

Artificial Intelligence

Lecture 08 – Behavior Trees

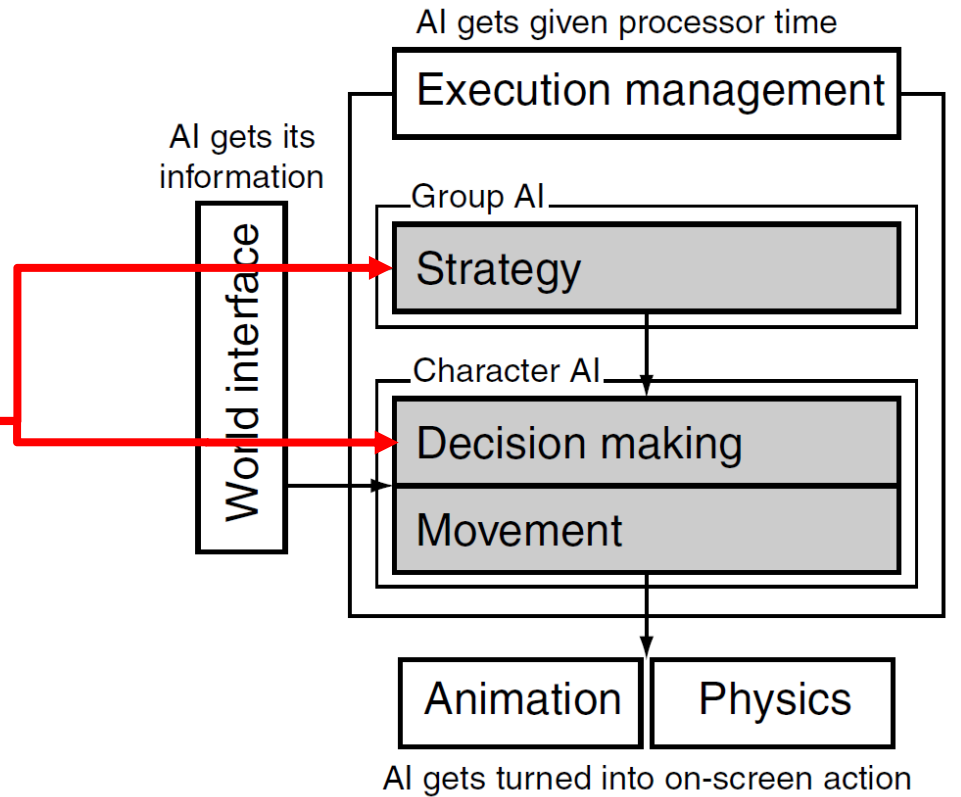
Edirlei Soares de Lima

<edirlei.lima@universidadeeuropeia.pt>



Game AI – Model

- Pathfinding
- Steering behaviours
- Finite state machines
- Automated planning
- **Behaviour trees**
- Randomness
- Sensor systems
- Machine learning



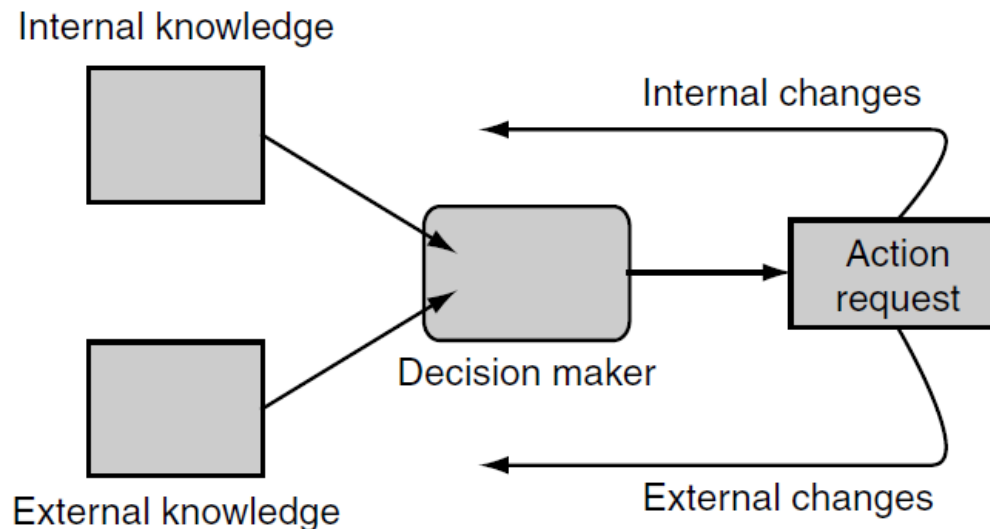
Decision Making

- In game AI, decision making is the ability of a character/agent to decide what to do.
- The agent processes a set of information that it uses to generate an action that it wants to carry out.
 - **Input:** agent's knowledge about the world;
 - **Output:** an action request;



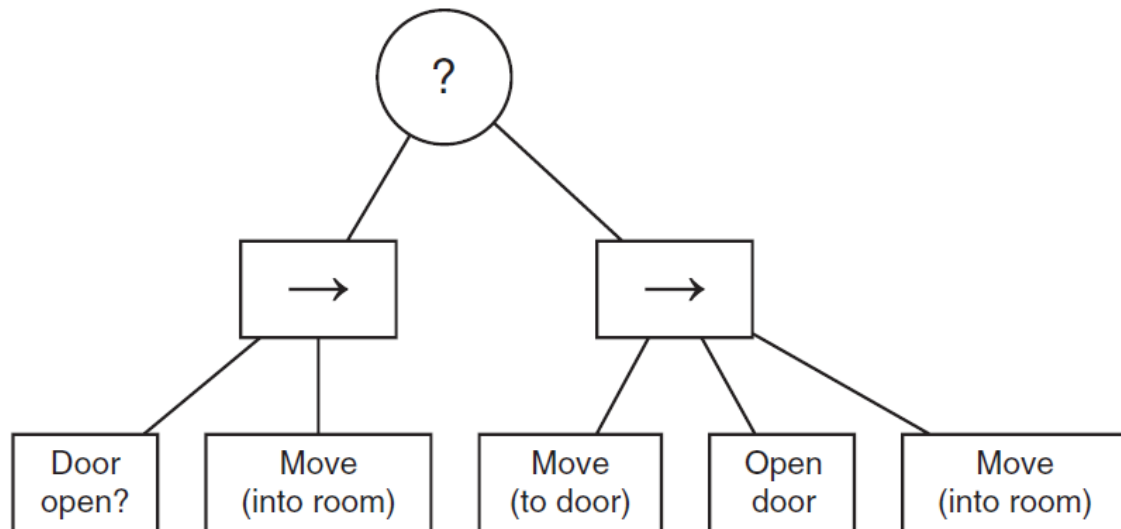
Decision Making

- The knowledge can be broken down into external and internal knowledge.
 - **External knowledge:** information about the game environment (e.g. characters' positions, level layout, noise direction).
 - **Internal knowledge:** information about the character's internal state (e.g. health, goals, last actions).



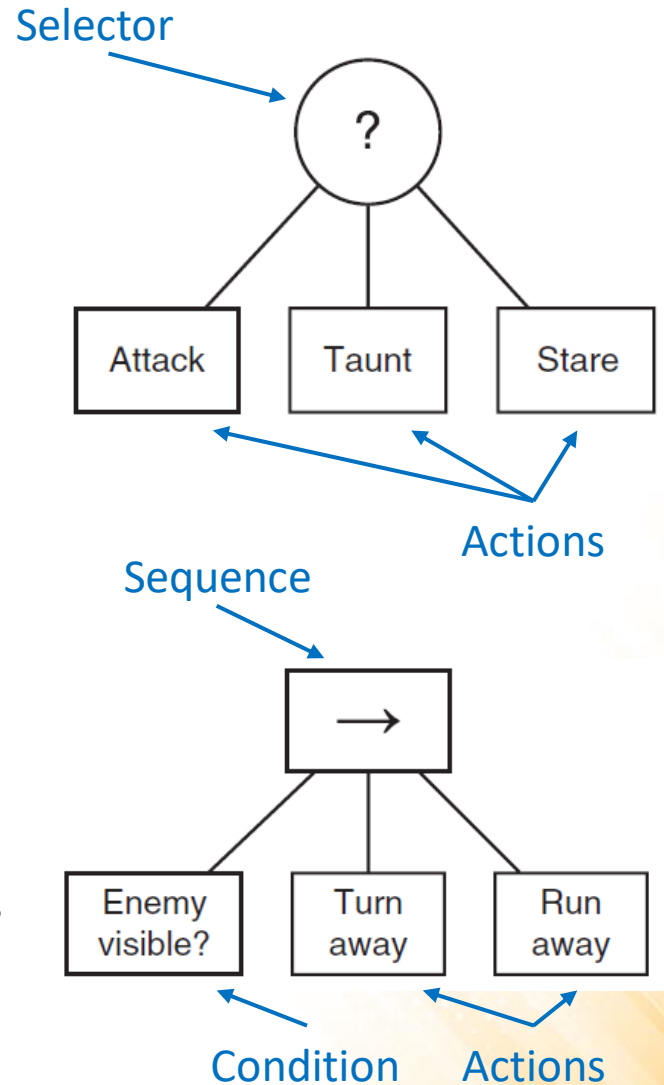
Behavior Tree

- Behavior trees have a lot in common with Hierarchical State Machines but, instead of a state, the main building block of a behavior tree is a task.
 - A task can be something as simple as looking up the value of a variable in the game state, or executing an animation.
 - Tasks can be decomposed into sub-trees to represent more complex actions.

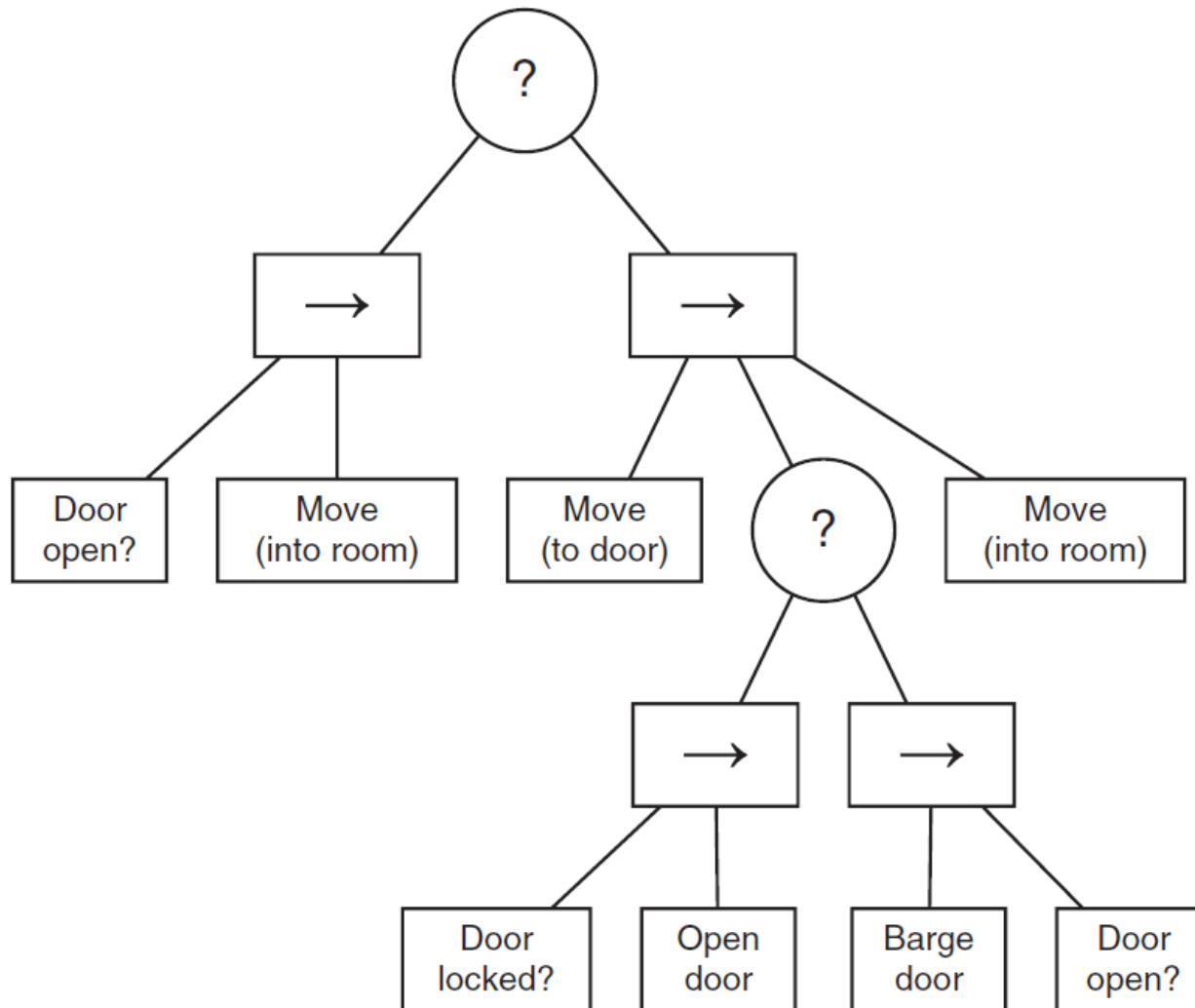


Behavior Tree – Tasks

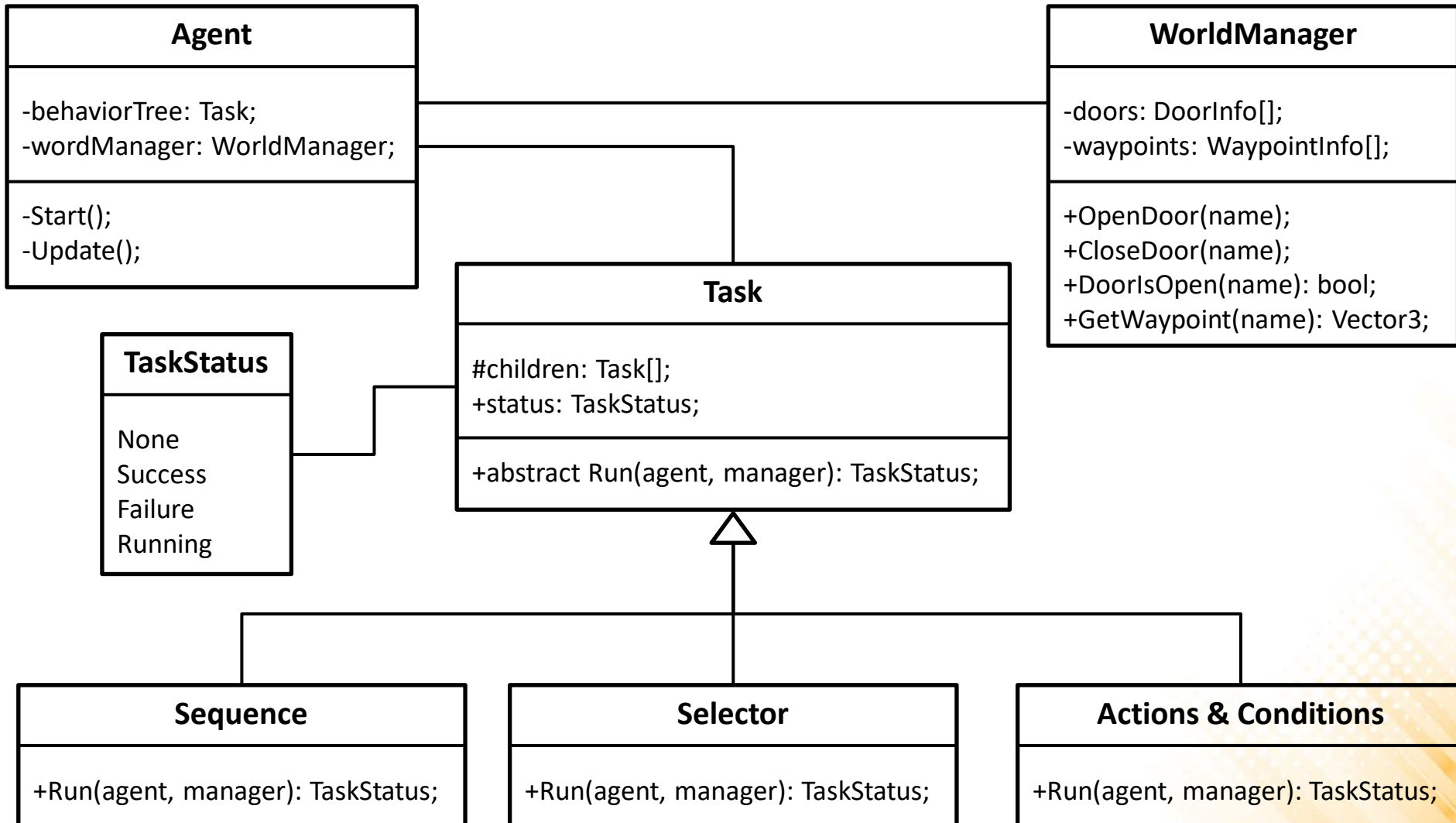
- Behavior trees are composed of three types of tasks:
 - Conditions: test some property of the game (e.g. proximity, line of sight, state of the character).
 - Actions: alter the state of the game (e.g. animation, movement, state change, dialog).
 - Composites: Selector and Sequence.
 - Selector: returns immediately with a success status code when one of its children runs successfully.
 - Sequence: returns immediately with a failure status code when one of its children fails. As long as its children are succeeding, it will keep going.



Behavior Tree – Example



Unity Implementation – Class Diagram



Base Task Class

- **Task Class:**

```
public abstract class Task
{
    protected List<Task> children;
    public TaskStatus status;

    public abstract TaskStatus Run(Agent agent,
                                   WorldManager wordManager);

    public Task(){
        children = new List<Task>();
        status = TaskStatus.None;
    }

    public void AddChildren(Task task){
        children.Add(task);
    }
}
```

Composite Classes

- **Sequence Class:**

```
public class Sequence : Task {  
  
    public override TaskStatus Run(Agent agent,  
                                   WorldManager wordManager){  
        int successCount = 0;  
        foreach (Task task in children){  
            if (task.status != TaskStatus.Success){  
                TaskStatus childrenStatus = task.Run(agent, wordManager);  
                if (childrenStatus == TaskStatus.Failure){  
                    status = TaskStatus.Failure;  
                    return status;  
                }  
            }  
            else if (childrenStatus == TaskStatus.Success){  
                successCount++;  
            }  
  
            ...  
        }  
    }  
}
```

Composite Classes

```
    ...  
  
    else{  
        break;  
    }  
}  
else{  
    successCount++;  
}  
}  
if (successCount == children.Count)  
    status = TaskStatus.Success;  
else  
    status = TaskStatus.Running;  
return status;  
}  
}
```

Composite Classes

- **Selector Class:**

```
public class Selector : Task {  
  
    public override TaskStatus Run(Agent agent,  
                                   WorldManager wordManager){  
  
        int failureCount = 0;  
        foreach (Task task in children){  
            if (task.status != TaskStatus.Failure){  
                TaskStatus childrenStatus = task.Run(agent, wordManager);  
                if (childrenStatus == TaskStatus.Success){  
                    status = TaskStatus.Success;  
                    return status;  
                }  
            }  
            else if (childrenStatus == TaskStatus.Failure){  
                failureCount++;  
            }  
  
            ...  
        }  
    }  
}
```

Composite Classes

```
        ...  
  
        else{  
            break;  
        }  
    }  
}  
if (failureCount == children.Count)  
    status = TaskStatus.Failure;  
else  
    status = TaskStatus.Running;  
return status;  
}  
}
```

Condition Classes

- **DoorOpenCondition Class:**

```
public class DoorOpenCondition : Task {
    private string doorName;

    public DoorOpenCondition(string door) {
        doorName = door;
    }

    public override TaskStatus Run(Agent agent,
                                   WorldManager wordManager) {
        if (wordManager.DoorIsOpen(doorName)) {
            status = TaskStatus.Success;
        }
        else{
            status = TaskStatus.Failure;
        }
        return status;
    }
}
```

Action Classes

- **MoveAction Class:**

```
public class MoveAction : Task{
    private string destination;

    public MoveAction(string dest){
        destination = dest;
    }

    public override TaskStatus Run(Agent agent,
                                   WorldManager wordManager){
        NavMeshAgent navMeshAgent = agent.GetComponent<NavMeshAgent>();
        Vector3 dest = wordManager.GetWaypoint(destination).position;
        if (status == TaskStatus.None){
            navMeshAgent.destination = dest;
            status = TaskStatus.Running;
        }
        else if (IsAtDestination(navMeshAgent)){
            status = TaskStatus.Success;
        }
        return status;
    }
}
```

Action Classes

```
...

private bool IsAtDestination(NavMeshAgent navMeshAgent){
    if (!navMeshAgent.pathPending){
        if (navMeshAgent.remainingDistance <=
            navMeshAgent.stoppingDistance){
            if (!navMeshAgent.hasPath ||
                navMeshAgent.velocity.sqrMagnitude == 0f){
                return true;
            }
        }
    }
    return false;
}
```


Action Classes

- **OpenDoorAction Class:**

```
public class OpenDoorAction : Task{
    private string doorName;

    public OpenDoorAction(string door){
        doorName = door;
    }

    public override TaskStatus Run(Agent agent,
                                   GameWorldManager wordManager){
        if (!wordManager.DoorIsOpen(doorName)){
            wordManager.OpenDoor(doorName);
        }
        status = TaskStatus.Success;
        return status;
    }
}
```

World Manager Class

```
public class WorldManager : MonoBehaviour {
    [SerializeField] private DoorInfo[] doors;
    [SerializeField] private WaypointInfo[] waypoints;

    public void OpenDoor(string doorName) {
        for (int x = 0; x < doors.Length; x++) {
            if (doors[x].name == doorName) {
                doors[x].transform.Translate(Vector3.right * 2f);
                doors[x].open = true;
                break;
            }
        }
    }

    public void CloseDoor(string doorName) {
        for (int x = 0; x < doors.Length; x++) {
            if (doors[x].name == doorName) {
                doors[x].transform.Translate(Vector3.left * 2f);
                doors[x].open = false;
                break;
            }
        }
    }
}
```

World Manager Class

...

```
public bool DoorIsOpen(string doorName){
    for (int x = 0; x < doors.Length; x++){
        if (doors[x].name == doorName){
            return doors[x].open;
        }
    }
    return false;
}

public Transform GetWaypoint(string name){
    foreach (WpInfo wp in waypoints){
        if (wp.name == name)
            return wp.transform;
    }
    return null;
}
}
```

Agent Class

```
public class Agent : MonoBehaviour
{
    [SerializeField] private WorldManager wordManager;
    private Task behaviorTree;
    private TaskStatus behaviorTreeStatus = TaskStatus.None;

    void Start() {
        Task sequenceMoveToRoom = new Sequence();
        sequenceMoveToRoom.AddChildren(new DoorOpenCondition("Door1"));
        sequenceMoveToRoom.AddChildren(new MoveAction("Room1"));

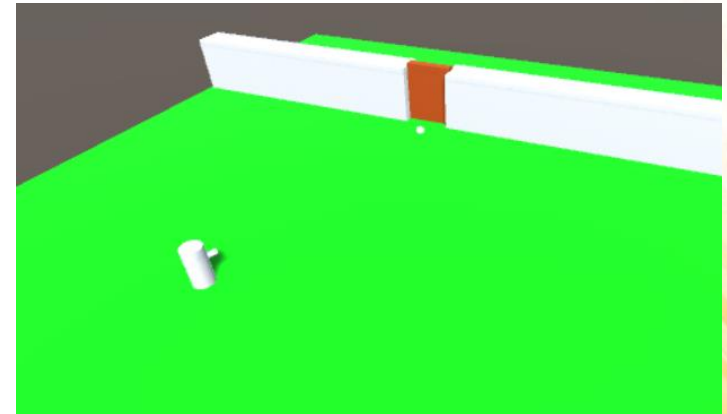
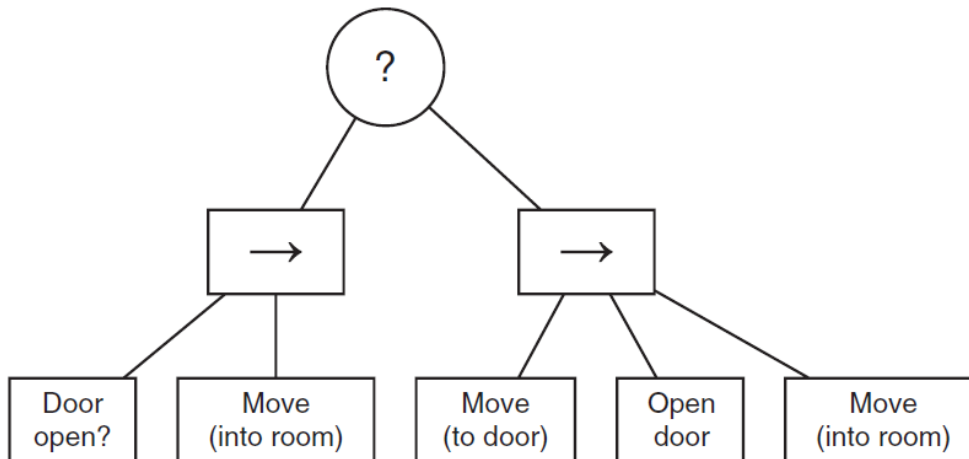
        Task sequenceOpenDoorMoveToRoom = new Sequence();
        sequenceOpenDoorMoveToRoom.AddChildren(new MoveAction("Door1"));
        sequenceOpenDoorMoveToRoom.AddChildren(new OpenDoorAction("Door1"));
        sequenceOpenDoorMoveToRoom.AddChildren(new MoveAction("Room1"));

        behaviorTree = new Selector();
        behaviorTree.AddChildren(sequenceMoveToRoom);
        behaviorTree.AddChildren(sequenceOpenDoorMoveToRoom);
    }
}
```

Agent Class

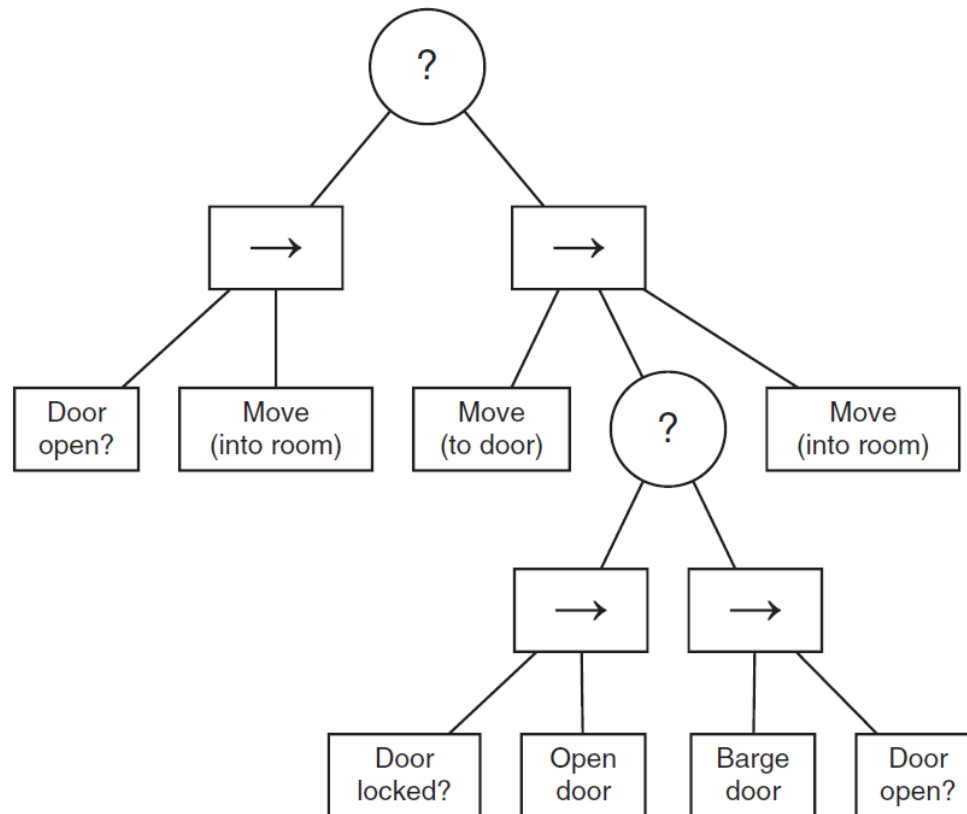
...

```
void Update(){  
    if ((behaviorTreeStatus == TaskStatus.None) ||  
        (behaviorTreeStatus == TaskStatus.Running)){  
        behaviorTreeStatus = behaviorTree.Run(this, wordManager);  
    }  
}
```



Exercise 1

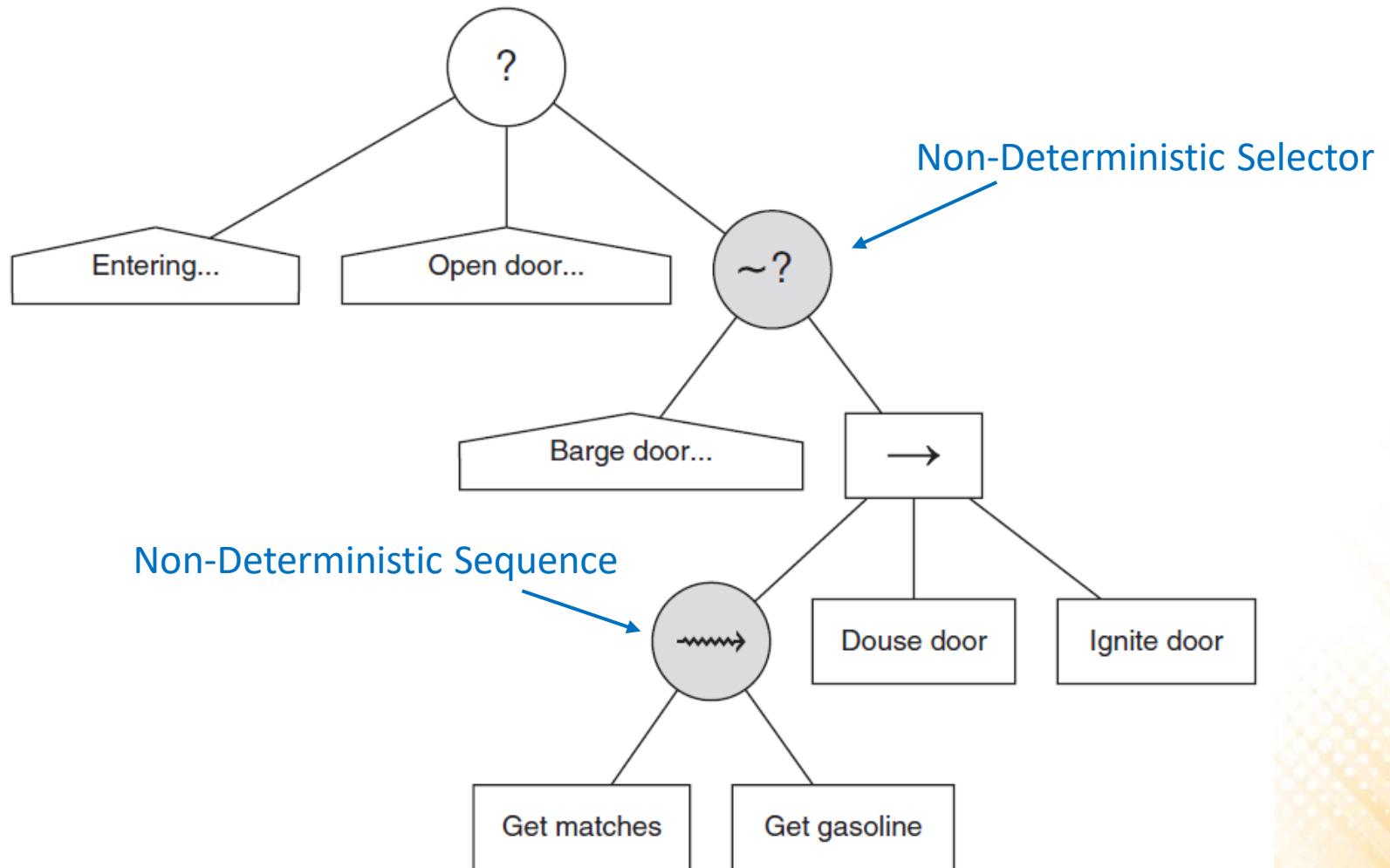
1) Implement and test the following behavior tree:



Non-Deterministic Composite Tasks

- Sometimes the order in which tasks are executed is extremely important. But there are some tasks that don't need to be executed in a particular order.
 - Executing tasks in same order can lead to predictable AI who always try the same things.
 - Example (sequence): get matches and gasoline to burn the door.
 - Example (selector): invade the room through the door or through the window.
- Non-deterministic composites can be implemented by shuffling the order of the children nodes before iterating through them.

Non-Deterministic Composite Tasks



Non-Deterministic Composite Tasks

- **NonDeterministicSequence Class:**

```
public class NonDeterministicSequence : Task {  
  
    private bool shuffledOrder;  
  
    public NonDeterministicSequence()  
    {  
        shuffledOrder = false;  
    }  
  
    public override TaskStatus Run(Agent agent,  
                                   WorldManager wordManager) {  
        if (!shuffledOrder) {  
            Shuffle(children);  
            shuffledOrder = true;  
        }  
        ...  
    }  
}
```

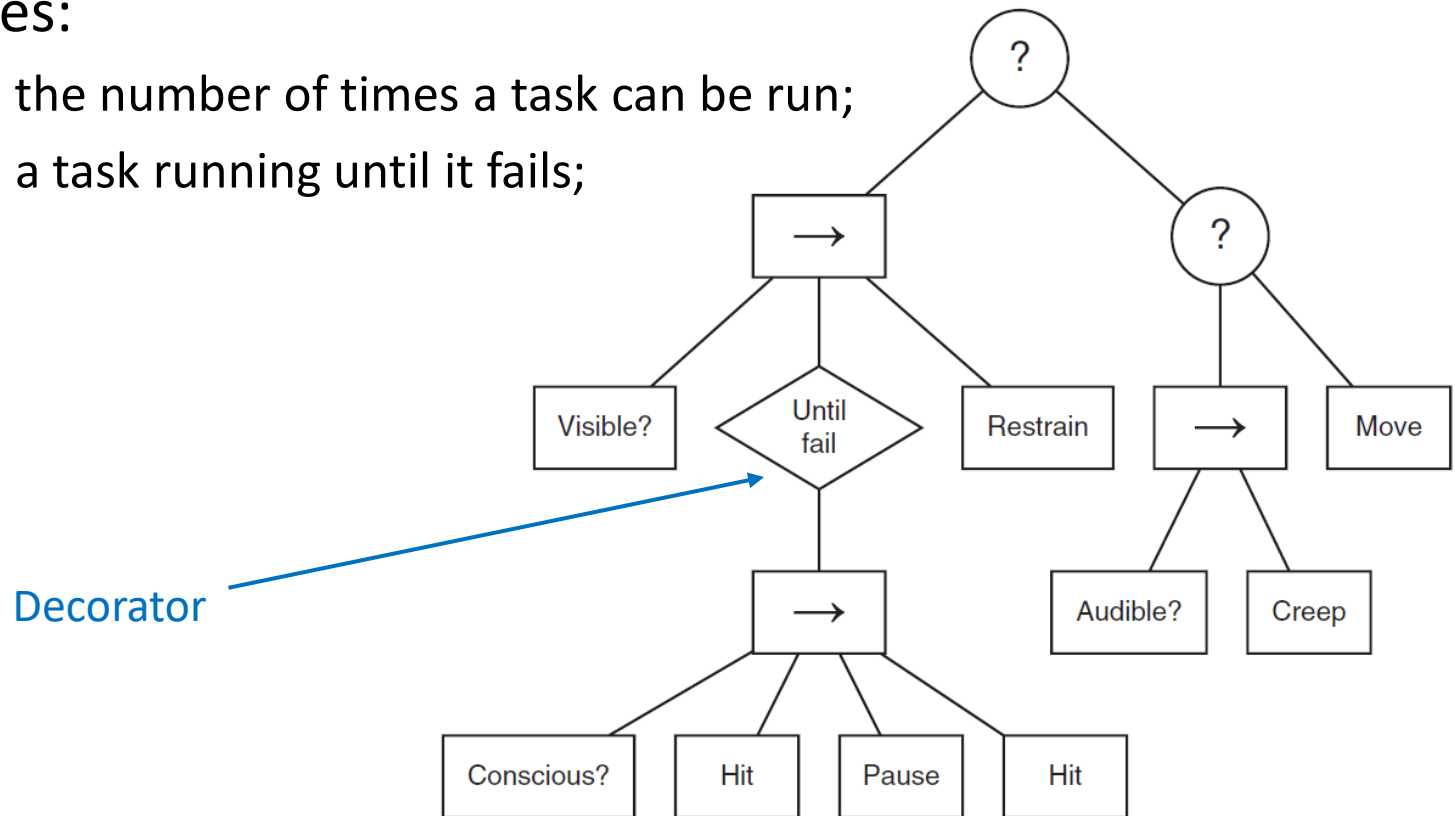
Non-Deterministic Composite Tasks

- **Simple Shuffle method:**

```
public void Shuffle(List<Task> list)
{
    int n = list.Count;
    while (n > 1)
    {
        int k = Random.Range(0, n);
        Task value = list[k];
        list[k] = list[n];
        list[n] = value;
        n--;
    }
}
```

Decorators

- Decorator is a type of task that has one single child task and modifies its behavior in some way.
- Examples:
 - Limit the number of times a task can be run;
 - Keep a task running until it fails;



Decorator Classes

- **Decorator Class:**

```
public abstract class Decorator : Task {  
    protected Task child;  
  
    new public void AddChildren(Task task)  
    {  
        child = task;  
    }  
}
```

- **UntilFailDecorator Class:**

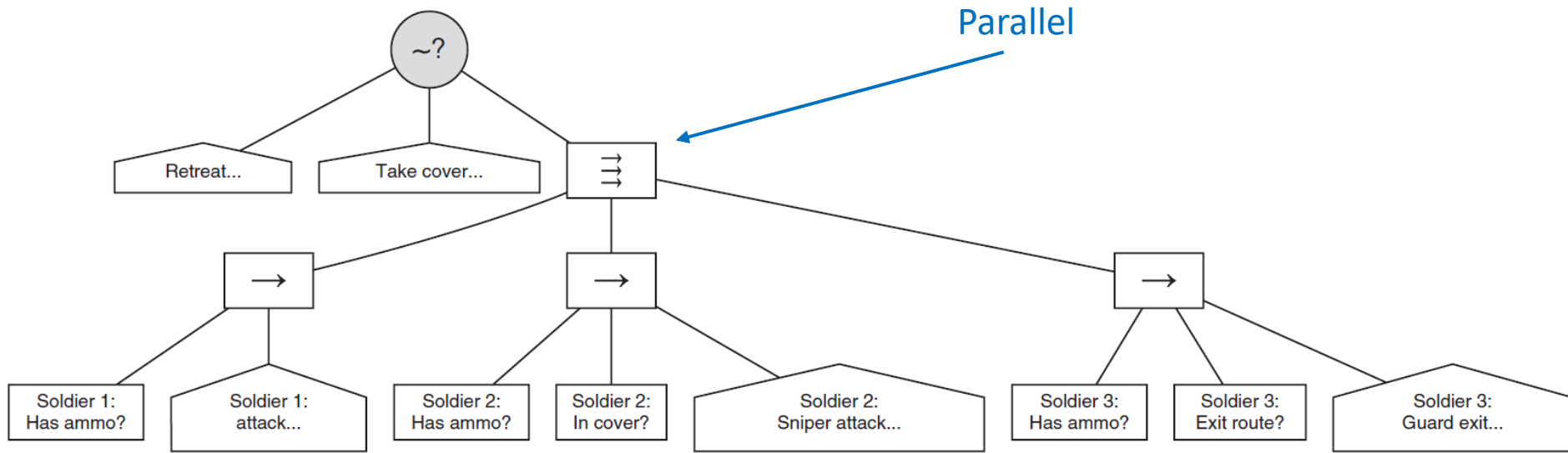
```
public class UntilFailDecorator : Decorator{  
    public override TaskStatus Run(Agent agent,  
                                   WorldManager wordManager){  
        if (status == TaskStatus.None)  
            status = TaskStatus.Running;  
        if (child.Run(agent, wordManager) == TaskStatus.Failure)  
            status = TaskStatus.Success;  
        return status;  
    }  
}
```

Parallel Tasks

- When parallel actions are necessary, we can add a third type of composite tasks to the behavior tree: Parallel.
- Rather than running all children tasks one at a time, it runs them all simultaneously.
 - Example: a character rolling into cover at the same time as shouting an insult and changing primary weapon.
- The Parallel task acts in a similar way to the Sequence task. It has a set of child tasks, and it runs them simultaneously until one of them fails.

Parallel Tasks

- At a higher level, we can also use Parallel tasks to control the behavior of a group of characters.

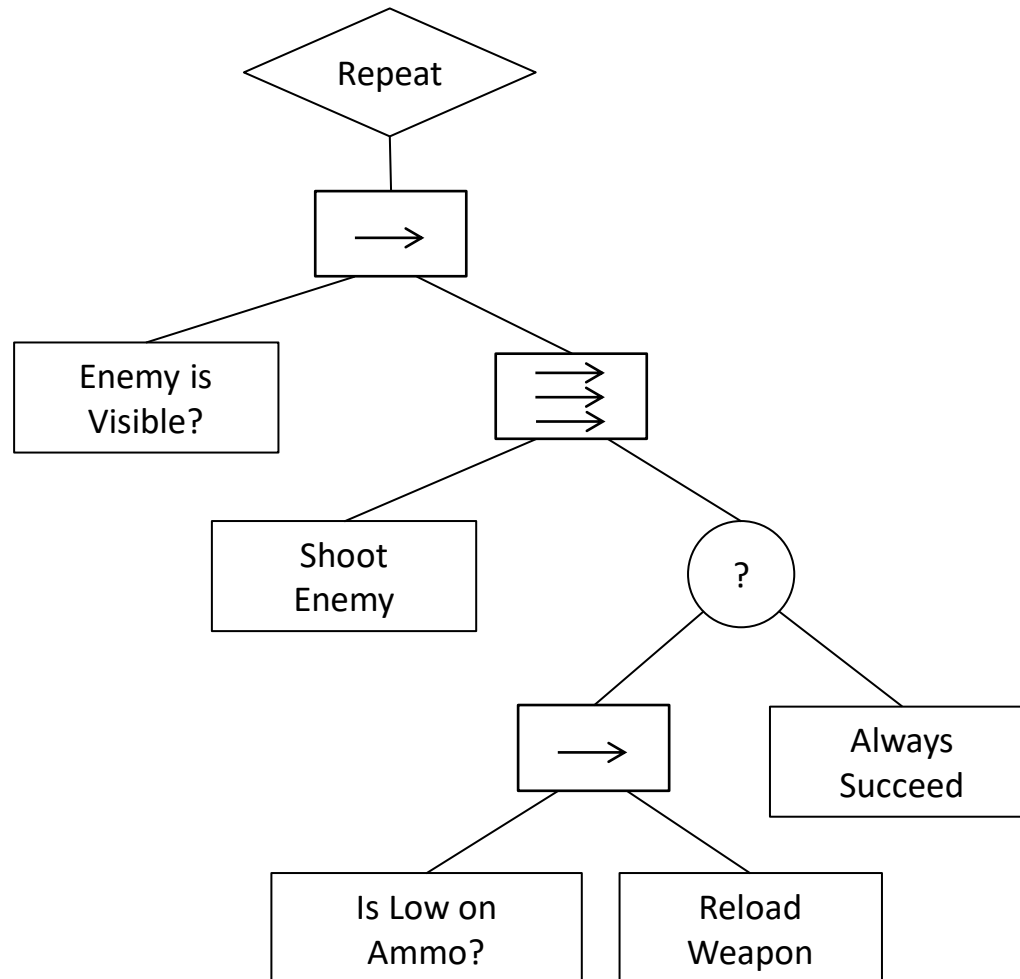


Parallel Class

```
public class Parallel : Task {
    public override TaskStatus Run(Agent agent, WorldManager wordManager){
        int successCount = 0;
        foreach (Task task in children){
            if (task.status != TaskStatus.Success){
                TaskStatus childrenStatus = task.Run(agent, wordManager);
                if (childrenStatus == TaskStatus.Failure){
                    status = TaskStatus.Failure;
                    return status;
                }
                else if (childrenStatus == TaskStatus.Success)
                    successCount++;
            }
            else
                successCount++;
        }
        if (successCount == children.Count)
            status = TaskStatus.Success;
        else
            status = TaskStatus.Running;
        return status;
    }
}
```

Exercise 2

2) Implement and test the following behavior tree:



Further Reading

- Millington, I., Funge, J. (2009). **Artificial Intelligence for Games** (2nd ed.). CRC Press. ISBN: 978-0123747310.
 - Chapter 5.4: Behavior Trees

