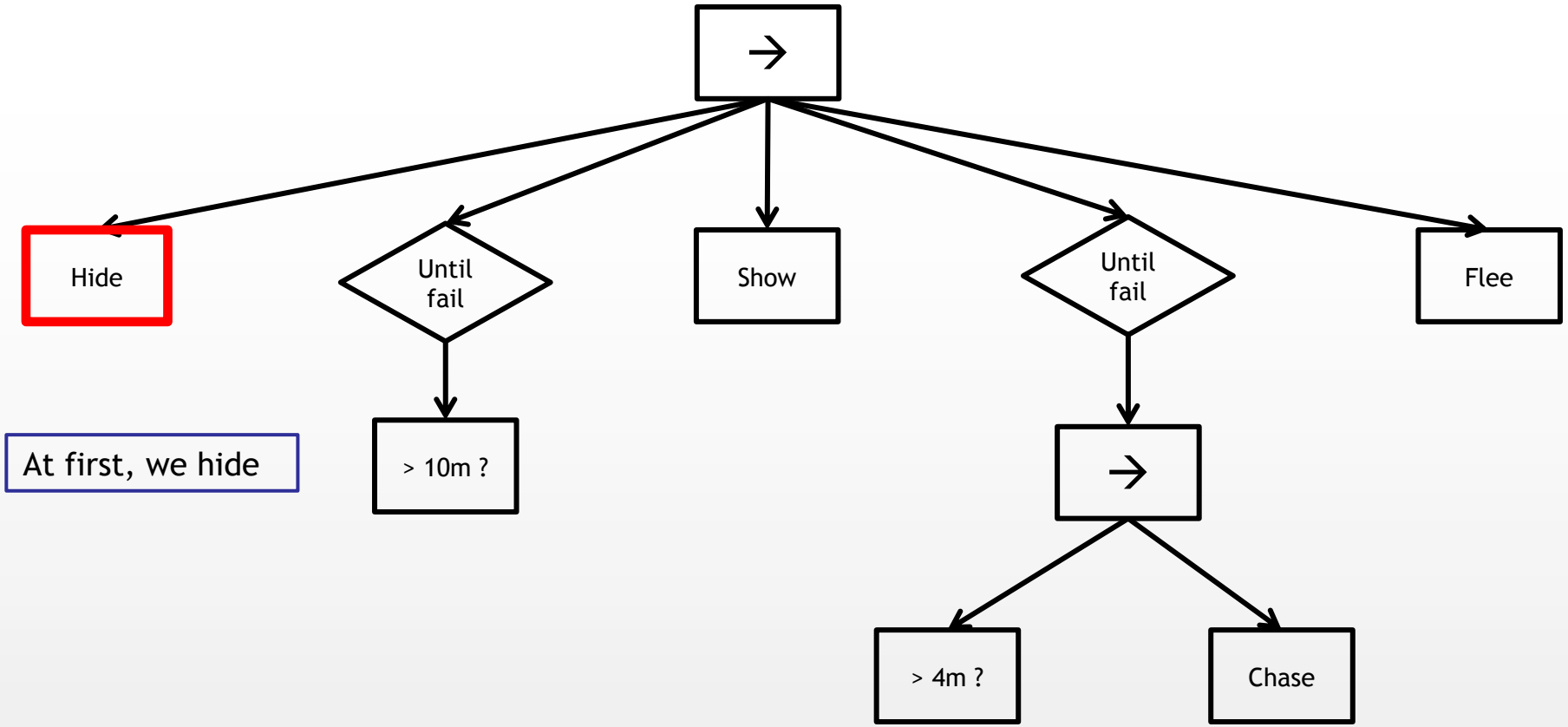
- A sentinel is hidden, waiting for an intruder
- When the intruder comes closer than the sensing range (10 meters) the sentinel pops up and start chasing the intruder
  - Chasing means going as close as 5 meters from the intruder
- If the intruder engages the sentinel, the sentinel escapes and calls for backup
  - Engaging means to go closer than 4 meters to the sentinel



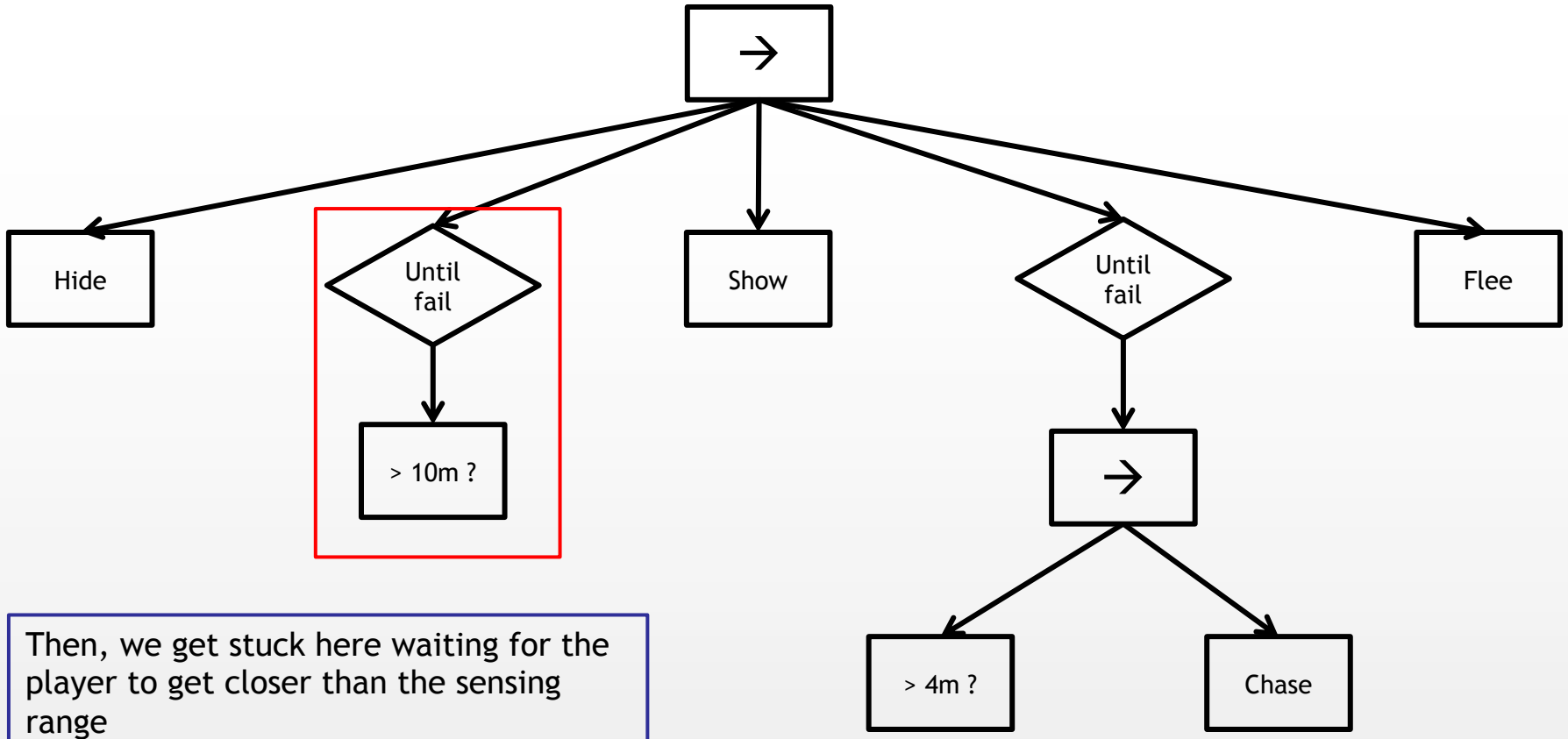
Bully Sentinel



Bully Sentinel



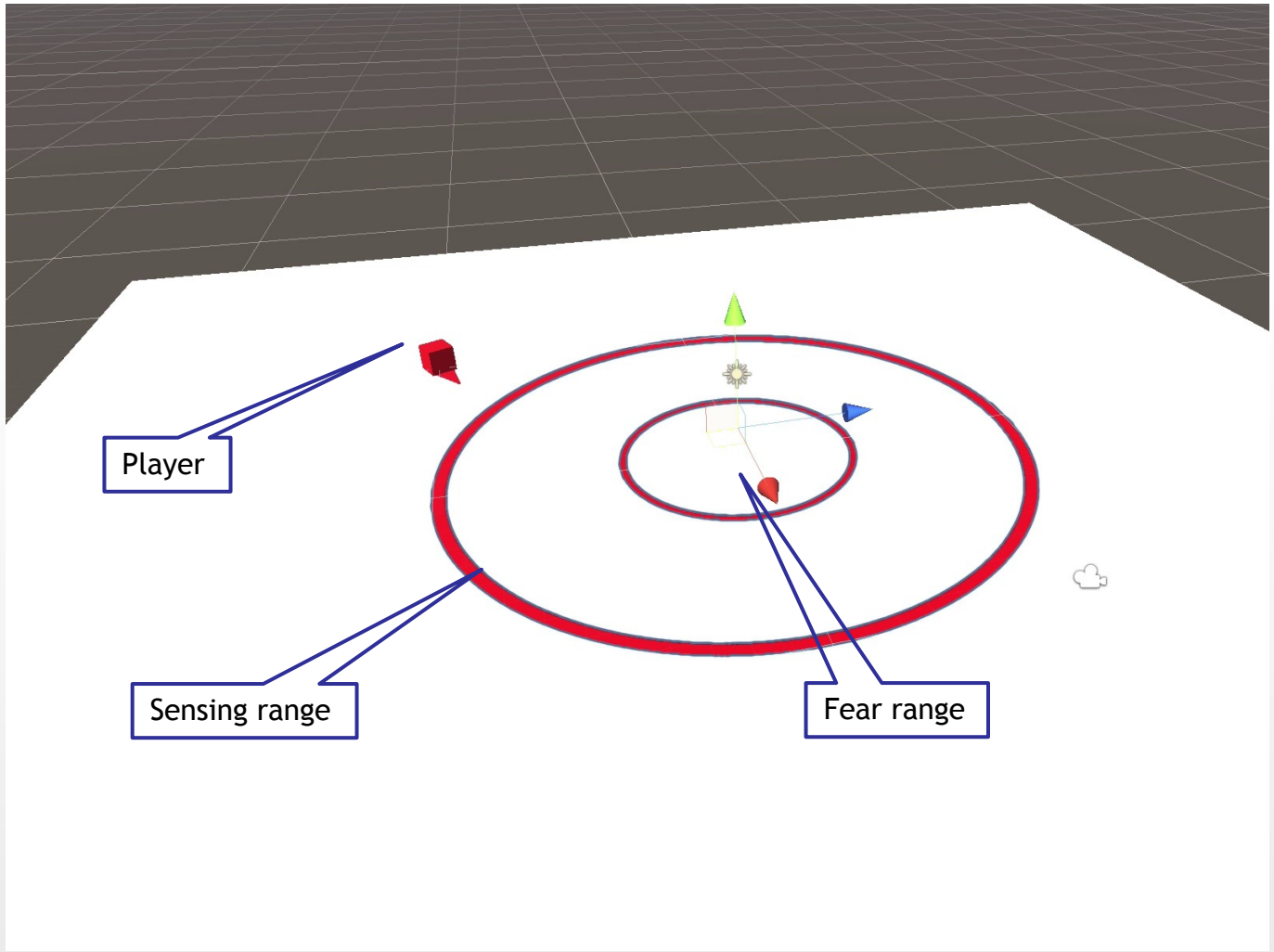
Bully Sentinel



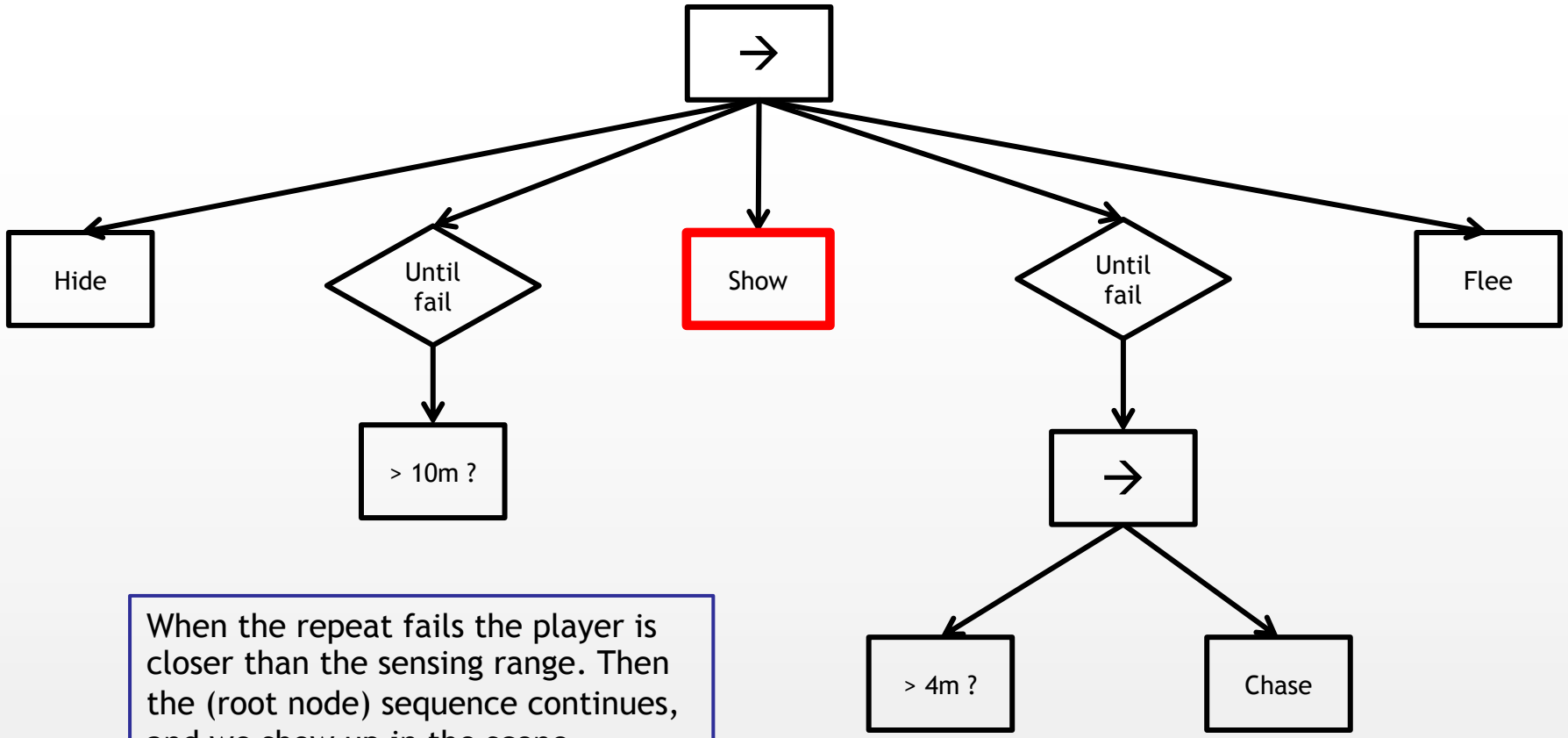
Then, we get stuck here waiting for the player to get closer than the sensing range

Bully Sentinel

Player approaching, sentinel hidden



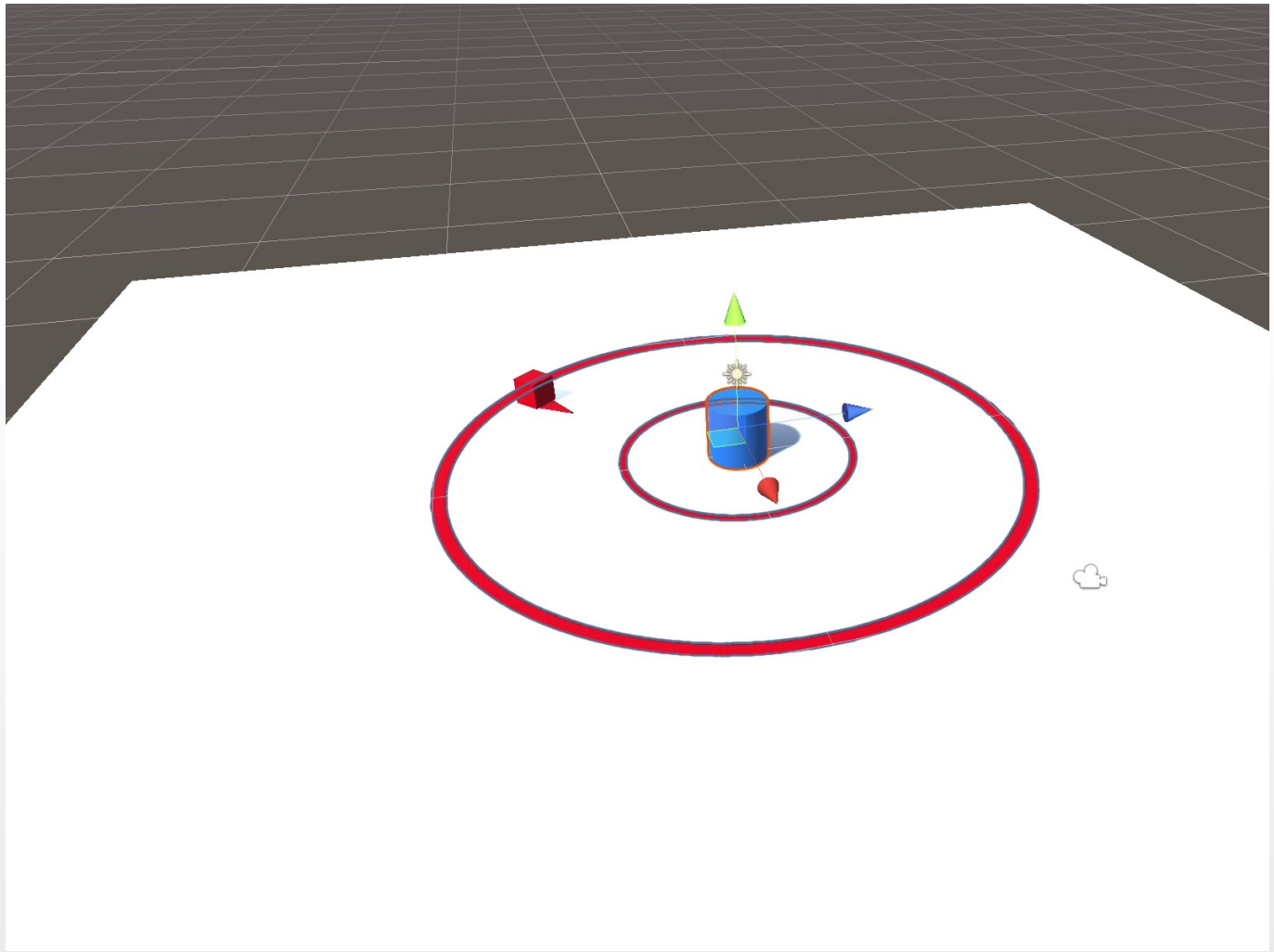
Bully Sentinel



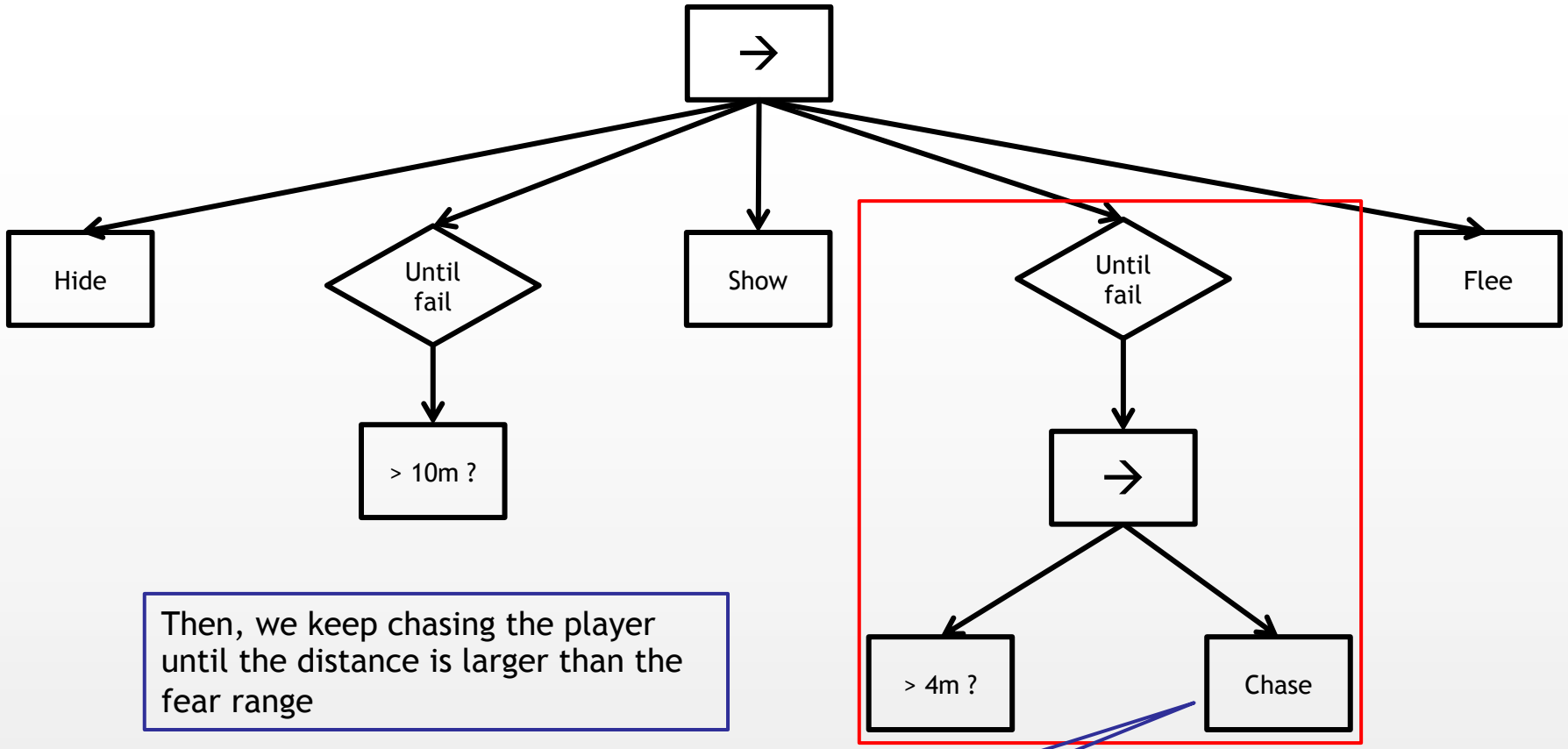
When the repeat fails the player is closer than the sensing range. Then the (root node) sequence continues, and we show up in the scene

Bully Sentinel

Player in sensing range, sentinel visible



Bully Sentinel

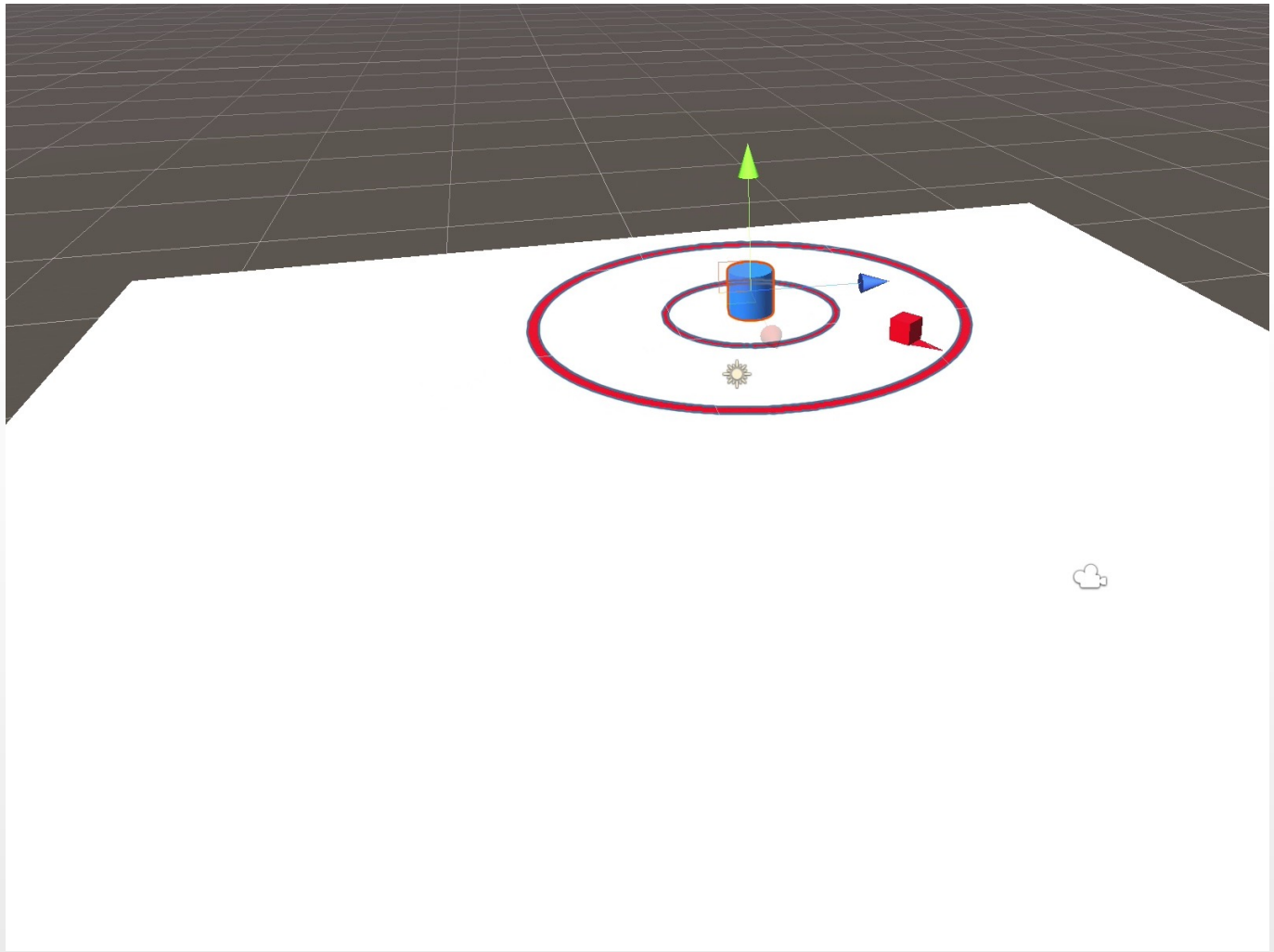


Then, we keep chasing the player until the distance is larger than the fear range

The “stop at 5 meters” logic is embedded in the chase action

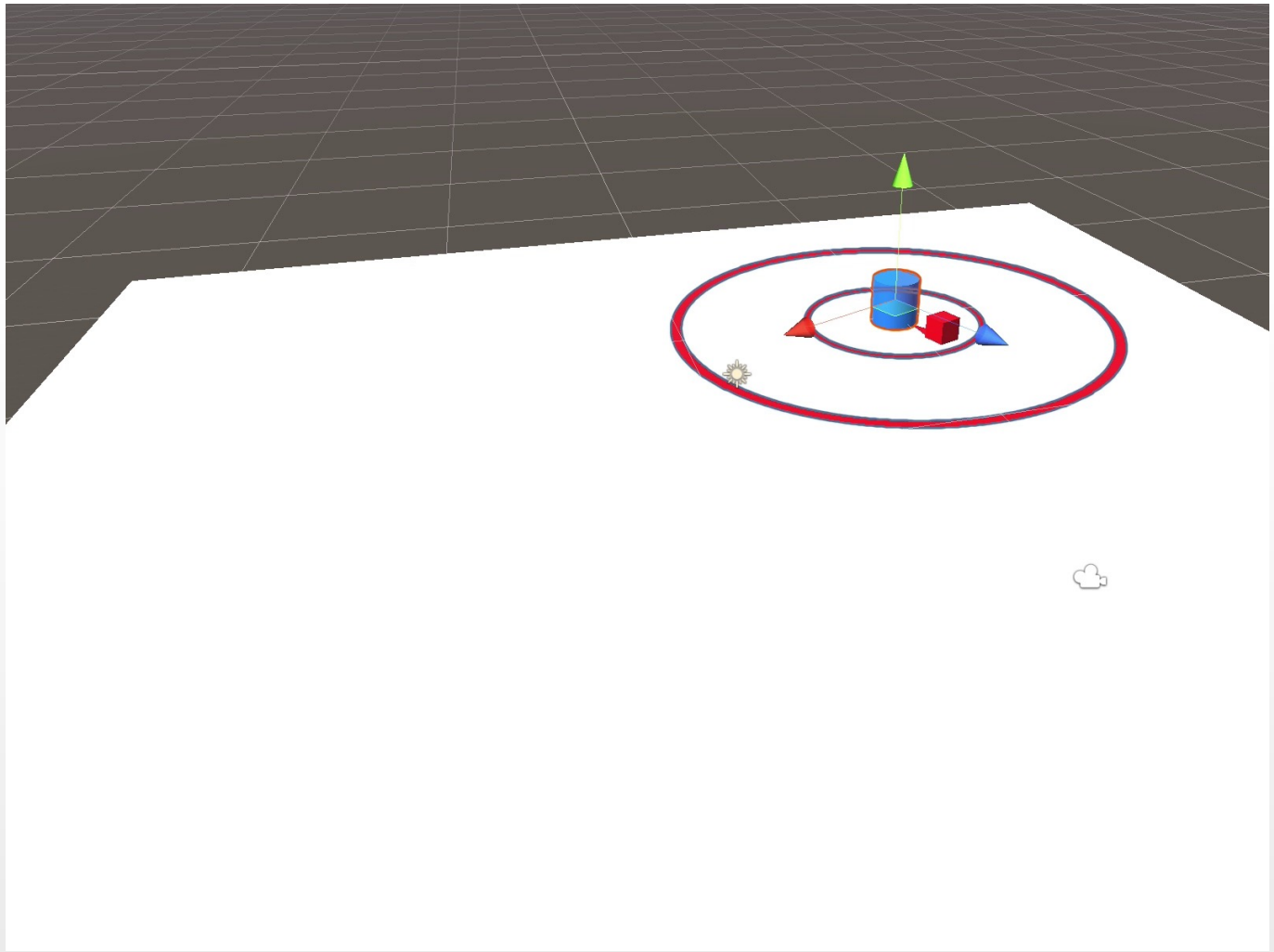
Bully Sentinel

Player moving around, sentinel chasing

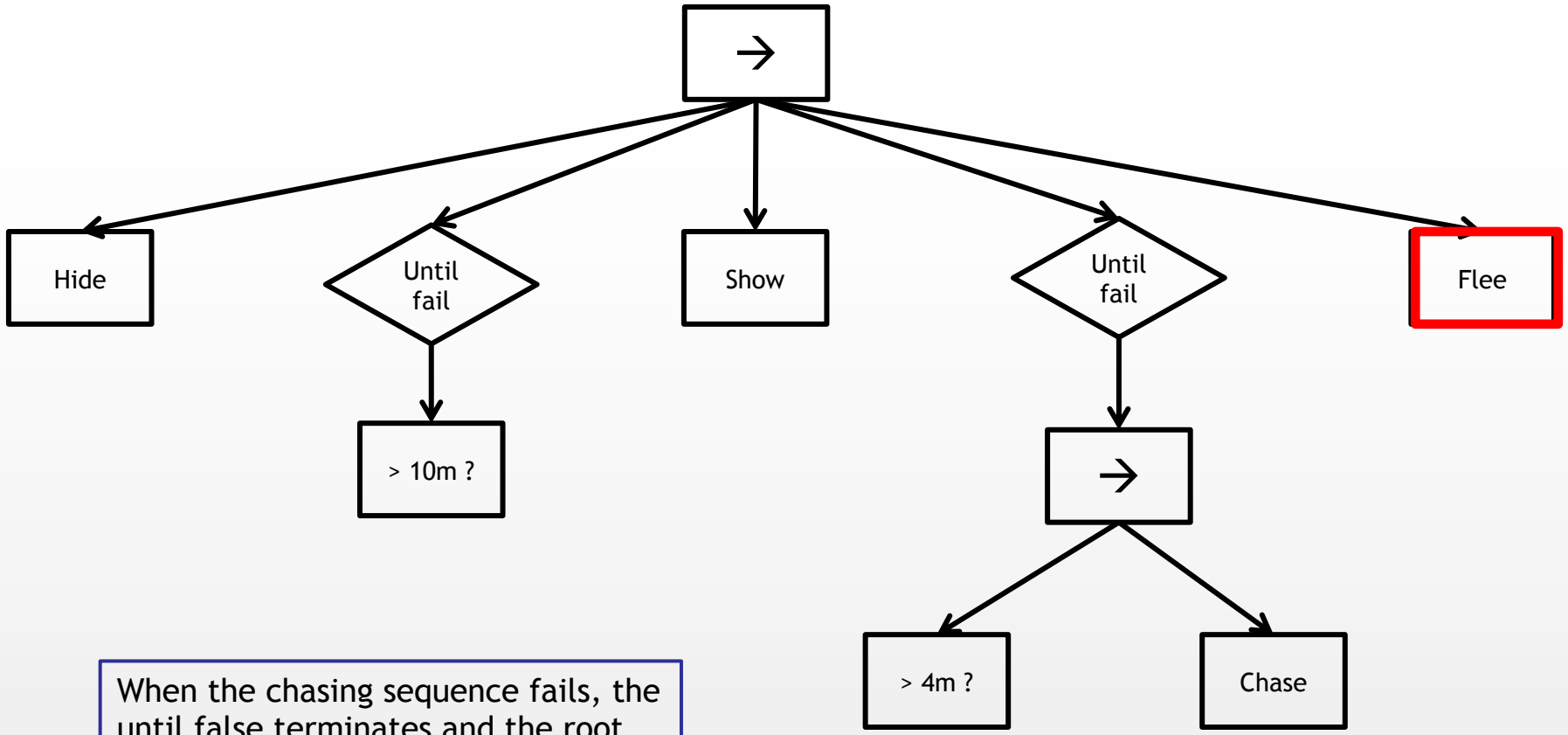


Bully Sentinel

Player moving closer than fear range, sentinel starts feeling



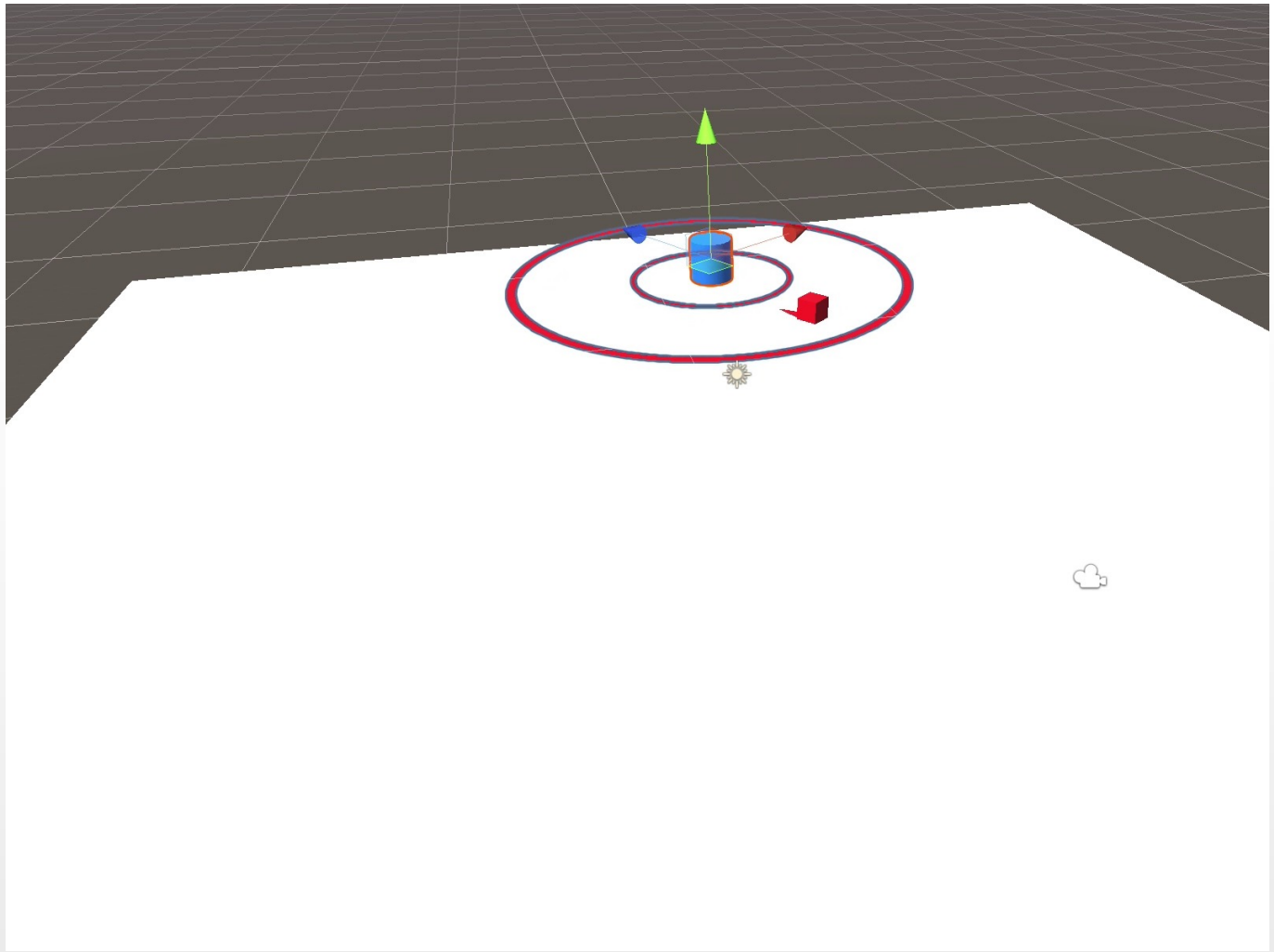
Bully Sentinel



When the chasing sequence fails, the until false terminates and the root sequence continues. So, the sentinel starts to flee

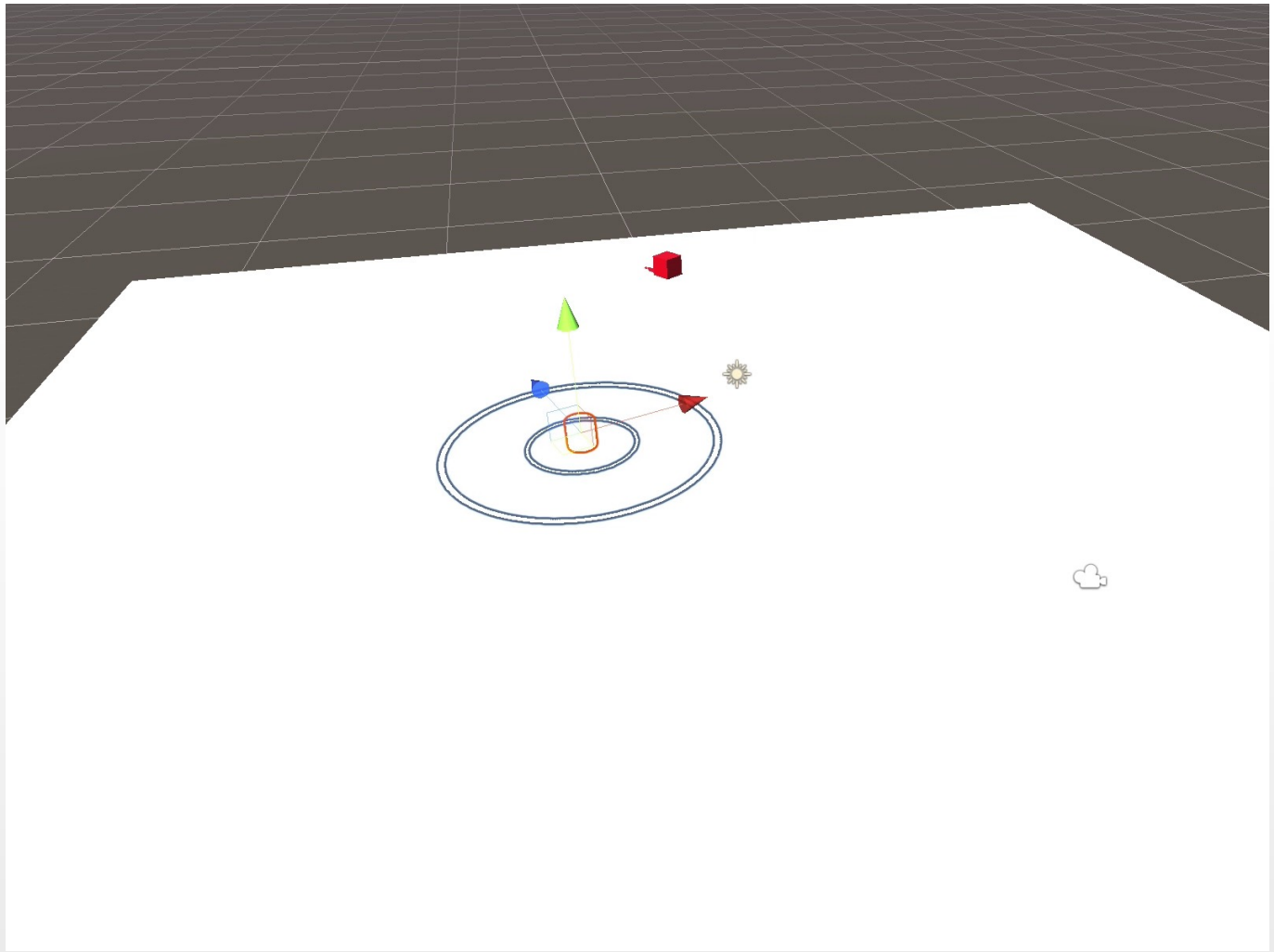
Bully Sentinel

Player chasing, sentinel feeling



Bully Sentinel

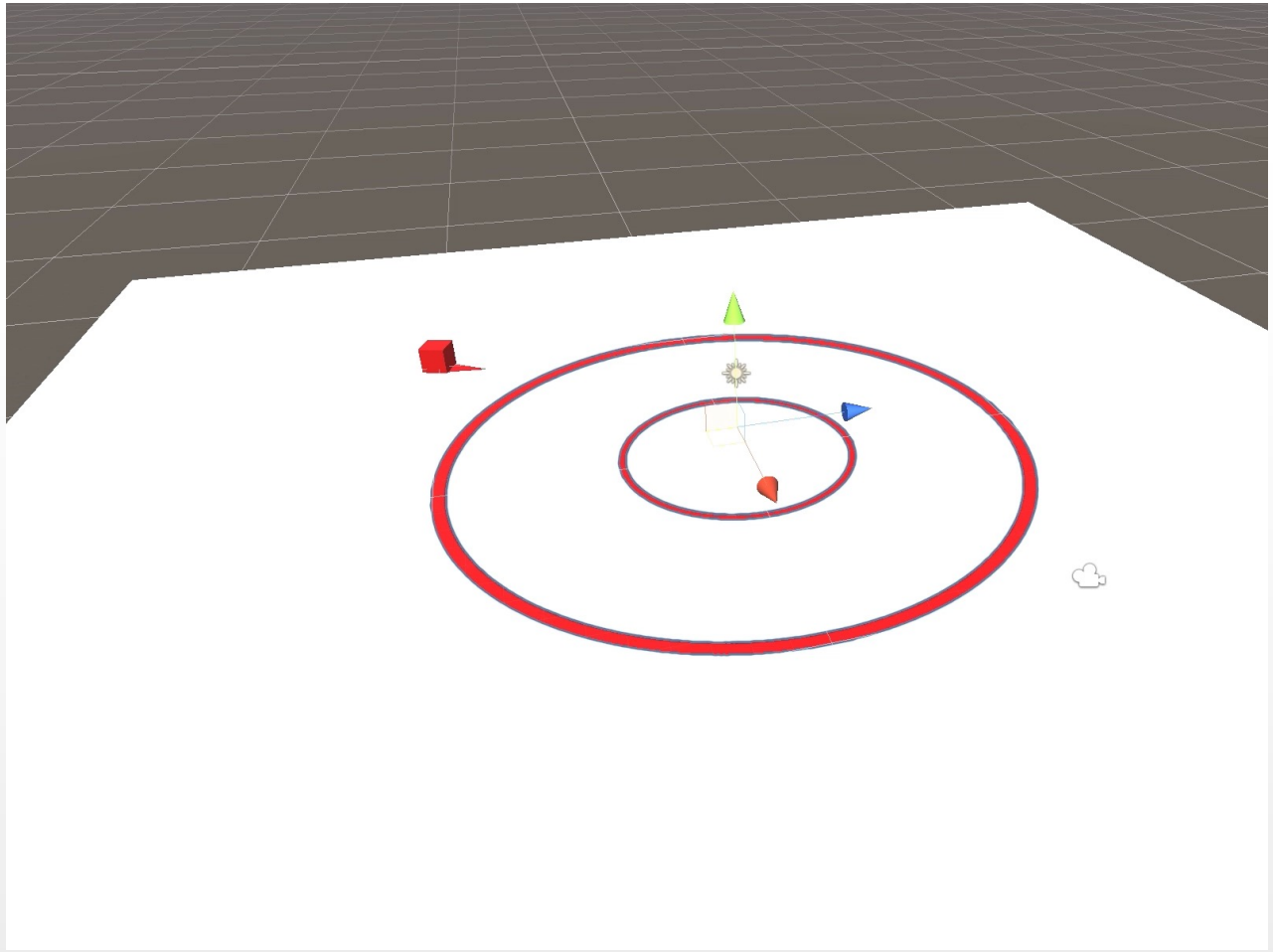
Player chasing, sentinel flying



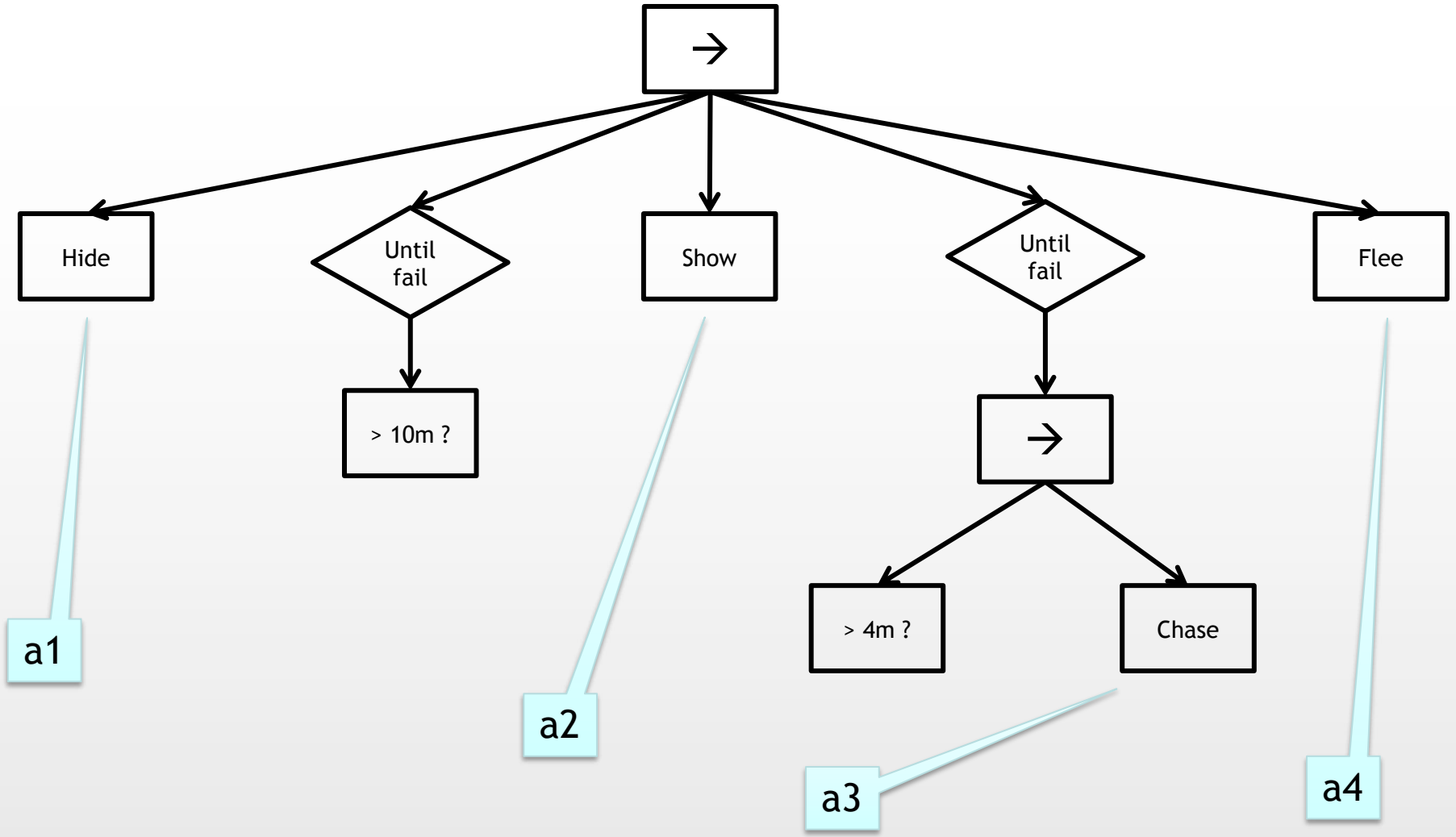
Unfortunately, the sentinel's behaviour tree is not smart enough to also keep it on the platform

The End of a Bully

Scene: Bully
Folder: Behaviour Trees



Actions for Bully Sentinel



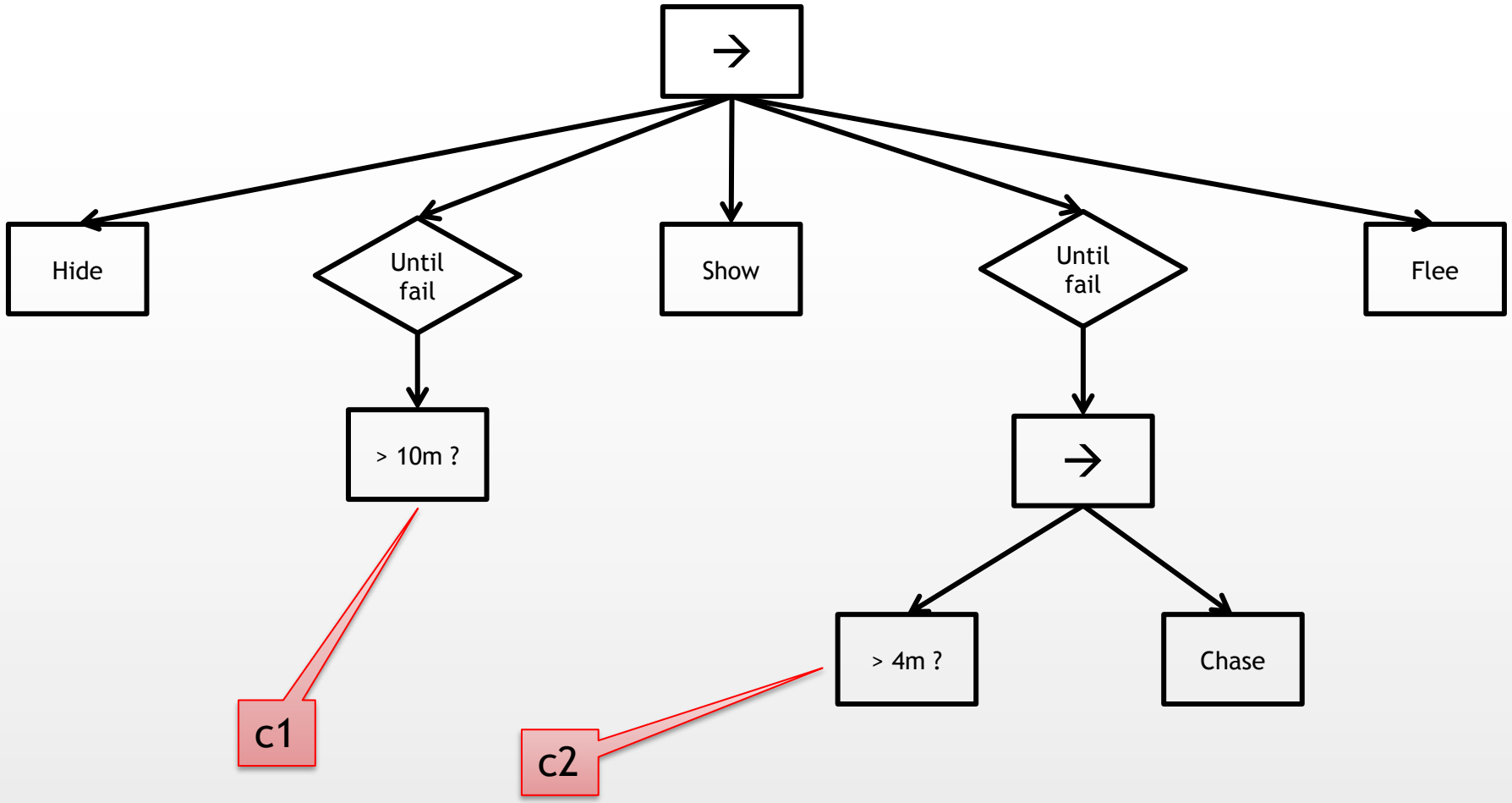
Actions for Bully Sentinel

```
// ACTIONS

public bool Hide() {
    GetComponent<MeshRenderer> ().enabled = false;
    return true;
}


public bool Show() {
    GetComponent<MeshRenderer> ().enabled = true;
    return true;
}

public bool Chase() {
    GetComponent<SeekBehaviour> ().destination = target;
    GetComponent<FleeBehaviour> ().destination = null;
    return true;
}

public bool Flee() {
    GetComponent<SeekBehaviour> ().destination = null;
    GetComponent<FleeBehaviour> ().destination = target;
    return true;
}
```

We just set target for seek and flee in order to start the agent movement.

We will see these components in detail when dealing with movement

Conditions for Bully Sentinel

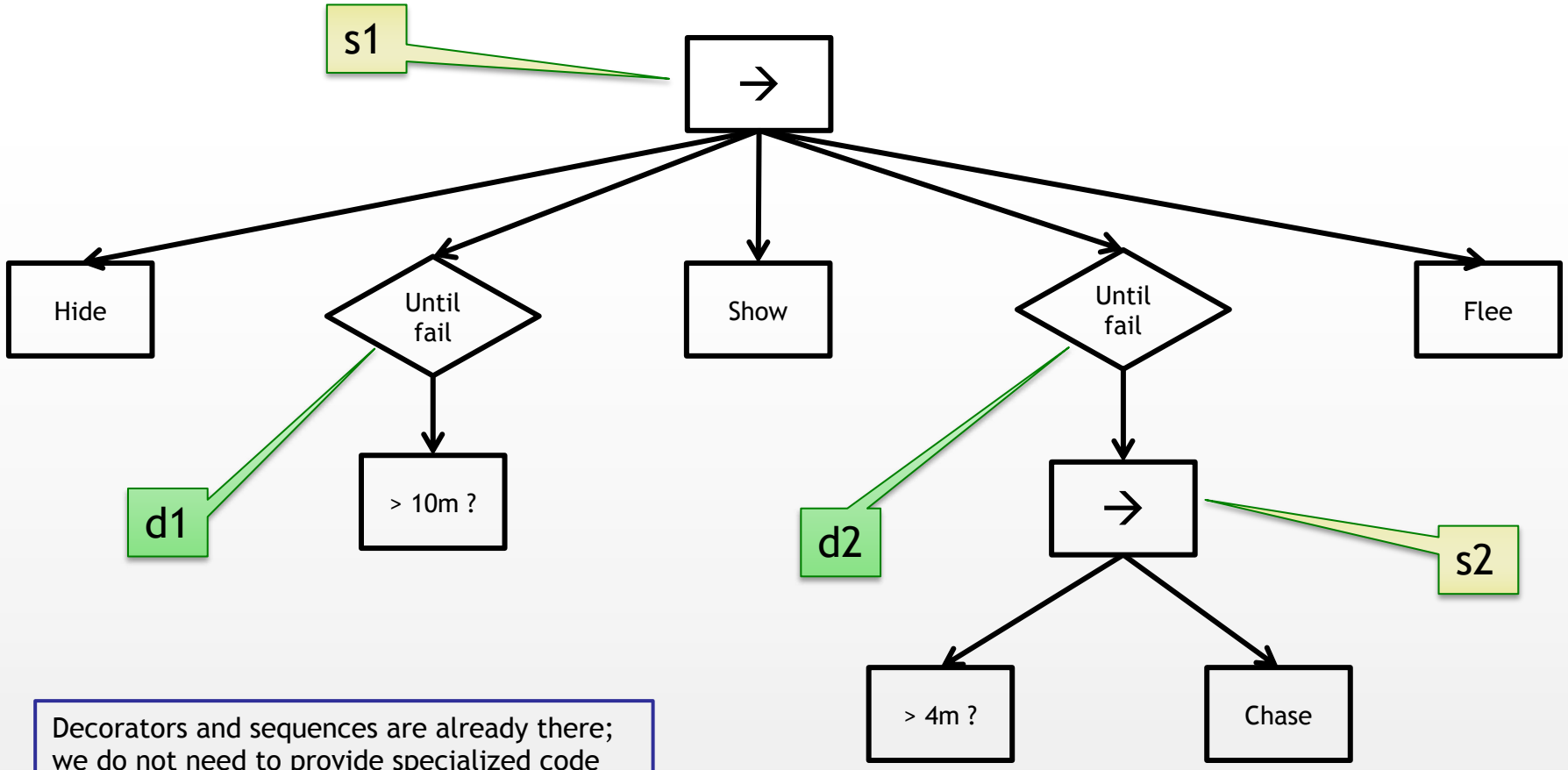


Conditions for Bully Sentinel

This is the same logic as in the other examples involving sentinels and sensing range

```
// CONDITIONS

private Transform target;

public bool EnemyOutsideRange() {
    foreach (GameObject go in GameObject.FindGameObjectsWithTag(targetTag)) {
        if ((go.transform.position - transform.position).magnitude < sensingRange) {
            target = go.transform;
            return false;
        }
    }
    return true;
}

public bool EnemyNotTooClose() {
    return (target.transform.position - transform.position).magnitude > fearRange;
}
```

Decorators and Composites for Bully Sentinel



Decorators and sequences are already there; we do not need to provide specialized code for them.

NOTE: Same point holds for selectors

Final Result: STUCK!

```
[Range (0f, 20f)] public float sensingRange = 10f;
[Range (0f, 20f)] public float fearRange = 4f;
public string targetTag = "Player";

void Start () {
    BTAction a1 = new BTAction (Hide);
    BTAction a2 = new BTAction (Show);
    BTAction a3 = new BTAction (Chase);
    BTAction a4 = new BTAction (Flee);

    BTCondition c1 = new BTCondition (EnemyOutsideRange);
    BTCondition c2 = new BTCondition (EnemyNotTooClose);

    BTDecorator d1 = new BTDecoratorUntilFail (c1);

    BTSequence s2 = new BTSequence (new IBTTask[] { c2, a3 });
    BTDecorator d2 = new BTDecoratorUntilFail (s2);

    BTSequence s1 = new BTSequence (new IBTTask[] { a1, d1, a2, d2, a4 });

    BehaviorTree AI = new BehaviorTree(s1);

    AI.Run (); // System stops HERE!
}
```

The creation of the tree is correct, but the game is simply refusing to run.

At least, it IS running, but is not giving you control over the player

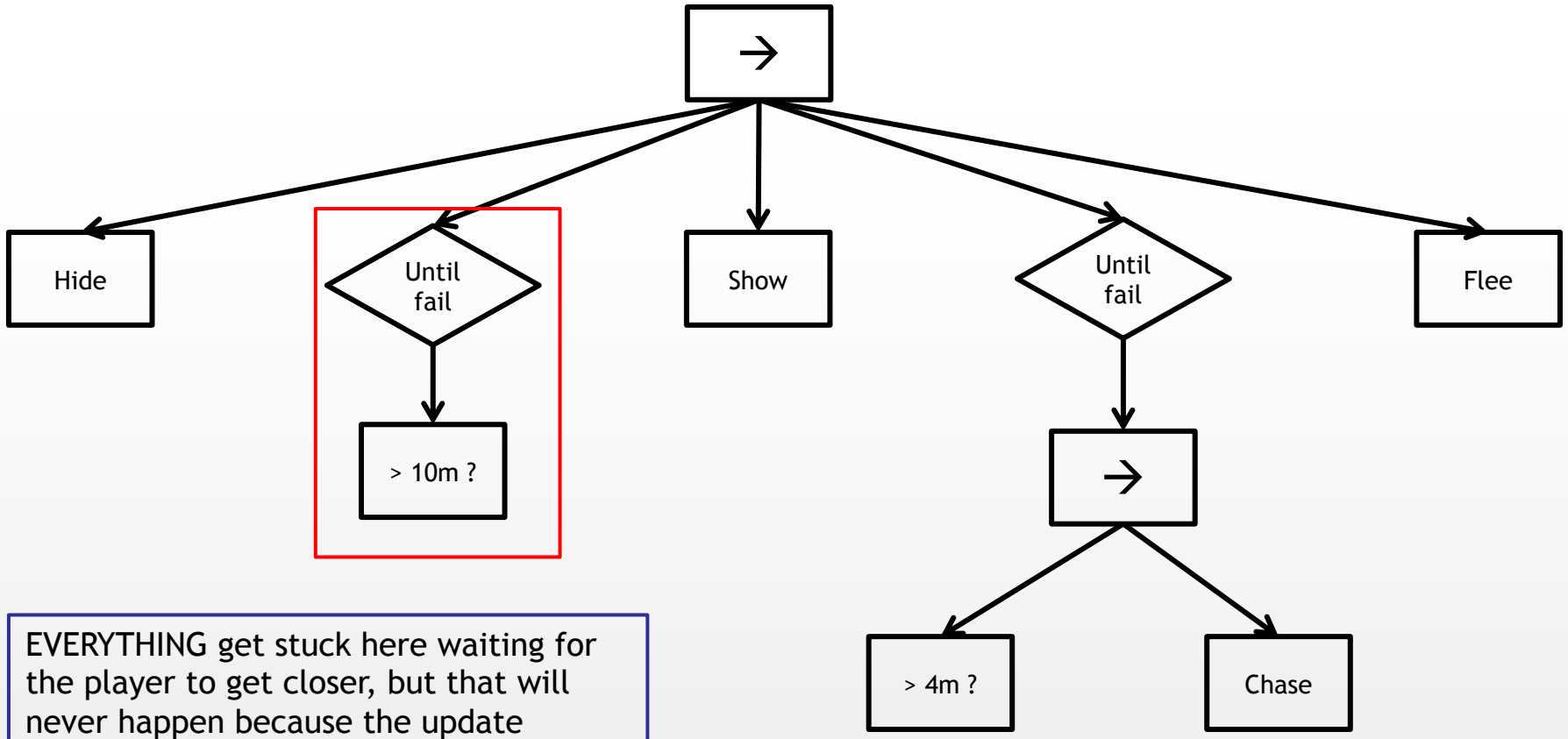
If you already started the scene, it does not get stuck. Check, and see the script I use there is not this one :)

I am sure you will forgive me ... eventually

Why is This not Working?

- Unity, as already said many times, is executed as a single thread due to data integrity and performance reasons
- When you start the game, the Behaviour Tree is evaluated and, when it start waiting for the player to get closer than the sensing range, the control is never released back to the engine
- The really bad news is that **you cannot run it as a multithreaded application**
  - Multithreading is possible in Unity (yes!), but only the main thread can interact with scene objects
  - You can defer computational work (typically number crunching) to secondary thread, but nothing more than that
  - You cannot spawn a secondary thread to evaluate the tree since you will not be able to “sense” what is on the scene (e.g., the distance between sentinel and player)

Bully Sentinel is a Single Thread



EVERYTHING get stuck here waiting for the player to get closer, but that will never happen because the update method of the player controller cannot be executed

How to Make it Work?

Source: CRBehaviourTree
Folder: Behaviour Trees

- Our only option is to revert to coroutines and use the main thread for the rest of the game
  - Nevertheless, we cannot just evaluate the tree from a coroutine, because the coroutine is run inside the main thread and we will get stuck .. again
  - We must create a coroutine to evaluate the tree one step at a time and releasing each time the control to the main thread

- First, we must modify the code to make it coroutine-compliant

```
namespace CRBT {

    public delegate bool BTCall();

    public abstract class IBTTask {
        //  0 -> fail
        //  1 -> succes
        // -1 -> call me again
        public abstract int Run();
    }
```

To avoid changing the name to all classes we already described, I opted to put classes with the same name in a different namespace.
MonoBT for the monothread version.
CRBT for the coroutine version

Semantic change here!

Implementation with Coroutines

```
public class BTCondition : IBTTask {

    public BTCall Condition;

    public BTCondition(BTCall call) { Condition = call; }

    override public int Run() { return Condition() ? 1 : 0; }
}

public class BTAction : IBTTask {

    public BTCall Action;

    public BTAction(BTCall call) { Action = call; }

    override public int Run() { return Action() ? 1 : 0; }
}
```

Either fail or succeed.
No need to call again

Delegates are provided by the user. We cannot force our semantic outside the library. Anyway, we must always suppose condition and action delegates will run for a short time without stopping the overall execution

Implementation with Coroutines

```
public class BTSelector : BTComposite {

    public BTSelector(IBTTask[] tasks) : base(tasks) { ; }

    override public int Run() {
        while (index < children.Length) {
            int v = children[index].Run();
            if (v == -1) { return -1; }
            if (v == 0) { index += 1; return -1; }
            if (v == 1) { index = 0; return 1; }
        }
        // Otherwise the selector fails
        index = 0;
        return 0;
    }
}

public class BTSequence : BTComposite {

    public BTSequence(IBTTask[] tasks) : base(tasks) { ; }

    override public int Run() {
        while (index < children.Length) {
            int v = children[index].Run();
            if (v == -1) { return -1; }
            if (v == 0) { index = 0; return 0; }
            if (v == 1) { index += 1; return -1; }
        }
        // Otherwise the selector succeed
        index = 0;
        return 1;
    }
}
```

Differently from the previous version, we must keep an index to remember which child was run last time

A selector returns failure if all children failed, success in case one child is successful, and to be called again in all other cases.
Other cases are a child needing to be called again or switching to the next child

A sequence return failure if one children fails, success if all children are successful, and to be called again in all other cases.
Other cases are a child needing to be called again or switching to the next child

Implementation with Coroutines

```
public class BTDecoratorFilter : BTDecorator {

    private BTCall Condition;

    public BTDecoratorFilter(BTCall condition, IBTTask task) : base(task) {
        Condition = condition;
    }

    override public int Run() { return Condition() ? Child.Run() : 0; }
}

public class BTDecoratorLimit : BTDecorator {

    public int maxRepetitions;
    public int count;

    public BTDecoratorLimit(int max, IBTTask task) : base(task) {
        maxRepetitions = max;
        count = 0;
    }

    override public int Run() {
        if (count >= maxRepetitions) return 0;
        int v = Child.Run();
        if (v != -1) count += 1;
        return v;
    }
}
```

In a filter decorator, if the condition is met, the return value of the child is returned. This way, the filter will be called again if the child needs to be called again.

Here we are assuming the condition is not going to change over time. To avoid this, we can just call the condition delegate once and cache the result.

In a limit decorator we will always return the value returned by the child, but the counter will be updated only if the child has returned a result (no matter if it is success or failure)

Implementation with Coroutines

```
public class BTDecoratorUntilFail : BTDecorator {

    public BTDecoratorUntilFail(IBTTask task) : base(task) { ; }

    override public int Run() {
        if (Child.Run() != 0) return -1;
        return 1;
    }
}

public class BTDecoratorInverter : BTDecorator {

    public BTDecoratorInverter(IBTTask task) : base(task) { ; }

    override public int Run() {
        int v = Child.Run();
        if (v == 1) return 0;
        if (v == 0) return 1;
        return v; // -1
    }
}
```

The until fail decorator will ask to be called over and over until the child reports failure

The inverted decorator will return an inverted value unless the child is not asking to be called again

Implementation with Coroutines

```
public class BehaviorTree {

    public IBTTask root;

    public BehaviorTree(IBTTask task) { root = task; }

    public bool Step() {
        return root.Run() < 0 ? true : false;
    }
}
```

The Behaviour Tree will return true if another processing step is required and false otherwise (no matter the return value of the root node).

This is not a change in semantic, but just a convenience to iterate steps during evaluation

Implementation with Coroutines

```
public class CRBTSentinel : MonoBehaviour {

    [Range (0f, 20f)] public float sensingRange = 10f;
    [Range (0f, 20f)] public float fearRange = 4f;
    public string targetTag = "Player";
    public float reactionTime = .2f;

    private BehaviorTree AI;

    void Start () {
        BTAction a1 = new BTAction (Hide);
        BTAction a2 = new BTAction (Show);
        BTAction a3 = new BTAction (Chase);
        BTAction a4 = new BTAction (Flee);

        BTCondition c1 = new BTCondition (EnemyOutsideRange);
        BTCondition c2 = new BTCondition (EnemyNotTooClose);

        BTDecorator d1 = new BTDecoratorUntilFail (c1);

        BTSequence s2 = new BTSequence (new IBTTask[] { c2, a3 });
        BTDecorator d2 = new BTDecoratorUntilFail (s2);

        BTSequence s1 = new BTSequence (new IBTTask[] { a1, d1, a2, d2, a4 });

        AI = new BehaviorTree(s1);

        StartCoroutine(Patrol());
    }

    public IEnumerator Patrol() {
        while (AI.Step()) {
            yield return new WaitForSeconds(reactionTime);
        }
    }
```

Just change here and add a method for the coroutine

This coroutine stops when the Behaviour Tree processing ends

Limitations of Behaviour Trees

- They are unpractical to represent state-based behaviors
  - Every time we start evaluation from the root node. There is no memory information
- Alarm behaviors are difficult to implement
  - Because that is requiring to break the tree syntax or to check for all possible alarms at every single step
  -
- Behaviour Trees require to avoid thinking in term of states and use work pipelines
  - It’s not impossible, just a bit difficult
- Best combination: use an FSM with a BTs connected to each state
  - The current state of the FSM will determine which BT the character is running (i.e., how to represent the current state of action)

References

- On the textbook
  - § 5.4 (excluded 5.4.4, 5.4.5, 5.4.6)

- What we skipped
  - Parallel tasks execution
  - Sharing data between tasks
  - Reusing/cloning trees