

Содержание

Введение.....	5
1 Обзор существующих инструментов	7
1.1 Общие сведения.....	7
1.2 Типы вершин	9
1.2.1 Лист	9
1.2.2 Композит	10
1.2.3 Декоратор.....	11
1.2.4 Взаимодействие различных вершин	11
1.3 Деревья поведения в современных фреймворках.....	12
1.3.1 Unreal Engine	12
1.3.2 LibGDX.....	14
1.3.3 Unity3D.....	17
1.4 Итоги	18
1.5 Постановка задачи.....	19
2 Библиотека Behavior Tree	20
2.1 Проектирование схемы классов.....	20
2.2 Описание реализованных классов и их методов	23
2.3 Пример использование библиотеки Behavior Tree	23
3 Визуальный редактор стратегий	33
3.1 Выбор инструментов и технологий.....	33
3.2 Проектирование схемы классов.....	33
3.3 Описание реализованных компонентов.....	34

3.4	Пример использования редактора	40
	Заключение	41
	Список использованных источников	42
	Приложение А Задание на выполнение бакалаврской работы	43
	Приложение Б Руководство пользователя.....	45
	Приложение В Исходный код программы	46
	Приложение Г Объемы рынка игр	47

Введение

Объем рынка игр растет с каждым годом (смотри рисунок Г.1 и Г.2). В первую очередь развитие игровых приложений обязано стремительному прогрессу, но также важная роль в разработке игр ложится на плечи программистов. Именно с их помощью реализуются идеи геймдизайнеров, именно они собирают воедино воображения художников с целью создания новой игры.

Программисту, в настоящее время, чтобы написать игру, что называется, «с нуля» необходимо позаботиться о многих важных аспектах: игровая платформа, графическая система, аудио система, системы моделирования игровой физики и виртуального интеллекта игры. Каждая из этих областей объемна и потребует много времени, чтобы связать воедино все системы в соответствии с игровой логикой.

Для того, чтобы упростить и ускорить работу над созданием игры программисты написали множество библиотек для работы с графическими системами и для моделирования игровой физики. Такие библиотеки, как Lightweight Java Game Library, предоставляют инструменты для работы с графической и аудио системами, а библиотека box2d предоставляет инструменты для моделирования реалистичной физики в плоском пространстве. Однако для упрощения реализации виртуального интеллекта долгое время не было придумано, по большому счету, ничего. Над созданием искусственного интеллекта в игре программист трудился самостоятельно, так как этот процесс не был сложным или утомительным. В конечном итоге модуль виртуального интеллекта содержал в себе некоторое количество условных операторов.

Такое решение обнажило бы свою слабую сторону в играх с развитым поведением персонажей потому, что условных операторов становилось очень много, что затрудняло отладку и дальнейшую поддержку игры. Пример такой игры – Spore. В ней все игровые персонажи имели если не уникальное, то редко повторяющееся поведение, для создания которого в Spore использовался совершенно другой подход.

Только в последнее десятилетие (примерно с момента выхода игры Spore) виртуальный интеллект в играх стал развиваться и появились некоторые библиотеки AIEngine (Artificial Intelligence Engine), которые обобщали накопленные программистами знания об интеллекте в играх. Эти библиотеки предоставляют инструменты для создания конечных автоматов поведения, для обработки взаимодействия автономных объектов. Но конечные автоматы в общем виде сложны и часто запутаны. Чтобы стратегия объекта была ясной были созданы деревья поведения – конечные автоматы древовидной структуры, состоящие из вершин трех типов: вершины-действия, вершины-условия и управляющие вершины. Вершина-действие содержит в себе некоторое возможное действие объекта (бежать, искать, стрелять), вершина-условие содержит в себе некоторый предикат (есть патроны?, враг рядом?), в зависимости от которого выбирается следующее состояние, управляющая вершина организует порядок обхода дочерних вершин (параллельно, до первой успешной вершины, до первой неуспешной вершины).¹

Цель данной работы - проектирование библиотеки и реализация визуального средства для создания стратегий поведения виртуальных игровых персонажей на основе деревьев поведения.

¹ Деревья поведения подробно описаны в главе 1.

1 Обзор существующих инструментов

1.1 Общие сведения

Для разработки виртуальных игровых объектов необходимо создать и поддерживать большой набор их поведений. Например, в военных играх виртуальному персонажу необходимо распознавать опасность и убегать в укрытие, когда уровень здоровья ниже 10%. От количества разнообразных вариантов действий, которые могут использовать игровые персонажи, зависит количество игровых ситуаций, которые могут быть распознаны и приняты во внимание. Чем больше различного поведения игроки будут встречать в играх даже от несущественных (фоновых) объектов², тем интереснее будет игра.

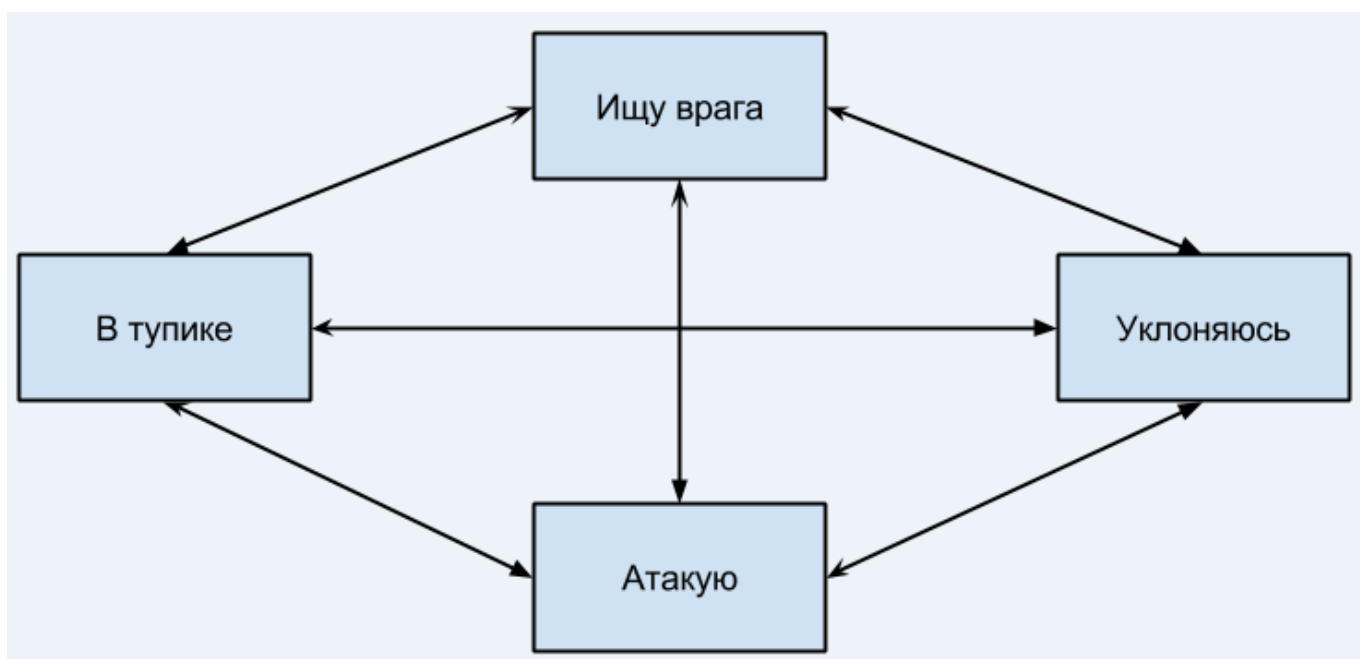


Рисунок 1.1 - Конечный автомат поведения игрового объекта в военных играх

Для создания поведения долгое время использовались конечные автоматы, где каждое поведение может быть представлено графически (рисунок 1.1). Одновременно автомат может находиться в одном состоянии, которое представляет

² Несущественные объекты – это игровые объекты, такие как массовка, животные, птицы, исключая ситуации, когда именно эти персонажи – главные герои в играх. Поведение несущественных объектов никак не влияет на ход игры.

поведение объекта. Каждое состояние имеет логику переходов с соответствующими проверками. Например, если игрок находится в состоянии «В тупике» и обнаружил врага, и здоровье игрока больше 50%, то следующим состоянием будет «Атакую».

Такой подход к реализации принятия решений игровыми объектами имеет ряд недостатков:

- **Расширяемость:** конечный автомат с большим количеством состояний теряет преимущество графического представления, со временем такое поведение станет невозможно понять.
- **Изменяемость:** при добавлении/удалении поведения (состояния) необходимо изменить все другие состояния, которые связаны с новым/старым поведением; большие изменения могут приводить к ошибкам логики поведения объекта в целом.
- **Параллельные алгоритмы:** запускать состояния автомата параллельно не представляется возможным.

В 2005 году был создан более эффективный способ принятия решения игровыми объектами по сравнению с конечными автоматами – деревья поведения.

Дерево поведения – это направленный связный ациклический³ граф, имеющий единственную вершину, в которую не входят ребра – корень дерева. Из пары вершин, соединенных ребром, та, из которой выходит ребро, называется родительской вершиной, а другая дочерней вершиной. Вершина, не имеющая дочерних, называется листом. Каждое поддерево дерева поведения определяет различное поведение. Вершины, находящиеся между корнем дерева и листьями могут быть двух типов – декораторами или композитами. Корень дерева поведения периодически генерирует сигнал, который передает дочерним вершинам, заставляя их выполнять алгоритм, определенный типом вершины. Как только сигнал достигнет листа, то лист

³ В данном случае дерево поведения не должно содержать циклов даже если не учитывать направления ребер.

произведет некоторые вычисления и вернет одно из 4 состояний: «успешно» (success), «не успешно» (failure), «запущено» (running), «ошибка» (error). Возвращенное состояние передается родительским вершинам, для принятия решений в соответствии с типом вершины. Процесс закончится тогда, когда корневая вершина вернет некоторое состояние.

1.2 Типы вершин

Все вершины дерева поведения делятся на три типа: декоратор, композит, лист.

1.2.1 Лист

Лист – это не делимая часть дерева поведения. Эти вершины не имеют дочерних вершин, они принимают сигнал, производят некоторые вычисления и возвращают результат родительской вершине.

Существует два вида листовых вершин: лист-условие и лист-действие. Лист-условие выполняет проверку некоторого условия и возвращает соответствующий результат (успешно, не успешно или ошибка). Лист-действие выполняет действие и возвращает результат успешно, запущено или ошибка.



Рисунок 1.2 - Графическое представление листовых вершин

Графически лист-условие изображается овалом, лист-действие прямоугольником (рисунок 1.2).

1.2.2 Композит

Композит имеет одну или больше дочерних вершин. Он принимает и передает сигнал дочерним вершинам в некотором порядке, и также решает какое и когда вернуть состояние. Композит всегда возвращает одно из трех состояний: «успешно», «не успешно» или «ошибка». Все композитные вершины изображаются в виде квадрата со специальным символом внутри.

Существует три вида композитной вершины (рисунок 1.3): композит-селектор, композит-последовательность и параллельный композит. Композит-селектор обрабатывает дочерние вершины до тех пор, пока дочерняя вершина возвращает результат «не успешно», затем пробрасывает полученный результат родительской вершине и заканчивает выполнение. Если все дочерние вершины вернули результат «не успешно», то композит-селектор вернет результат «не успешно». Специальный символ для композита-селектора – знак вопроса.

Композит-последовательность обрабатывает дочерние вершины до тех пор, пока они возвращают результат «успешно», затем пробрасывает полученный результат родительской вершине и заканчивает выполнение. Если все дочерние вершины вернули результат «успешно», то композит-последовательность вернет результат «успешно». Специальный символ для композита-последовательности – стрелка вправо.

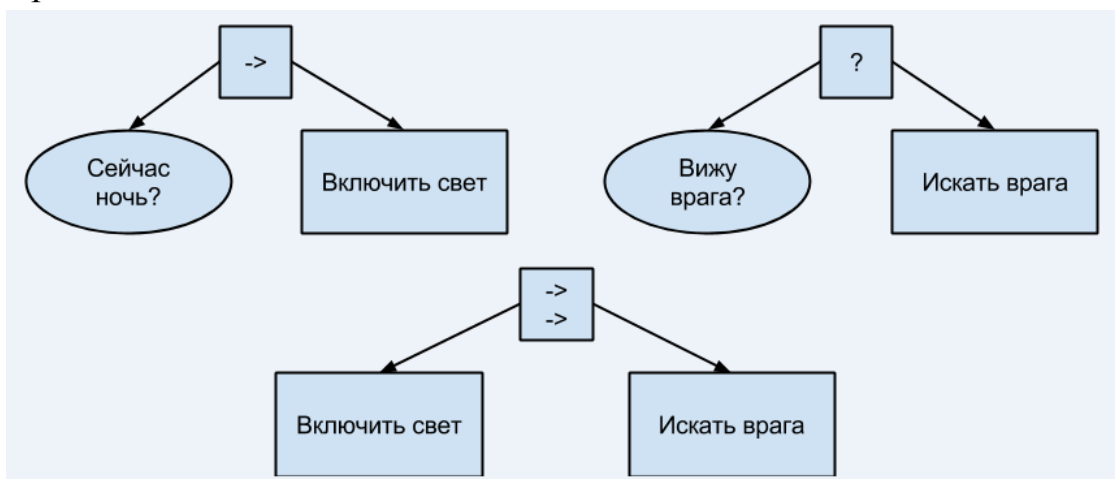


Рисунок 1.3 – Примеры вершин типа «композит»

Параллельный композит обрабатывает все вершины одновременно, возвращает

«успешно», если количество дочерних вершин с результатом «успешно» превышает некоторую константу S (которая может быть различна для разных параллельных композитов), возвращает результат «не успешно», если количество дочерних вершин с результатом «не успешно» превышает некоторую константу F , которая так же может быть определена для конкретного параллельного композита, иначе возвращает результат «запущено». Специальный символ для параллельного композита – две стрелки вправо.

1.2.3 Декоратор

Декоратор – это специальная вершина, которая имеет ровно одну дочернюю вершину. Цель, которую преследует декоратор, – изменить возвращаемое дочерней вершиной значение, или повлиять на частоту передаваемого сигнала дочерней вершины. Например, декоратор может делать инверсию возвращаемого значения, а может повторить сигнал, передаваемый дочерней вершине 3 раза. Декоратор изображается в виде ромба с пояснением внутри.

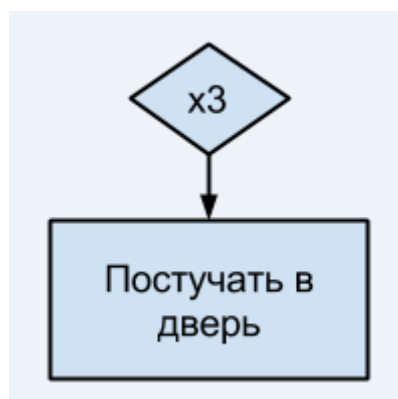


Рисунок 1.4 – Пример вершины-декоратора, который три раза передаст сигнал дочерней вершине

1.2.4 Взаимодействие различных вершин

Автономным объектам необходимо в процессе принятия решения хранить промежуточную информацию об окружающем мире. Эта информация формирует систему «восприятия» мира для объекта, она может включать, например, последнюю видимую позицию врага, количество видимых объектов, последнее совершенное

действие или любые другие вычисленные данные. Таким образом дерево поведения некоторого игрового объекта должно хранить и использовать некоторую информацию о мире.

Для решения данной задачи применяется blackboard [3] (рисунок 1.5), который активно используется вершинами дерева поведения для чтения и записи информации. Blackboard – это ассоциативный массив, к которому имеют доступ все вершины дерева поведения.



Рисунок 1.5 - Использование пула памяти

1.3 Деревья поведения в современных фреймворках

В некоторых достаточно крупных игровых фреймворках существует возможность создания стратегии персонажей с помощью деревьев поведения. Мы рассмотрим три фреймворка, использующих эту технологию: Unreal Engine, LibGDX, Unity3D.

1.3.1 Unreal Engine

UnrealEngine – игровой движок, разрабатываемый компанией Epic Games. Различные версии этого фреймворка были использованы во многих современных играх.

Для реализации стратегии для автономных объектов в UnrealEngine можно использовать деревья поведения [4], причем в данном игровом движке есть возможность создать дерево поведения, используя графический интерфейс, при этом не написав ни строчки кода.

Рассмотрим использование деревьев поведения в UnrealEngine на примере:

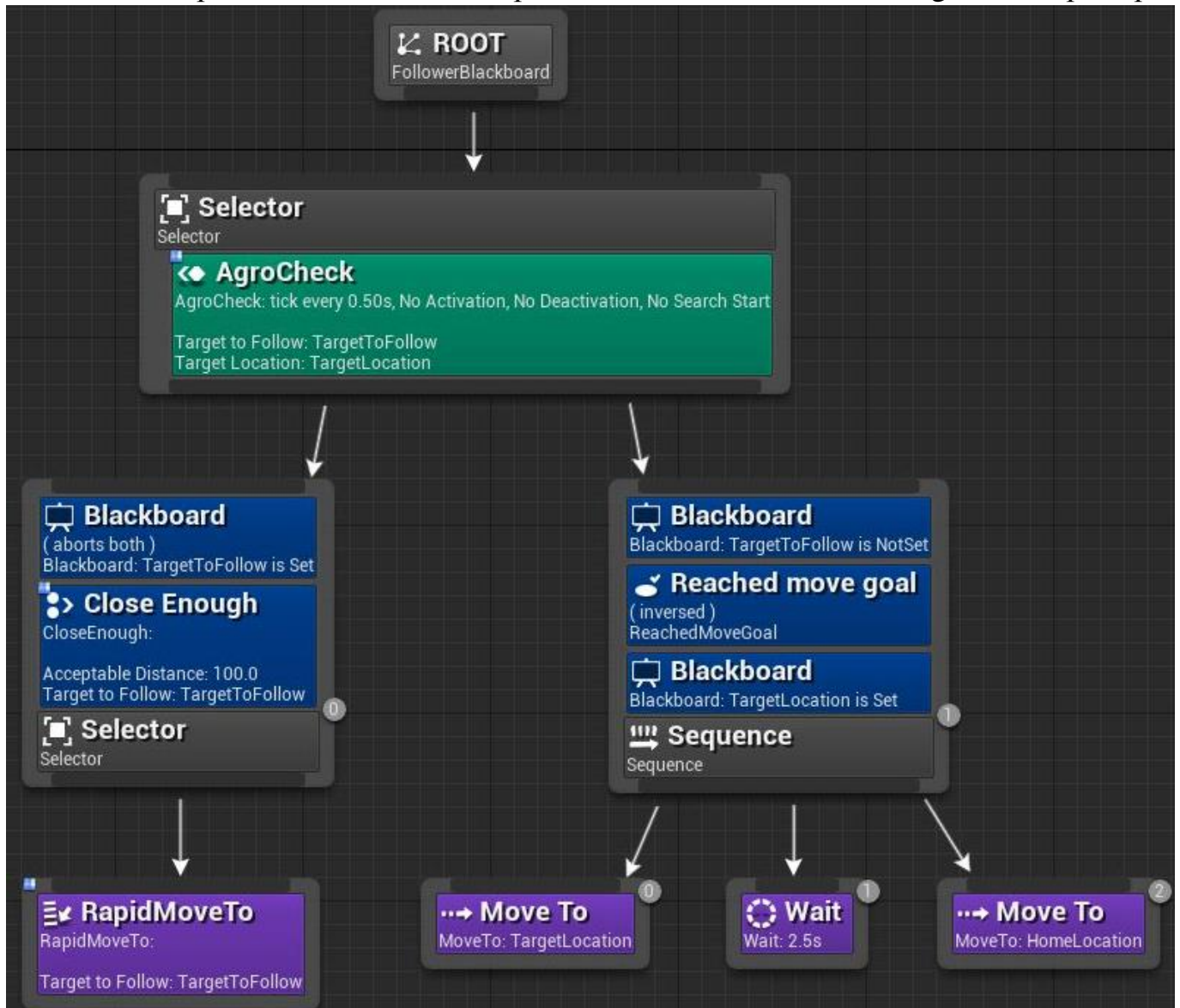


Рисунок 1.6 - Реализация дерева поведения в UnrealEngine

имеется игровой мир в виде комнаты со стенами, в котором находятся два персонажа, один из них – человек, другой – виртуальный интеллект, задача виртуального интеллекта – найти человека и подойти к нему. Для реализации такого рода стратегии в UnrealEngine необходимо построить дерево поведения (смотри рисунок 1.6).

Данное дерево состоит из двух вершин-селекторов, вершины-последовательности, четырех вершин-действий. В первой вершине-селектор находится дополнительное действие «AgroCheck», которое реализует «зрение» виртуального интеллекта, устанавливая, в частности, две переменные: позиция и ссылка на персонажа, которого необходимо отыскать. Композит-селектор слева включает в себя два декоратора, которые проверяют установлена ли ссылка на персонажа и приемлема ли до него дистанция, при верном ответе на эти два вопроса выполняется действие – быстрое движение к искомому объекту. Если же объект пропадет из поля видимости, то тогда сигнал пойдет к левому композиту-последовательности, который содержит три декоратора, проверяющих, что цель не установлена, цель не достигнута и некоторая позиция установлена, в этом случае будут выполняться следующие действия. Сначала виртуальный интеллект переместится до установленной позиции (скорее всего это та позиция, где он последний раз видел другого персонажа), затем подождет 2.5 секунды и отправится на исходную точку.

1.3.2 LibGDX

LibGDX – кроссплатформенный игровой фреймворк, написанный на языке Java и C++. LibGDX используют для написания мобильных приложений и игр. В своем составе этот фреймворк имеет модуль AI [5], который реализует алгоритмы нахождения кратчайших путей, взаимодействия автономных объектов (steering behavior), а также алгоритмы принятия решений на основе behavior tree.

Рассмотрим создание дерева поведения на том же примере, что и в предыдущем разделе. Так как в деревьях поведения LibGDX AI нет декораторов, проверяющих некоторое условие, то аналогичное дерево поведения будет выглядеть как показано на рисунке 1.7.

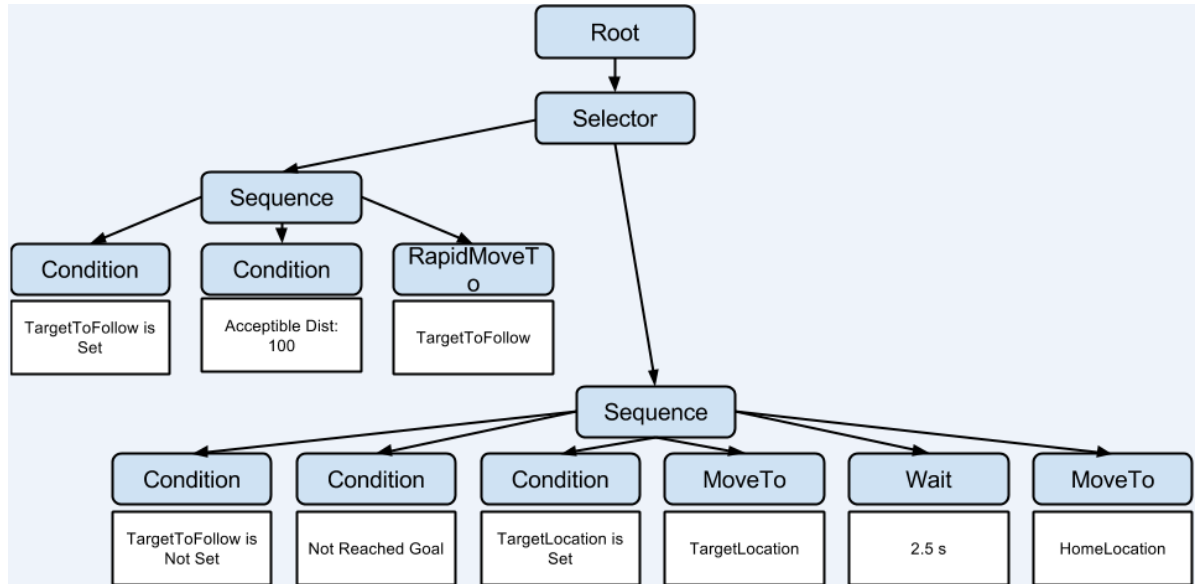


Рисунок 1.7 - Дерево поведения с использованием модуля LibGDX AI

Так как в LibGDX AI не предусмотрено графического средства для создания деревьев поведения, то полученное на рисунке 1.7 дерево необходимо создать непосредственно в коде.

```

private BehaviorTree<Blackboard> createHeroBehavior() {
    ConditionTask<Blackboard> targetToFollowIsSet = new ConditionTask<>() {
        bb -> bb.targetToFollow != null);
    ConditionTask<Blackboard> acceptableDist = new ConditionTask<>() {
        bb -> dist(bb.targetToFollow.location, me.location) <= 100);
    RapidMoveTo<Blackboard> rapidMoveTo =
        new RapidMoveTo<>(bb.targetToFollow.location);
    Sequence<Blackboard> sequenceSeeMan = new Sequence<>() {
        targetToFollowIsSet, acceptableDist, rapidMoveTo);

    ConditionTask<Blackboard> targetToFollowIsNotSet = new ConditionTask<>() {
        bb -> bb.targetToFollow == null);
    ConditionTask<Blackboard> notReachedGoal = new ConditionTask<>() {
        bb -> dist(bb.targetLocation, me.location) > 1);
    ConditionTask<Blackboard> targetLocationIsSet = new ConditionTask<>() {
        bb -> bb.targetLocation != null);
  
```

```

        MoveTo<Blackboard> moveToTargetLocation = new MoveTo<>(bb.targetLocation);
        Wait<Blackboard> wait = new Wait<>(2500);
        MoveTo<Blackboard> moveToHomeLocation = new MoveTo<>(bb.homeLocation);
        Sequence<Blackboard> sequenceNotSeeMan = new Sequence<>(
            targetToFollowIsNotSet,
            notreachedGoal,
            targetLocationIsSet,
            moveToTargetLocation,
            wait,
            moveToHomeLocation
        );

        Selector<Blackboard> selector = new Selector<>(sequenceSeeMan, sequenceNotSeeMan);
        Selector<Blackboard> root = new Selector<Blackboard>(selector);
        BehaviorTree<Blackboard> bt = new BehaviorTree<>(root, blackboard);
        return bt;
    }

```

В LibGDX для каждой вершины дерева поведения есть общий класс – **Task**. Наследуясь от него можно создавать пользовательские типы вершин, которые не предусмотрены в базовой структуре классов. Так, лист-условие **Condition**, листы-действия **RapidMoveTo**, **MoveTo** и **Wait** необходимо создать и реализовать их логику. Так как композит-последовательность передает сигнал на исполнение дочерним вершинами до тех пор, пока они возвращают «успешно», то данное дерево будет аналогичным дереву в разделе 1.3.1. Действительно, если некоторый лист-действие вернет «не успешно», то композит-селектор прекратит передачу сигнала на исполнение и следующим дочерним вершинам сигнал не будет передан.

В методе **createHeroBehavior** описан процесс создания дерева поведения с использованием LibGDX AI. Лист-условие **Condition** задается предикатом с одним параметром – **blackboard**, с помощью которого и определяется собственно истинность условия. Листы-действия **RapidMoveTo** и **MoveTo** задаются с помощью функции, которая возвращает позицию, к которой необходимо двигаться. Лист-действие **Wait** задается длительностью интервала задержки в миллисекундах.

1.3.3 Unity3D

Unity3D – игровой движок, разрабатываемый компанией Unity Technologies. В силу наличия бесплатной версии и огромного количества поддерживаемых платформ [7] этот движок весьма популярен среди многих крупных разработчиков игр (Blizzard, EA, QuartSoft, Ubisoft).

В составе Unity3D отсутствуют инструменты для работы с искусственным интеллектом, но в магазине плагинов [8] можно найти дополнение, позволяющее создавать деревья поведения для игровых объектов. Один из таких плагинов – Behaviour Machine Free. Рассмотрим создание дерева поведения с использованием данного плагина на примере из предыдущего раздела.

В Behaviour Machine Free много стандартных вершин, таких как Translate (переместить объект по указанному направлению на указанную длину), IsFloatLess/IsFloatGreater (проверить, что некоторое число с плавающей точкой меньше/больше заданного числа), GetDistance (получить расстояние между двумя объектами и записать результат в blackboard), IsSee (проверить, видит один объект другой или нет).

Таким образом, для того чтобы создать необходимое поведение для персонажа, нужно в графическом интерфейсе для определения способа принятия решений создать структуру дерева поведения и структуру blackboard (рисунок 1.8), а затем задать параметры для каждой вершины дерева. Так, для **Agro Check** необходимо задать объекты, между которыми нужно проверять видимость, для **Acceptible Dist** необходимо задать переменную и число, чтобы проверять, что переменная меньше числа, для **Wait** необходимо задать количество миллисекунд задержки и т.д.

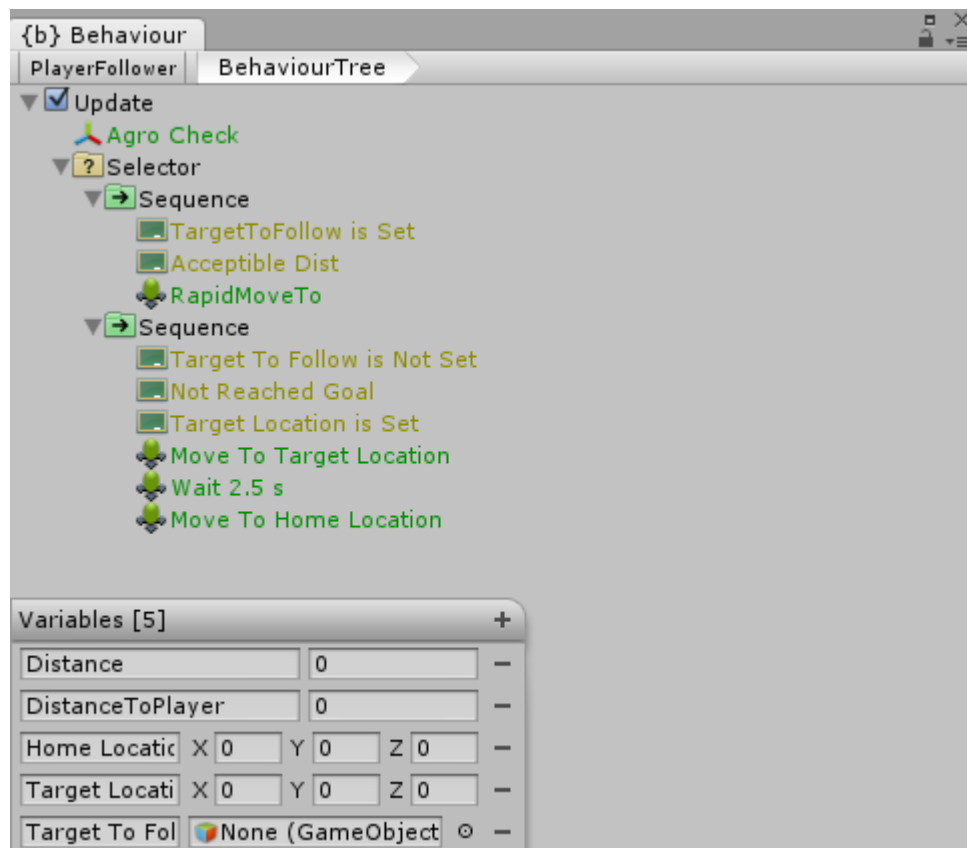


Рисунок 1.8 - Дерево поведения с использованием Behaviour Machines на Unity3D

1.4 Итоги

Таким образом, мы рассмотрели три крупных игровых фреймворка, в которых есть возможность использовать деревья поведения для создания логики принятия решений виртуальным интеллектом, и решили одну задачу с помощью этих фреймворков. Были отмечены следующие недостатки: использовать деревья

поведения UnrealEngine или Behaviour Machine Free в проектах, написанных на других фреймворках не представляется возможным, а для модуля Behavior Tree из LibGDX AI не существует визуального средства создания деревьев поведения.

1.5 Постановка задачи

Необходимо разработать кроссплатформенную библиотеку и визуальное средство для создания деревьев поведения, которые не будут зависимы от конкретного игрового фреймворка. Библиотека должна обладать стандартизированным и расширяемым программным интерфейсом. Также необходимо разработать демонстрационный пример, созданный при помощи визуального средства проектирования деревьев поведения, показывающий основные возможности библиотеки.

2 Библиотека Behavior Tree

2.1 Обоснование выбора языка программирования

Для реализации библиотеки Behavior Tree мы выбрали язык Java 1.8 по нескольким причинам:

- Java – кроссплатформенный язык программирования
- Наличие анонимных методов
- Низкий порог вхождения

2.2 Проектирование схемы классов

На рисунке 2.1 частично представлена разработанная структура классов библиотеки Behavior Tree.

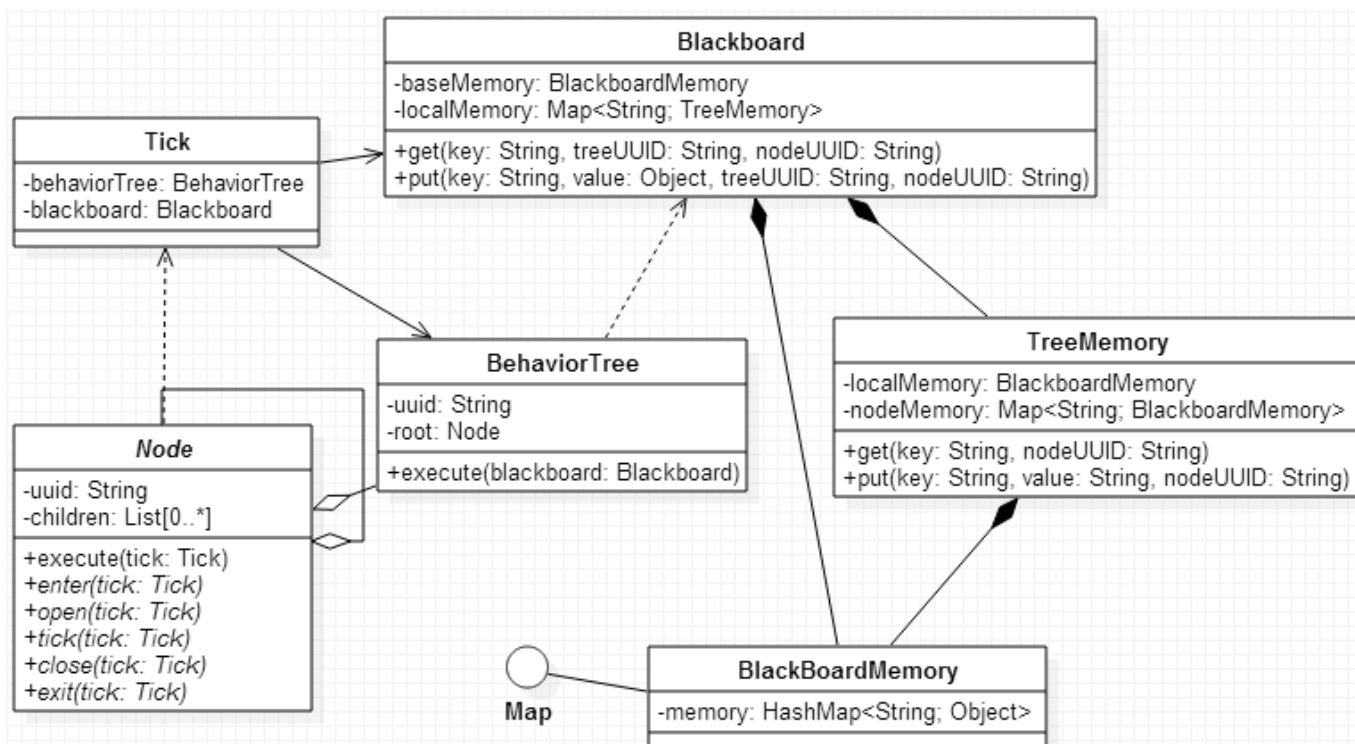


Рисунок 2.1 - Схема классов библиотеки Behavior Tree

Рассмотрим подробно структуру Blackboard. Для эффективного хранения/чтения/записи переменных мы разбили «память» на несколько областей: глобальная область, область дерева, область вершины в дереве.

Глобальная область содержит переменные, логически не связанные ни с одним поведением (baseMemory). Такие переменные могут отражать некоторую общую

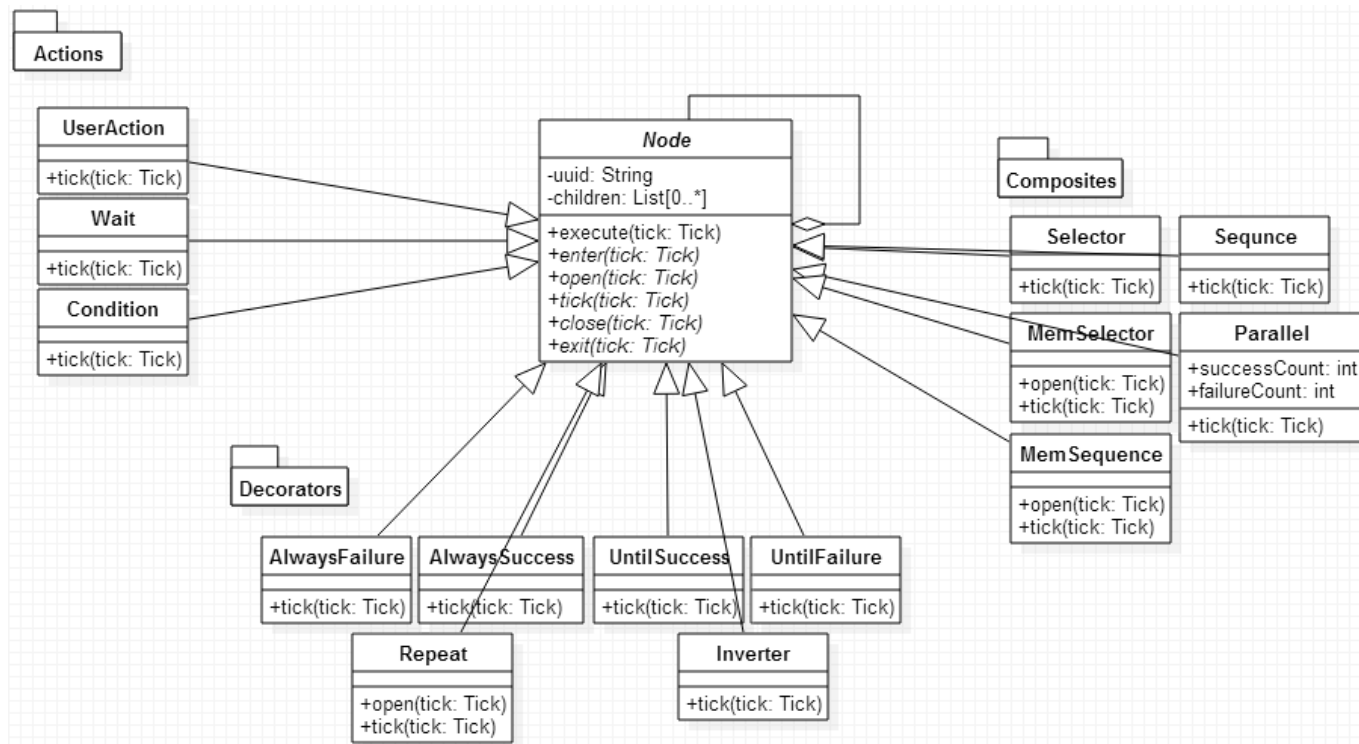
характеристику персонажа, такую, как количество забитых мячей в матче или текущее состояние здоровья.

Область дерева (TreeMemory) содержит переменные, логически связанные с одним поведением. Такие переменные могут хранить информацию о поведении, например, «защита»: количество видимых врагов, место ближайшего укрытия и количество дееспособных объектов в своем отряде. К области памяти дерева поведения можно получить доступ при указании уникального идентификатора дерева `treeUUID`.

Область вершины в дереве содержит локальную информацию, характерную только для конкретной вершины в конкретном поведении. Мы используем эту область для хранения информации о последней запущенной вершине в композитах с запоминанием. Пользователю рекомендуется здесь хранить ключевую информацию, используемую в пользовательских листах-действие. К области вершины в дереве можно получить доступ при указании уникальных идентификаторов дерева (`treeUUID`) и вершины (`nodeUUID`) в нем.

Каждая вершина и каждое дерево поведения имеют уникальные идентификаторы для определения участка памяти для хранения переменных в Blackboard. Эти уникальные идентификаторы генерируются по стандарту идентификации UUID [10] при создании объекта.

Для того, чтобы каждая вершина имела доступ к набору переменных и дереву поведения мы используем объект класса `Tick`, который как сигнал на исполнение передается дочерним вершинам. Объект класса `Tick` создается один раз на принятие одного решения, то есть этот объект создается в методе `execute` класса `BehaviorTree`



и передается корневой вершине дерева поведения.

Рисунок 2.2 - Различные классы вершин

Все вершины, представленные на рисунке 2.2, наследуются от класса `Node` и реализуют некоторые его абстрактные функции, необходимые для обеспечения логики вершины.

Так сигнал на исполнение запускает метод `tick`, которая реализует ядро логики вершины. Например, вершина-действие `Wait` в этом методе проверяет, прошло ли достаточное время с момента первого запуска. Абстрактный метод `enter` класса `Node` запускается всякий раз, когда сигнал на исполнение пришел в вершину. Метод `open` запускается только в том случае, если после последнего запуска данной вершины она вернула результат не «запущено». Метод `close` запускается в том случае, если текущее результат не «запущено». Таким образом, в один момент принятия решения могут быть ситуации, когда метод `open` был запущен, а метод `close` – нет, или наоборот.

Метод `exit` выполняется всякий раз перед возвратом результата исполнения логики вершины.

Выше описанная логика реализована в методе `execute` класса `Node`. Именно этот метод запускается, когда необходимо передать сигнал на исполнение вершине.

2.3 Описание реализованных классов и их методов

Описание реализованных классов и методов выполнено в виде Doxygen [9] комментариев и приведено в приложении Д. Мы выбрали Doxygen комментарии, так как они являются стандартом де-факто комментирования кода на различных языках.

2.4 Разработка примера использования библиотеки Behavior Tree

Для демонстрации примера использования разработанной библиотеки решим задачу из раздела 1.3.1. При решении задачи будем использовать игровой фреймворк LibGDX [5] для подсистемы ввода/вывода, так как он имеет простой в использовании интерфейс, и физический движок `box2d` [11] для обработки взаимодействия объектов.

Виртуальный персонаж (далее компьютер) и персонаж, управляемый человеком (далее человек) будут иметь общий метод – двигаться к некоторой точке с некоторой скоростью, поэтому создадим общий для них класс `Man`:

```
class Man extends Circle {
    Body physicBody;

    public Man(Body physicBody, float x, float y) {
        this.physicBody = physicBody;
        this.setPosition(x, y);
        this.setRadius(manRadius);
    }
    @Override
    public void setPosition(float x, float y) {
        super.setPosition(x, y);
        this.physicBody.setTransform(x, y, 0);
    }
    public void moveTo(Vector2 location, float velocity) {
        float manAng = physicBody.getAngle();
```

```

float manToLocAng = (float) Math.atan2(
    location.y - physicsBody.getPosition().y,
    location.x - physicsBody.getPosition().x
);
if (Math.abs(manAng - manToLocAng) > 0.1) {
    physicsBody.setLinearVelocity(Vector2.Zero);
    physicsBody.setAngularVelocity(manAng > manToLocAng ? -5 : 5);
} else {
    physicsBody.setLinearVelocity(
        new Vector2(
            location.x - physicsBody.getPosition().x,
            location.y - physicsBody.getPosition().y
        ).nor().scl(velocity)
    );
    physicsBody.setAngularVelocity(0);
}
}
public void stop() {
    physicsBody.setAngularVelocity(0);
    physicsBody.setLinearVelocity(Vector2.Zero);
}
}

```

Метод `moveTo(Vector2 location, float velocity)` сначала разворачивает персонажа по направлению к точке `location`, а затем двигает вдоль этого направления со скоростью `velocity`. Метод `stop()` устанавливает линейную и угловую скорость значением ноль.

Класс для человека не будет иметь каких-либо дополнительных методов, так как его задача — двигаться туда, куда укажет пользователь:

```

class Human extends Man {
    public Human(Body physicsBody, float x, float y) {
        super(physicsBody, x, y);
    }
}

```

Класс для компьютера должен принимать решения о том, куда двигаться дальше. Поэтому в его конструкторе создадим дерево поведения с необходимой логикой и добавим метод принятия решения:

```

class Computer extends Man {
    BehaviorTree behaviorTree;
    public Computer(Body physicsBody, float x, float y) {
        super(physicsBody, x, y);
        behaviorTree = new BehaviorTree(
            new Selector(
                new AlwaysFailure(new UserAction(tick -> {
                    boolean seeHuman = isComputerSeeHuman();
                    tick.getBlackboard()
                        .put("targetToFollow", seeHuman ? human : null);
                    if (seeHuman) {
                        Vector2 hPos = human.physicsBody.getPosition();
                        tick.getBlackboard()
                            .put("targetLocation", new Vector2(hPos.x, hPos.y));
                    }
                    tick.getBlackboard().put("computerLocation", null);
                })),
            new Sequence(
                new Condition(
                    tick -> tick.getBlackboard().get("targetToFollow") != null
                ),
                new UserAction(tick -> {
                    Human human =
                        (Human) tick.getBlackboard().get("targetToFollow");
                    tick.getBlackboard()
                        .put("computerLocation", human.physicsBody.getPosition());
                })
            ),
            new Sequence(
                new Condition(tick ->
                    tick.getBlackboard().get("targetToFollow") == null),
                new Condition(tick ->
                    tick.getBlackboard().get("targetLocation") != null),
                new UserAction(tick -> {
                    Vector2 position = (Vector2)
                        tick.getBlackboard().get("targetLocation");
                    tick.getBlackboard().put("computerLocation", position);
                })
            )
        )
    }
}

```

```

    )
    );
}
public void makeDecision(Blackboard blackboard) {
    behaviorTree.execute(blackboard);
}
}

```

Дерево поведения, реализующее поведение компьютера, представлено на рисунке 2.3 (рисунок получен с использованием разработанной программы для построения деревьев поведения).

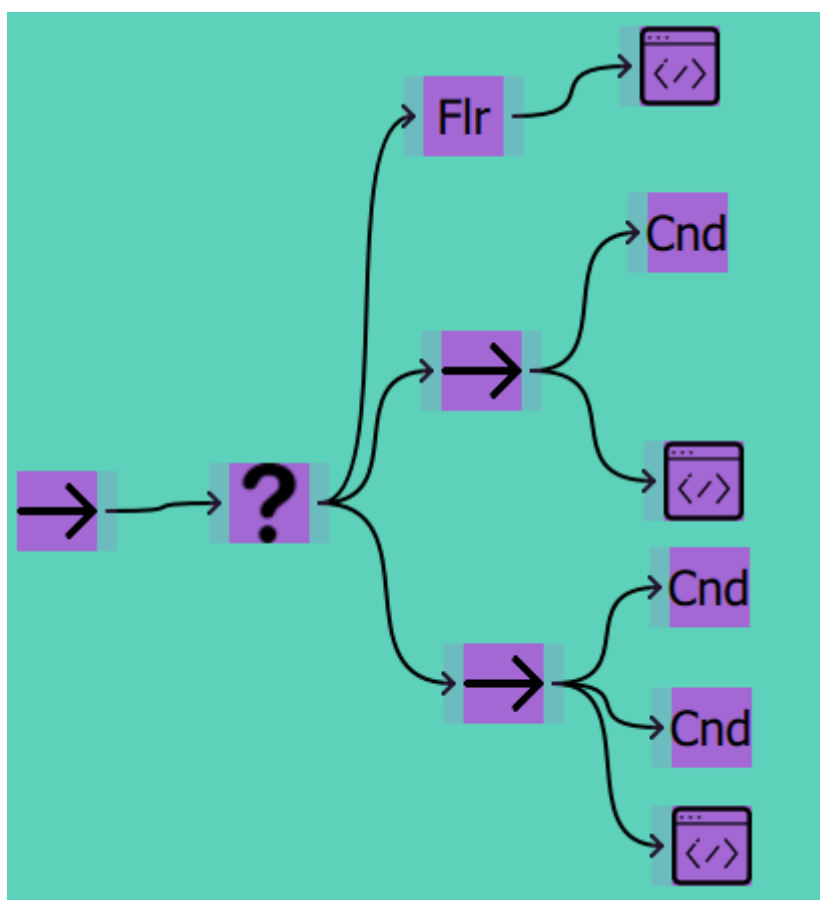


Рисунок 2.3 - Дерево поведения компьютера

Корень дерева поведения – композит-селектор, поэтому, если дочерняя вершина корня возвращает результат «не успешно», то сигнал передается следующей вершине. Первая дочерняя вершина у корня – это лист-действие, обернутый в декоратор, который всегда возвращает «не успешно», поэтому действия, описанные в этом листе будут выполняться каждый раз при принятии решения. Данный лист

действие устанавливает переменные `targetToFollow` – объект класса `Human`, если компьютер его видит (нет никаких препятствий на отрезке, соединяющем центры кругов объектов человека и компьютера соответственно), `targetToLocation` – последнее место, где был виден человек, `computerLocation` – точка, куда будет двигаться компьютер после принятия решения.

Следующая дочерняя вершина – композит-последовательность, поддерево с корнем в этой вершине определяет поведение в случае, если компьютер видит человека (первое условие), в этом случае вторая дочерняя вершина установит `computerLocation` точкой, где в данный момент находится человек.

Следующая дочерняя вершина – тоже композит последовательность, и определяет поведение компьютера в случае, когда он не видит человека. В этом случае `computerLocation` будет указывать на точку, где последний раз был виден человек.

Чтобы проверить видит ли компьютер человека или нет, необходимо провести отрезок между центрами кругов определяющих объекты компьютера и человека и проверить, не пересекает ли этот отрезок какое-нибудь препятствие. В нашем случае все препятствия – это прямоугольники. Функция, проверяющая видит, ли компьютер человека:

```
boolean isComputerSeeHuman() {
    Vector2 a = computer.physicBody.getPosition();
    Vector2 b = human.physicBody.getPosition();
    boolean isIntersect = false;
    for (Wall wall : walls) {
        float w = wall.getWidth();
        float h = wall.getHeight();
        float wallX = wall.getX();
        float wallY = wall.getY();
        isIntersect |=
            Intersector.intersectSegments(
                a.x, a.y, b.x, b.y,
                wallX - w / 2, wallY + h / 2, wallX + w / 2, wallY + h / 2,
                new Vector2()
            )
    }
}
```

```

        || Intersector.intersectSegments(
        a.x, a.y, b.x, b.y,
        wallX - w / 2, wallY - h / 2, wallX + w / 2, wallY - h / 2,
        new Vector2()
    )

        || Intersector.intersectSegments(
        a.x, a.y, b.x, b.y,
        wallX - w / 2, wallY + h / 2, wallX - w / 2, wallY - h / 2,
        new Vector2()
    )

        || Intersector.intersectSegments(
        a.x, a.y, b.x, b.y,
        wallX + w / 2, wallY + h / 2, wallX + w / 2, wallY - h / 2,
        new Vector2()
    );
    };
}
return !isIntersect;
}

```

Множество стен определяется классом `Wall`, который представляет прямоугольник:

```

class Wall extends Rectangle {
    Body physicBody;
    public Wall(Body physicBody, float x, float y) {
        this.physicBody = physicBody;
        this.setSize(wallWidth, wallHeigh);
        this.setPosition(x, y);
    }
    @Override
    public Rectangle setPosition(float x, float y) {
        Rectangle rect = super.setPosition(x, y);
        this.physicBody.setTransform(x, y, 0);
        return rect;
    }
}

```

Теперь необходимо создать все объекты в переопределенном методе класса `LibGDX ApplicationAdapter.create()`:

```

@Override
public void create() {

```

```

int w = Gdx.graphics.getWidth();
int h = Gdx.graphics.getHeight();
batch = new SpriteBatch();
debugRenderer = new Box2DDebugRenderer();
camera = new OrthographicCamera(w, h);
camera.translate(0, 0);
world = new World(new Vector2(0, 0), true);
// create Human
BodyDef def = new BodyDef();
def.type = BodyDef.BodyType.DynamicBody;
Body circle = world.createBody(def);
CircleShape circleShape = new CircleShape();
circleShape.setRadius(0.1f);
circle.createFixture(circleShape, 0.1f);
human = new Human(circle, 0, 0);
// create Computer
circle = world.createBody(def);
def.type = BodyDef.BodyType.DynamicBody;
circleShape = new CircleShape();
circleShape.setRadius(0.1f);
circle.createFixture(circleShape, 0.1f);
computer = new Computer(circle, 1.5f, 1);
blackboard = new Blackboard();
// create Walls
def.type = BodyDef.BodyType.StaticBody;
PolygonShape polygonShape = new PolygonShape();
polygonShape.setAsBox(0.1f, 1f);
Body rectangle = world.createBody(def);
rectangle.createFixture(polygonShape, 0.1f);
walls.add(new Wall(rectangle, -1, 0));
rectangle = world.createBody(def);
rectangle.createFixture(polygonShape, 0.1f);
walls.add(new Wall(rectangle, 1, 0));
// set input listener
Gdx.input.setInputProcessor(new InputProcessor() {
    @Override
    public boolean keyDown(int keycode) {
        return false;
    }
}

```

```

@Override
public boolean keyUp(int keycode) {
    return false;
}
@Override
public boolean keyTyped(char character) {
    return false;
}
@Override
public boolean touchDown(int screenX, int screenY, int pointer, int button) {
    int w = Gdx.graphics.getWidth();
    int h = Gdx.graphics.getHeight();
    humanLocation = new Vector2(
        (screenX - w / 2) / 100.0f, (h / 2 - screenY) / 100.0f);
    return false;
}
@Override
public boolean touchUp(int screenX, int screenY, int pointer, int button) {
    return false;
}
@Override
public boolean touchDragged(int screenX, int screenY, int pointer) {
    return false;
}
@Override
public boolean mouseMoved(int screenX, int screenY) {
    return false;
}
@Override
public boolean scrolled(int amount) {
    return false;
}
});
}

```

На каждой отрисовке кадра игры необходимо обновлять ее логику. Сделать это можно переопределив метод `ApplicationAdapter.render()`:

```

@Override
public void render() {
    ...

```

```
human.moveTo(humanLocation, 2);
computer.makeDecision(blackboard);
Vector2 computerLocation = (Vector2) blackboard.get("computerLocation");
computer.moveTo(computerLocation, 1);
...
}
```

Результат работы примера представлен на рисунках 2.4 и 2.5.

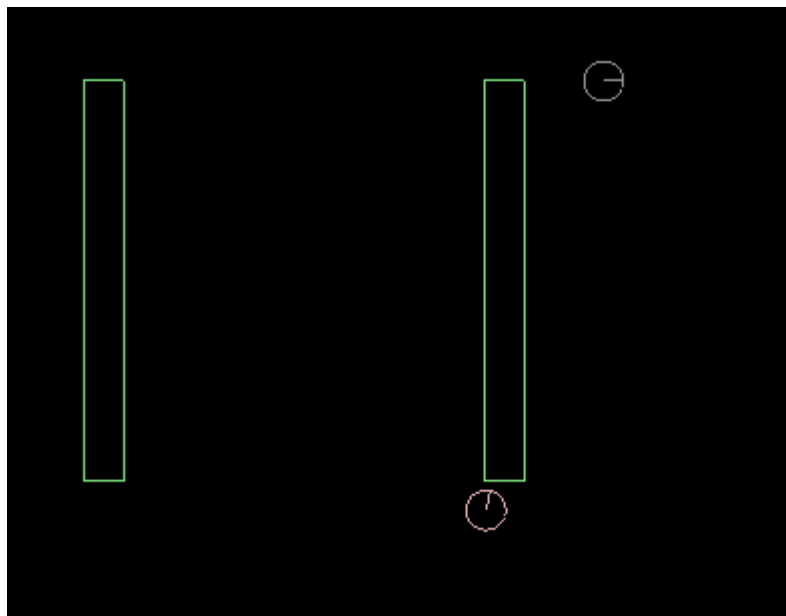


Рисунок 2.4 - Человек прячется от компьютера

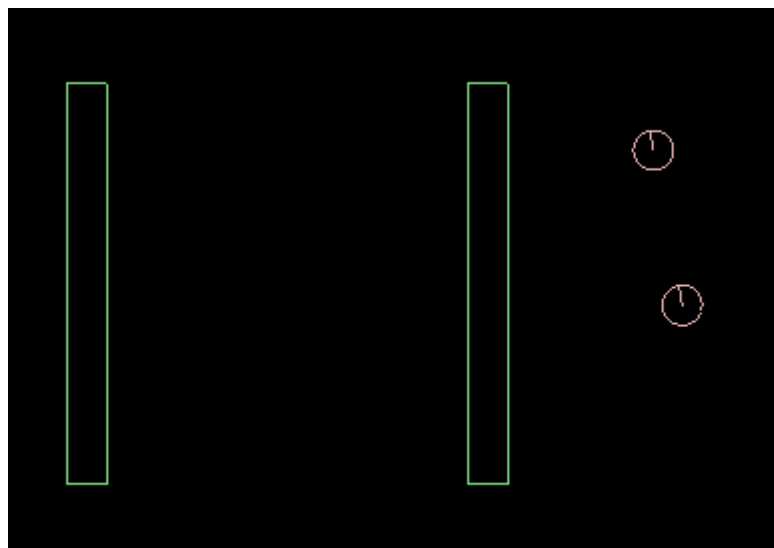


Рисунок 2.5 - Компьютер преследует человека

На рисунке 2.4 круг под прямоугольником представляет объект человека. В данном случае компьютер не видит человека, поэтому находится на месте. На рисунке 2.5 компьютер видит человека и идет вслед за ним.

3 Визуальный редактор стратегий

3.1 Выбор инструментов и технологий

Для реализации визуального средства создания деревьев поведений мы выбрали декларативный язык программирования QML с использованием скриптов на JavaScript под управлением и интеграцией компонентов Python с использованием библиотеки PyQt5.

Код на языке QML занимает меньше места, в отличие от других декларативных языков, таких как Xaml, HTML, FXML. Данный факт объясняется тем, что язык QML имеет много общего с языком JSON, тогда как большинство декларативных языков похожи на XML.

Альтернатива Python в качестве бэкенда для QML всего одна – это C++. Выбор был сделан в пользу Python по следующей причине: производительность Python-программиста выше производительности c++ программиста [13].

3.2 Проектирование схемы компонентов

В QML есть поддержка наследования компонентов [12]. На рисунке 3.1 представлена разработанная UML-диаграмма компонентов.

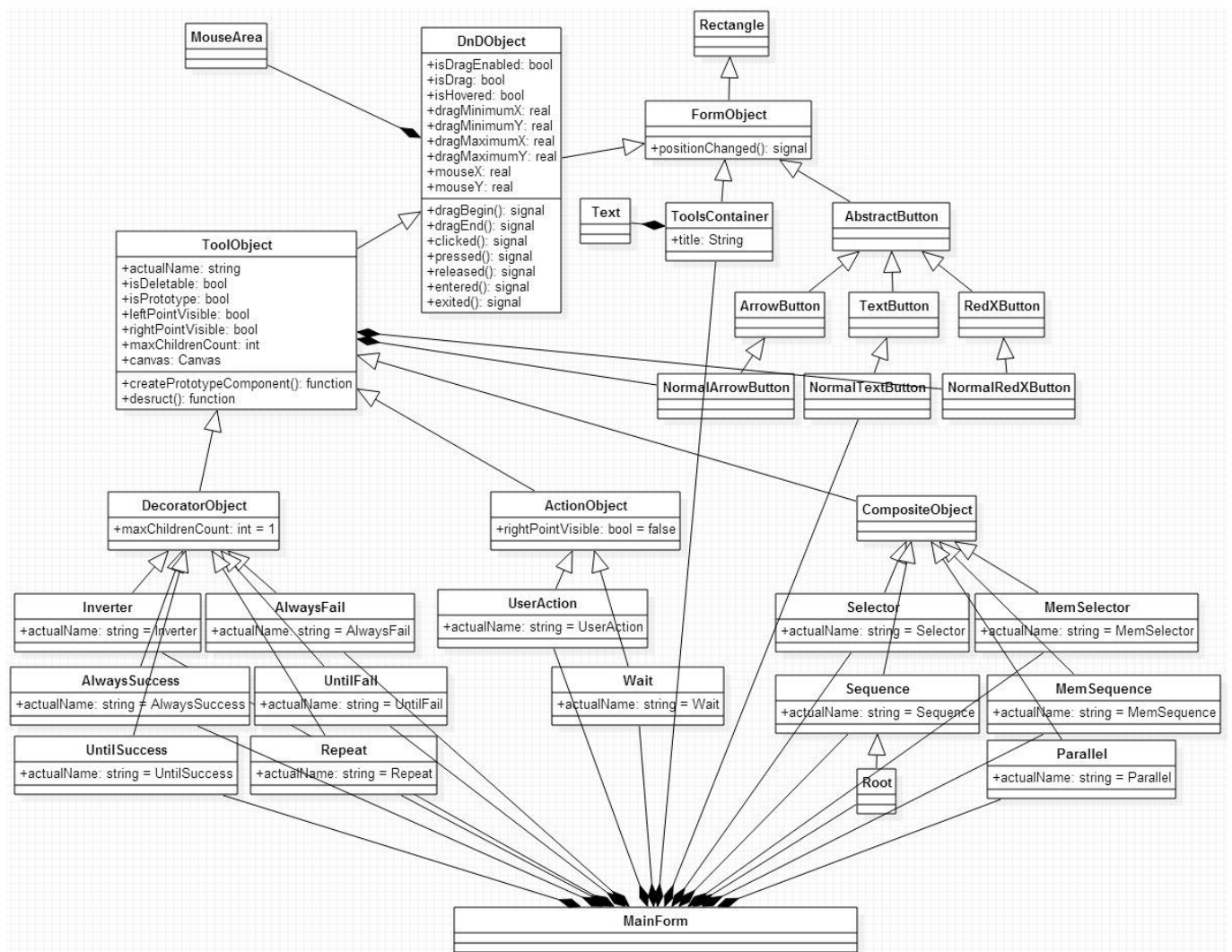


Рисунок 3.1 - UML-диаграмма компонентов QML

3.3 Описание реализованных компонентов

Корнем наследования является компонент `FormObject`, который наследуется от стандартного компонента `Rectangle`. Данный компонент содержит единственный сигнал, которого нет в `Rectangle` – `onPositionChanged`. Этот сигнал возбуждается при изменении положения объекта.

Компонент `DnDObject` представляет объекты, которыми можно оперировать при помощи Drag-and-drop способа. Описание компонента `DnDObject` представлено в таблице 3.1.

Таблица 3.1 – Описание компонента DnDObject

Тип элемента	Наименование	Описание
Поле	isDragEnabled	Bool: true, если манипулирование при помощи Drag-and-drop разрешено
Поле	isDrag	Bool: true, если элемент перетаскивается мышью
Поле	isHovered	Bool: true, если мышь находится над элементом
Поле	dragMinimumX	Real: минимальное значение координаты x объекта
Поле	dragMaximumX	Real: максимальное значение координаты x объекта
Поле	dragMinimumY	Real: минимальное значение координаты y объекта
Поле	dragMaximumY	Real: максимальное значение координаты y объекта
Поле	mouseX	Real: значение x-координаты курсора мыши относительно данного объекта
Поле	mouseY	Real: значение y-координаты курсора мыши относительно данного объекта
Сигнал	dragBegin	возбуждается на начало манипулирования при помощи Drag-and-drop
Сигнал	dragEnd	возбуждается после окончания манипулирования при помощи Drag-and-drop
Сигнал	Clicked	возбуждается после клика мышью на объект
Сигнал	Pressed	возбуждается при нажатии мышью на объект
Сигнал	Released	возбуждается после отжатия мыши на объекте
Сигнал	Entered	возбуждается при входе курсора мыши в поле объекта
Сигнал	Exited	возбуждается при выходе курсора мыши за поле

Продолжение таблицы 3.1

Тип элемента	Наименование	Описание
		объекта

Компонент ToolObject представляет объекты вершин дерева поведения, расположенные на левой панели инструментов. Полное описание ToolObject приведено в таблице 3.2.

Таблица 3.2 – Описание компонента ToolObject

Тип элемента	Наименование	Описание
Поле	actualName	String: имя элемента дерева поведения, используется при генерации дерева
Поле	isDeletable	Bool: true, если на объекте появляется кнопка для удаления
Поле	isPrototype	Bool: true, если объект находится на панели инструментов
Поле	leftPointVisible	Bool: true, если левая кнопка активна в данный момент
Поле	rightPointVisible	Bool: true, если правая кнопка активна в данный момент
Поле	maxChildrenCount	Int: максимальное количество присоединенных объектов
Функция	createPrototype	Возвращает созданный объект ToolObject
Функция	destruct	Удаляет вершину и связанные соединяющие кривые

Компонент CompositeObject является родительским ко всем компонентам, определяющим вершины композиты. На данный момент этот компонент не реализует внутри себя ничего. В дальнейшем он будет определять некоторые общие параметры для всех вершин композитов.

Компонент ActoinObject является родительским ко всем компонентам,

определяющим вершины листы. Так как у всех листов нет дочерних вершин, то в этом компоненте определена переменная `rightPointVisible` значением `false`.

Компонент `DecoratorObject` является родительским ко всем компонентам, определяющим вершины декораторы. Так как вершина декоратор имеет только одну дочернюю вершину, то в данном компоненте определена переменная `maxChildrenCount` значением 1.

Дочерние компоненты к `DecortorObject`, `ActionObject`, `CompositeObject` определяют лишь переменную `actualName` соответствующей строкой (`Inverter` – «`Inverter`», `AlwaysFail` – «`AlwaysFail`», `AlwaysSuccess` – «`AlwaysSuccess`» и т.д.), и содержат идентифицирующую иконку или текст, для того, чтобы можно было визуально отличить один компонент от другого.

Компонент `ToolsContainer` определяет прямоугольник с подписью, который используется как контейнер для вершин различных типов. Подробное описание `ToolsContainer` приведено в таблице 3.3.

Таблица 3.3 – Описание компонента `ToolsContainer`

Тип элемента	Наименование	Описание
Поле	<code>Title</code>	<code>String</code> : название контейнера, расположено левом верхнем углу

Компонент `AbstractButton` служит родительским для всех кнопок и содержит необходимые сигналы. Подробное описание компонента `AbstractButton` приведено в таблице 3.4.

Таблица 3.4 – Описание компонента `AbstractButton`

Тип элемента	Наименование	Описание
Поле	<code>isHovered</code>	<code>Bool</code> : <code>true</code> , если мышь находится над элементом
Поле	<code>isPressed</code>	<code>Bool</code> : <code>true</code> , если мышь была нажата, находясь над элементом
Поле	<code>mouseX</code>	<code>Real</code> : значение x-координаты курсора мыши

Продолжение таблицы 3.4

Тип элемента	Наименование	Описание
		относительно данного объекта
Поле	mouseY	Real: значение у-координаты курсора мыши относительно данного объекта
Сигнал	Clicked	возбуждается после клика мышью на объект
Сигнал	Pressed	возбуждается при нажатии мышью на объект
Сигнал	Released	возбуждается после отжатия мыши на объекте
Сигнал	Entered	возбуждается при входе курсора мыши в поле объекта
Сигнал	Exited	возбуждается при выходе курсора мыши за поле объекта
Сигнал	changeMousePosition	возбуждается, при движении мыши над объектом

Компонент `ArrowButton` представляет объект, который используется в `ToolObject` как кнопка, в которую могут входить соединительные кривые, и из которой могут выходить соединительные кривые. Подробное описание объекта `ArrowButton` представлено в таблице 3.5

Таблица 3.5 – Описание компонента `ArrowButton`

Тип элемента	Наименование	Описание
Сигнал	drawArrow	Возбуждается при перемещении курсора мыши при условии, что была нажата кнопка мыши в поле объекта и не отжата
Сигнал	drawArrowBegin	Возбуждается при первом перемещении курсора мыши при условии, что была нажата кнопка мыши в поле объекта и не отжата

Продолжение таблицы 3.5

Тип элемента	Наименование	Описание
Сигнал	drawArrowEnd	Возбуждается после отжатия кнопки мыши при условии, что был возбужден сигнал drawArrowBegin ранее

Компонент `TextButton` представляет объекты кнопок, на которых написан по центру текст. Подробное описание `TextButton` представлено в таблице 3.6.

Таблица 3.6 – Описание компонента `TextButton`

Тип элемента	Наименование	Описание
Поле	Text	String: текст, написанный на кнопке

Компонент `RedXButton` представляет объекты кнопок с текстом «X».

Компоненты с приставкой `Normal*` определяют цвета используемые на то или иное событие (нажатие кнопки мыши, наведение курсора мыши).

Для реализации соединительных кривых мы написали расширение для QML на Python – `BezierCurve`. В данном расширении мы переопределили стандартный метод `paint` для отрисовки кривой Безье. Полное описание данного расширения представлено в таблице 3.7.

Таблица 3.7 – Описание компонента `BezierCurve`

Поле	Наименование	Описание
Поле	startX	Real: x-координата точки начала кривой
Поле	startY	Real: y-координата точки начала кривой
Поле	endX	Real: x-координата точки конца кривой
Поле	endY	Real: y-координата точки конца кривой
Поле	curveWidth	Int: ширина кривой
Поле	startArrow	Bool: true, если есть стрелочка в начале кривой
Поле	endArrow	Bool: true, если есть стрелочка в конце кривой
Сигнал	startXChnaged	Возбуждается при изменении startX
Сигнал	startYChnaged	Возбуждается при изменении startY

Продолжение таблицы 3.7

Поле	Наименование	Описание
Сигнал	endXChnaged	Возбуждается при изменении endX
Сигнал	endYChnaged	Возбуждается при изменении endY

3.4 Пример использования редактора

Заключение

Заключение должно содержать краткую характеристику результатов выполненной работы (результатов решения поставленных задач), и рекомендации по улучшению системы.

Список использованных источников

1. Towards a Unified Behavior Trees Framework for Robot Control / A. Marzinotto [и др.]
.- Swedish Research Council and the European Union Project, 2013 .- 8 с.
2. Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees
/ P. Ogren .- Swedish Defence Research Agency, Stockholm, 2012 .- 8 с.
3. Behavior Trees for Hierarchical RTS AI / S. Delmer .- Plano: The Guildhall at SMU,
2013.- 10 с.
4. Unreal Engine 4 Documentation - <https://docs.unrealengine.com/latest/INT/>
5. LibGDX Documentation - <https://github.com/libgdx/libgdx/wiki>
6. Unity3D Documentation - <http://docs.unity3d.com/Manual/index.html>
7. Unity3D site – <https://unity3d.com>
8. Unity3D Asset Store - <https://www.assetstore.unity3d.com>
9. Doxygen
- 10.UUID <https://tools.ietf.org/html/rfc4122>
- 11.Box2D <http://box2d.org/manual.pdf>
- 12.QML <http://doc.qt.io/qt-5/qtqml-index.html>
- 13.<https://dot.kde.org/2006/08/09/phil-thompson-talks-about-pyqt>

Приложение А Задание на выполнение бакалаврской работы

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
«Алтайский государственный технический университет им. И. И. Ползунова»

УТВЕРЖДАЮ

Заведующий кафедрой ПМ

_____ Кантор С.А.
подпись *ФИО*

ЗАДАНИЕ № НА ВЫПОЛНЕНИЕ БАКАЛАВРСКОЙ РАБОТЫ

по направлению подготовки 231000 «Программная инженерия»
по профилю Разработка программно-информационных систем
студенту группы Никитину Алексею Александровичу
фамилия, имя, отчество

Тема Проектирование библиотеки и реализация визуального средства
создания стратегий поведения виртуальных игровых персонажей

Утверждена приказом ректора от _____ № _____

Срок выполнения работы _____

Задание принял к исполнению: _____

подпись

ФИО

Барнаул 2014 г.

1 Исходные данные

2. Содержание разделов работы

Наименование разделов работы и их содержание	Трудо-ёмкость, %	Срок выполнения	Консультант (Ф.И.О., подпись)
1 Расчетно-пояснительная записка			

2 Графическая часть			

3. Научно-библиографический поиск

3.1. По научно-технической литературе просмотреть Реферативные журналы

_____ за последние ____ года и научно-
технические журналы

_____ за последние ____ года.

3.2. По нормативной литературе просмотреть указатели государственных и отраслевых стандартов за последний год.

3.3. Патентный поиск провести за ____ лет по странам

Руководитель работы: Старолетов С.М.

Ф.И.О.

подпись

Приложение Б Руководство пользователя

Приложение В Исходный код программы

Приложение Г Объемы рынка игр

По оценке J'son & Partners Consulting, в 2015 году объем мирового рынка игр составит 88,4 миллиарда долларов, а объем российского рынка игр составит 1,83 миллиарда долларов. В целом рынок игр будет стабильно развиваться.

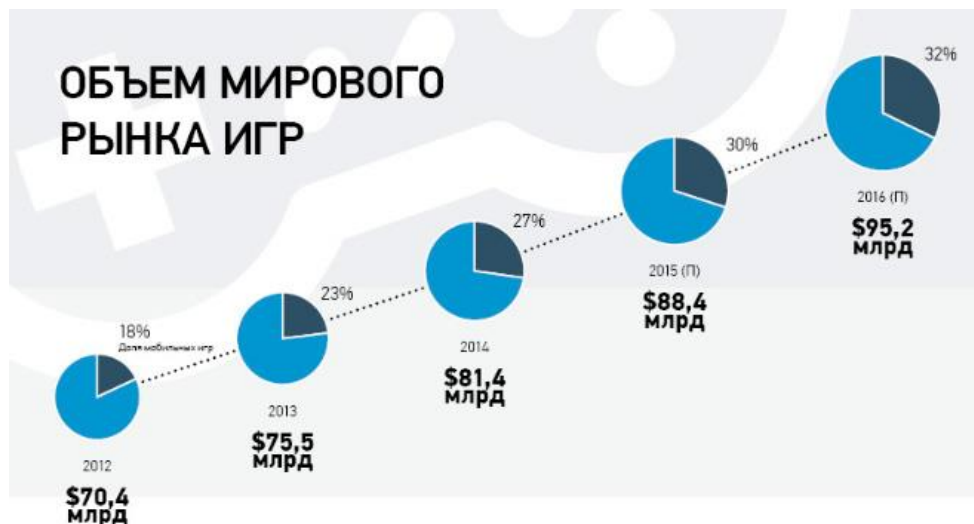


Рисунок Г.1 - Объем мирового рынка игр



Источник: J'son & Partners Consulting

Рисунок Г.2 - Объем российского рынка игр

Приложение Д Документация к библиотеке Behavior Tree

Класс Decorators.AlwaysFailure

Граф наследования Decorators.AlwaysFailure представлен на рисунке Д.1.

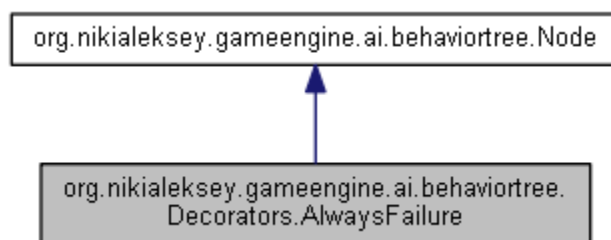


Рисунок Д.1 - Граф наследования Decorators.AlwaysFailure

Граф связей класса Decorators.AlwaysFailure представлен на рисунке Д.2.

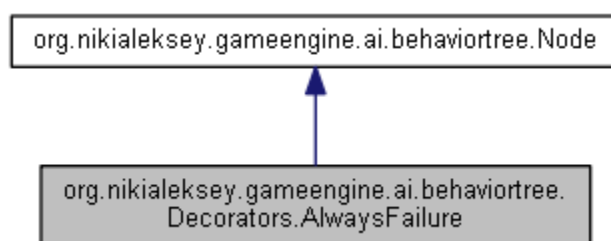


Рисунок Д.2 - Граф связей класса Decorators.AlwaysFailure

Открытые члены

- **AlwaysFailure** (**Node** node)
- void **enter** (**Tick** tick)
- void **open** (**Tick** tick)
- **Status** tick (**Tick** tick)
- void **close** (**Tick** tick)
- void **exit** (**Tick** tick)

Подробное описание

Класс представляет декоратор, который всегда возвращает статус FAILURE.

Автор:

Alexey Nikitin

Конструктор(ы)

Decorators.AlwaysFailure.AlwaysFailure (**Node** *node*)

Конструктор.

Аргументы:

- node - дочерняя вершина

Методы

Status Decorators.AlwaysFailure.tick (Tick *tick*)

Передаёт сигнал на исполнение дочерней вершине, всегда возвращает сигнал FAILURE

Аргументы:

- `tick` - объект тика

Возвращает:

FAILURE

Граф вызовов представлен на рисунке Д.3:

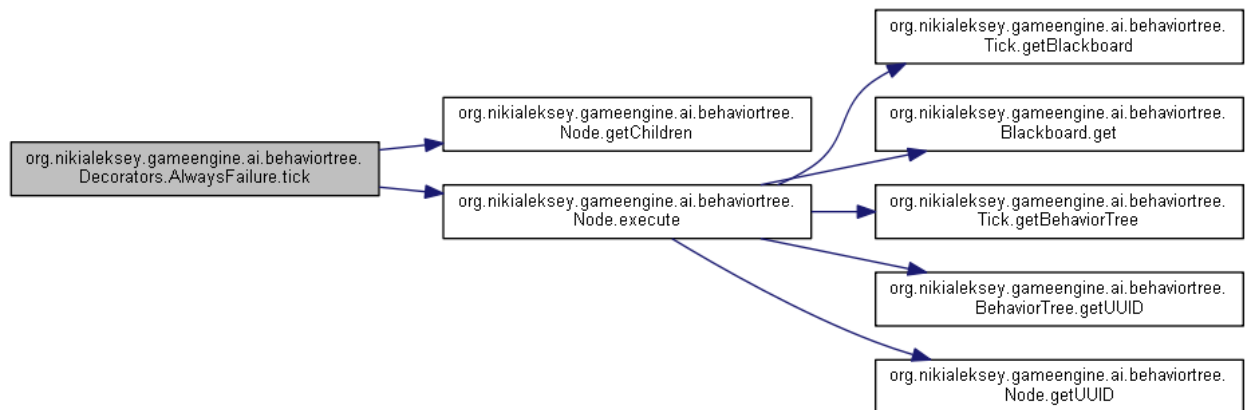


Рисунок Д.3 - Граф вызовов метода `AlwaysFailure.tick()`

Объявления и описания членов класса находятся в файле:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Decorators/AlwaysFailure.java`

Класс Decorators.AlwaysSuccess

Граф наследования `Decorators.AlwaysSuccess` представлен на рисунке Д.4:

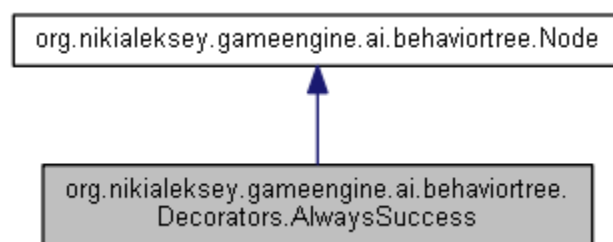


Рисунок Д.4 - Граф наследования `Decorators.AlwaysSuccess`

Граф связей класса `Decorators.AlwaysSuccess` представлен на рисунке Д.5:

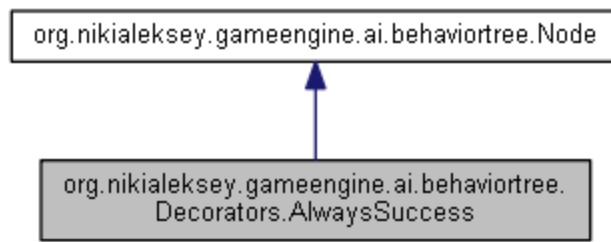


Рисунок Д.5 - Граф связей класса Decorators.AlwaysSuccess

Открытые члены

- **AlwaysSuccess** (**Node** node)
- void **enter** (**Tick** tick)
- void **open** (**Tick** tick)
- **Status** tick (**Tick** tick)
- void **close** (**Tick** tick)
- void **exit** (**Tick** tick)

Подробное описание

Класс представляет декоратор, который всегда возвращает статус SUCCESS.

Автор:

Alexey Nikitin

Конструктор(ы)

Decorators.AlwaysSuccess.AlwaysSuccess (**Node** *node*)

Конструктор.

Аргументы:

- node - дочерняя вершина

Методы

Status Decorators.AlwaysSuccess.tick (**Tick** *tick*)

Передаёт сигнал на исполнение дочерней вершине, всегда возвращает сигнал SUCCESS

Аргументы:

- tick - объект тика

Возвращает:

SUCCESS

Граф вызовов представлен на рисунке Д.6:

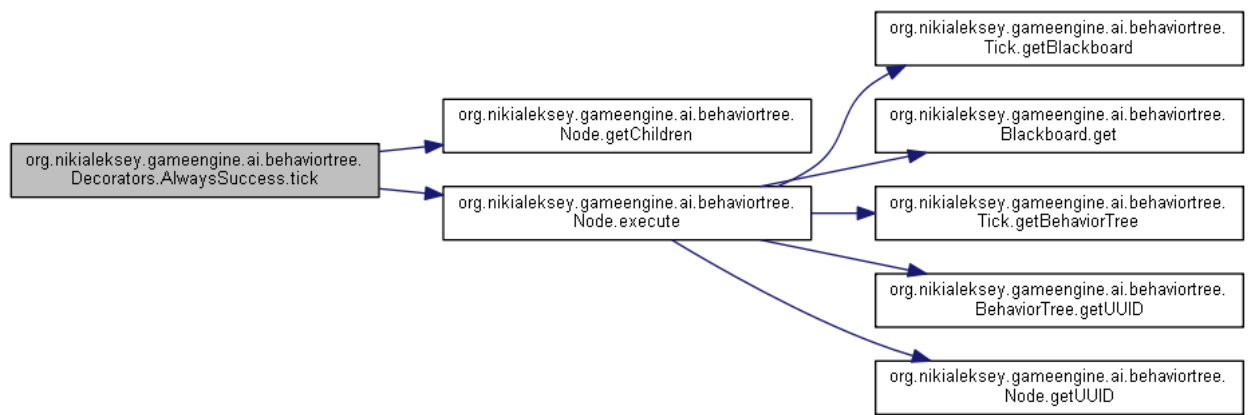


Рисунок Д.6 - Граф вызовов метода AlwaysSuccess.tick()

Объявления и описания членов класса находятся в файле:

- src/org/nikialeksey/gameengine/ai/behaviortree/Decorators/AlwaysSuccess.java

Класс BehaviorTree

Открытые члены

- **BehaviorTree** (Node root)
- **Status execute** (Blackboard blackboard)
- String **getUUID** ()

Подробное описание

Класс представляет дерево поведения.

Автор:

Alexey Nikitin

Конструктор(ы)

BehaviorTree.BehaviorTree (Node *root*)

Конструктор дерева поведения. Назначает уникальный идентификатор и устанавливает ссылку на корень дерева поведения.

Аргументы:

- root - корень дерева поведения, именно этой вершине будут передаваться сигналы на исполнение

Методы

Status BehaviorTree.execute (Blackboard *blackboard*)

Создает объект тика и передает сигнал на исполнение корню дерева поведения

Аргументы:

- blackboard - память для принятия решения

Возвращает:

объект статуса

Граф вызовов представлен на рисунке Д.7:

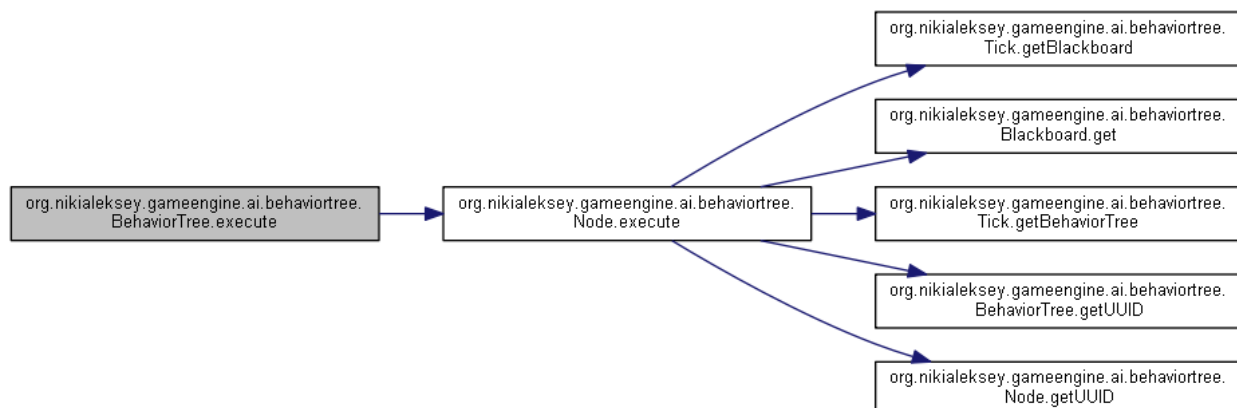


Рисунок Д.7 - Граф вызовов метода `BehaviorTree.execute()`

String `BehaviorTree.getUUID ()`

Возвращает уникальный идентификатор дерева поведения.

Возвращает:

строка, представляющая уникальный идентификатор дерева поведения.

Граф вызова представлен на рисунке Д.8:

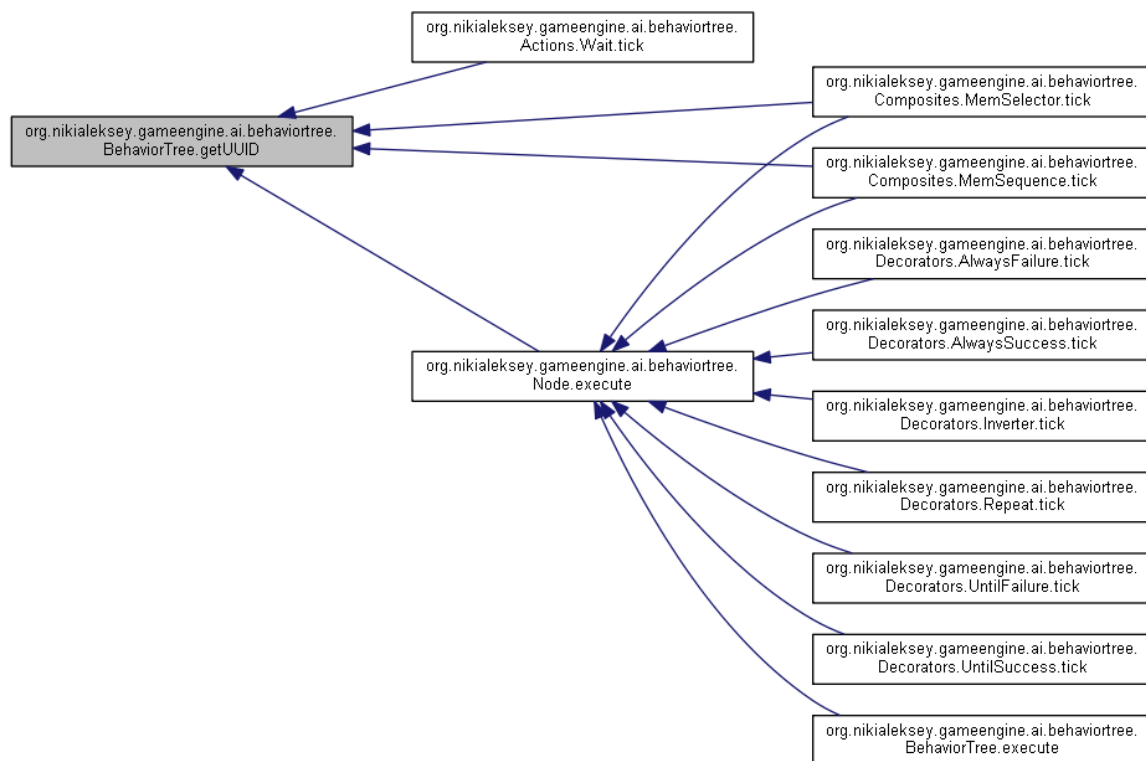


Рисунок Д.8 - Граф вызовов метода `BehaviorTree.getUUID()`

Объявления и описания членов класса находятся в файле:

- src/org/nikialeksey/gameengine/ai/behaviortree/BehaviorTree.java

Класс Blackboard

Классы

- class **BlackboardMemory**
- class **TreeMemory**

Открытые члены

- **Blackboard** ()
- Object **get** (String key)
- Object **get** (String key, String treeUUID)
- Object **get** (String key, String treeUUID, String nodeUUID)
- void **put** (String key, Object value)
- void **put** (String key, Object value, String treeUUID)
- void **put** (String key, Object value, String treeUUID, String nodeUUID)

Подробное описание

Класс представляет структуру памяти для наших персонажей, которую будут использовать вершины дерева поведения. Информация, хранящаяся в

Blackboard

структурирована следующим образом: глобальная информация(доступная из любого места), информация о дереве (доступная для всех вершин одного дерева), информация о вершине (доступная только вершине).

Память для простоты можно изобразить в виде JSON документа:

```
{
  'global key 1': globalObject1,           #
  'global key 2': globalObject2,           #  глобальная информация
  ...                                       #
  'tree1UUID': {
    'tree key 1': treeObject1,              #
    'tree key 2': treeObject2,              #  информация, доступная для всех
    ...                                     #  вершин одного дерева
    'nodeMemory': {
      'node key 1': nodeObject1,            #
      'node key 2': nodeObject2,            #  информация, доступная для одной вершины
      ...
    },
  },
  'tree2UUID': {
    ...
  },
  ...
}
```

Автор:

Alexey Nikitin

Конструктор(ы)

Blackboard.Blackboard ()

Конструктор. Инициализирует память.

Методы

Object Blackboard.get (String *key*)

Возвращает объект из глобальной памяти.

Аргументы:

- *key* - ключ

Возвращает:

объект из глобальной памяти

Граф вызова функции представлен на рисунке Д.9:

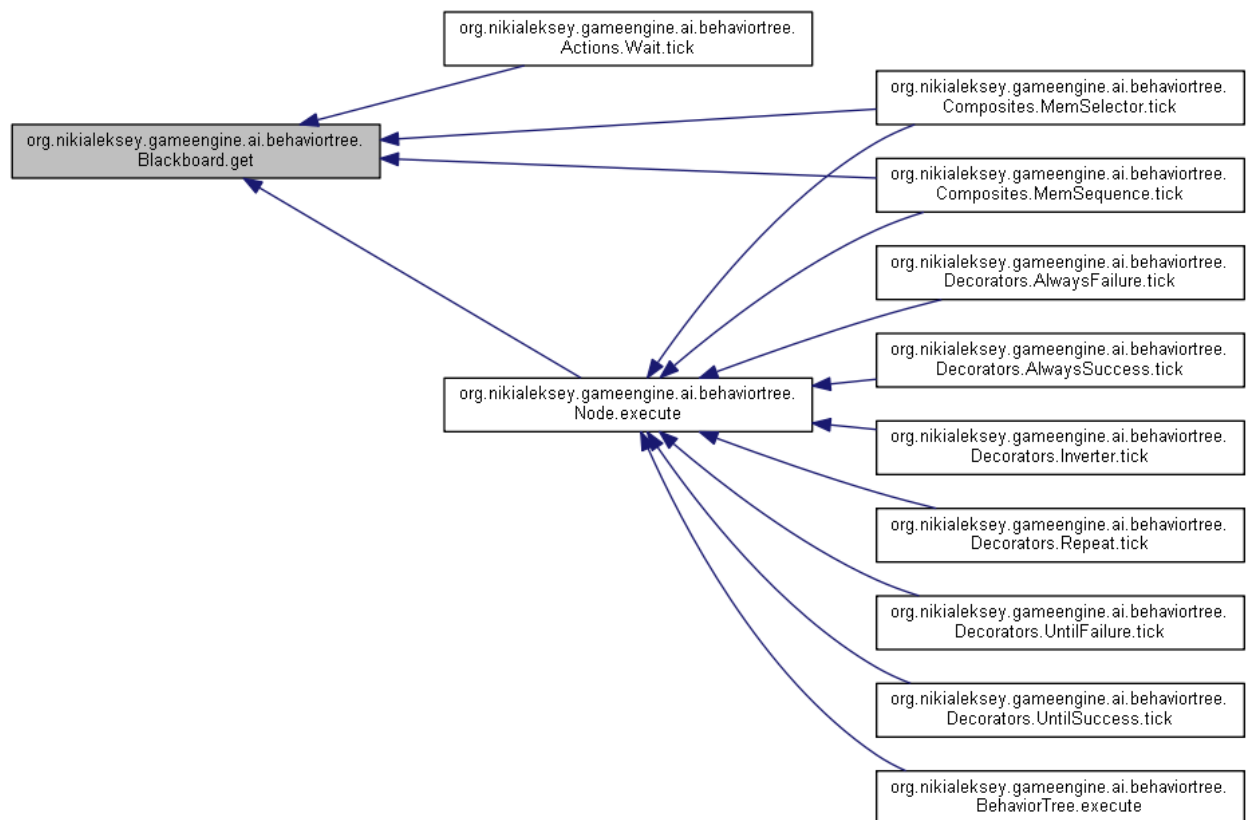


Рисунок Д.9 - Граф вызова метода `Blackboard.get()`

Object `Blackboard.get (String key, String treeUUID)`

Возвращает объект из памяти дерева с идентификатором

`treeUUID`

Аргументы:

- *key* – ключ
- *treeUUID* - уникальный идентификатор дерева

Возвращает:

объект из памяти дерева с идентификатором
`treeUUID`

`Object Blackboard.get (String key, String treeUUID, String nodeUUID)`

Возвращает объект из вершины дерева.

Аргументы:

- `key` – ключ
- `treeUUID` - уникальный идентификатор дерева
- `nodeUUID` - уникальный идентификатор вершины

Возвращает:

объект из вершины дерева

`void Blackboard.put (String key, Object value)`

Кладет объект в глобальную память.

Аргументы:

- `key` – ключ
- `value` - объект

Граф вызова функции представлен на рисунке Д.10:

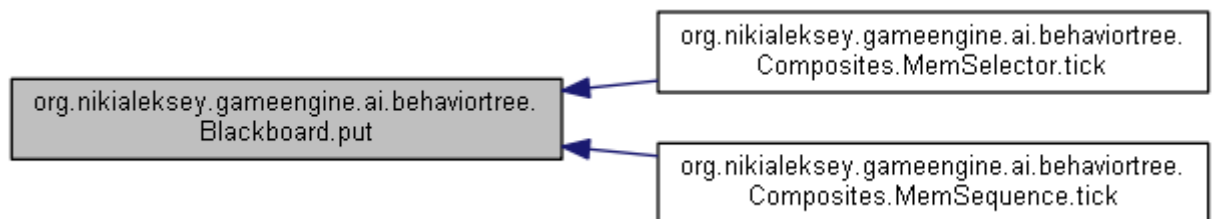


Рисунок Д.10 - Граф вызова метода `Blackboard.put()`

`void Blackboard.put (String key, Object value, String treeUUID)`

Кладет объект в память для дерева.

Аргументы:

- `key` – ключ
- `value` – объект
- `treeUUID` - уникальный идентификатор дерева

`void Blackboard.put (String key, Object value, String treeUUID, String nodeUUID)`

Кладет объект в память вершины.

Аргументы:

- key – ключ
- value – объект
- treeUUID - уникальный идентификатор дерева
- nodeUUID - уникальный идентификатор вершины

Объявления и описания членов класса находятся в файле:

- src/org/nikialeksey/gameengine/ai/behaviortree/Blackboard.java

Класс Decorators.Inverter

Граф наследования Decorators.Inverter представлен на рисунке Д.11:

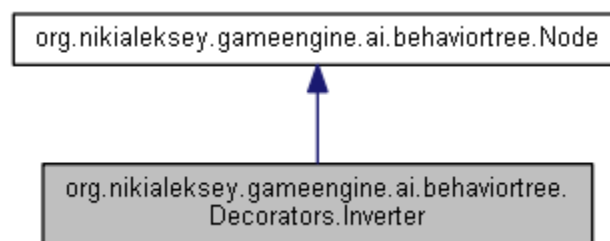


Рисунок Д.11 - Граф наследования Decorators.Inverter

Граф связей класса Decorators.Inverter представлен на рисунке Д.12:

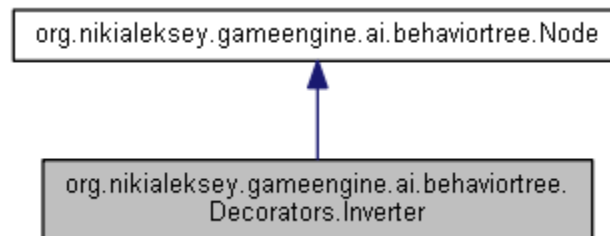


Рисунок Д.12 - Граф связей класса Decorators.Inverter

Открытые члены

- **Inverter** (**Node** node)
- void **enter** (**Tick** tick)
- void **open** (**Tick** tick)
- **Status** tick (**Tick** tick)
- void **close** (**Tick** tick)
- void **exit** (**Tick** tick)

Подробное описание

Представляет декоратор инвертер.

Автор:

Alexey Nikitin

Конструктор(ы)

Decorators.Inverter.Inverter (Node *node*)

Конструктор.

Аргументы:

- *node* – дочерняя вершина

Методы

Status Decorators.Inverter.tick (Tick *tick*)

Передаёт сигнал на исполнение дочерней вершине.

Аргументы:

- *tick* – объект тика

Возвращает:

SUCCESS, если дочерняя вершина вернула результат FAILURE; FAILURE, если дочерняя вершина вернула результат SUCCESS; иначе тот результат, который вернула дочерняя вершина.

Граф вызовов представлен на рисунке Д.13:

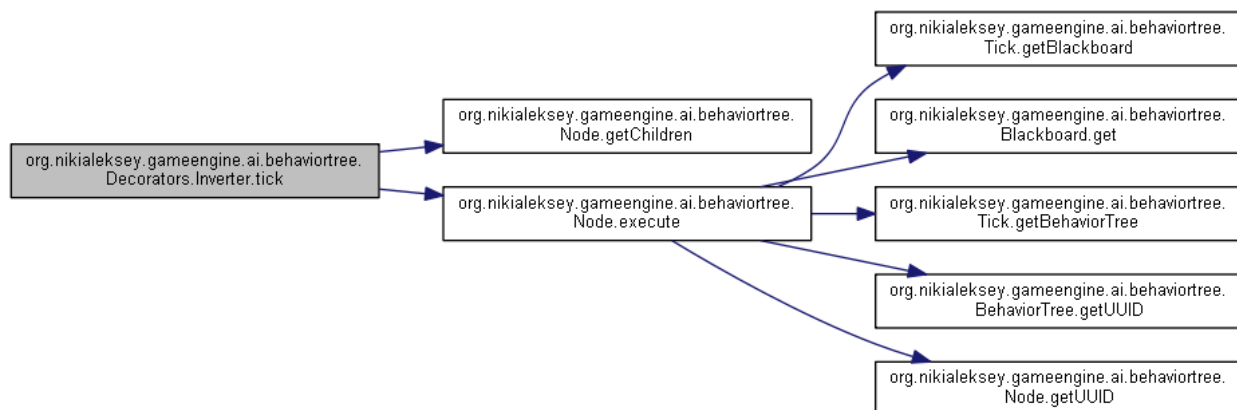


Рисунок Д.13 - Граф вызова метода Inverter.tick()

Объявления и описания членов класса находятся в файле:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Decorators/Inverter.java`

Класс Composites.MemSelector

Граф наследования Composites.MemSelector представлен на рисунке Д.14:

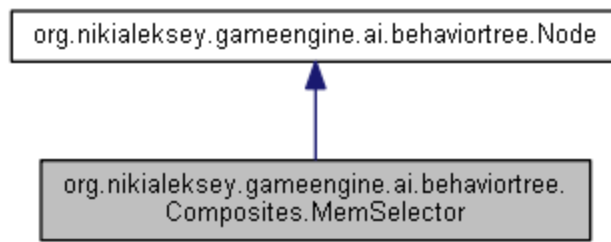


Рисунок Д.14 - Граф наследования Composites.MemSelector

Граф связей класса Composites.MemSelector представлен на рисунке Д.15:

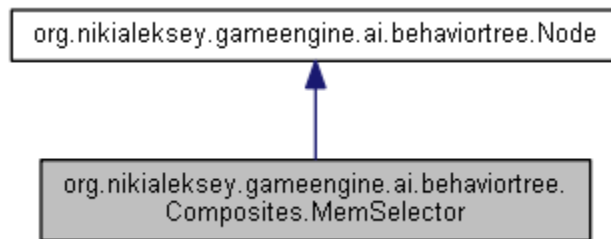


Рисунок Д.15 - Граф связей класса Composites.MemSelector

Открытые члены

- **MemSelector** (Node...nodes)
- void **enter** (Tick tick)
- void **open** (Tick tick)
- **Status tick** (Tick tick)
- void **close** (Tick tick)
- void **exit** (Tick tick)

Подробное описание

Класс представляет композит-селектор с запоминанием вершины, которая вернула результат RUNNING. На следующем тике данная вершина начнет давать сигнал на исполнение как раз последней запущенной дочерней вершине, а не с начала списка дочерних вершин.

Автор:

Alexey Nikitin

Конструктор(ы)

Composites.MemSelector.MemSelector (Node... *nodes*)

Конструктор.

Аргументы:

- *nodes* – список дочерних вершин

Методы

Status Composites.MemSelector.tick (Tick *tick*)

Передаёт сигнал на исполнение дочерним вершинам до тех пор, пока они возвращают статус FAILURE. Как только дочерняя вершина вернёт статус, отличный от FAILURE, этот статус будет сразу возвращён этой вершиной и выполнение закончится. Если это был статус RUNNING, то в таком случае будет запомнен в blackboard индекс данной дочерней вершины и в следующий раз передача сигнала на исполнение продолжится именно с этой вершины.

Аргументы:

- tick – объект тика

Возвращает:

либо статус дочерней вершины, которая вернула результат, отличный от FAILURE, либо FAILURE

Граф вызовов представлен на рисунке Д.16:

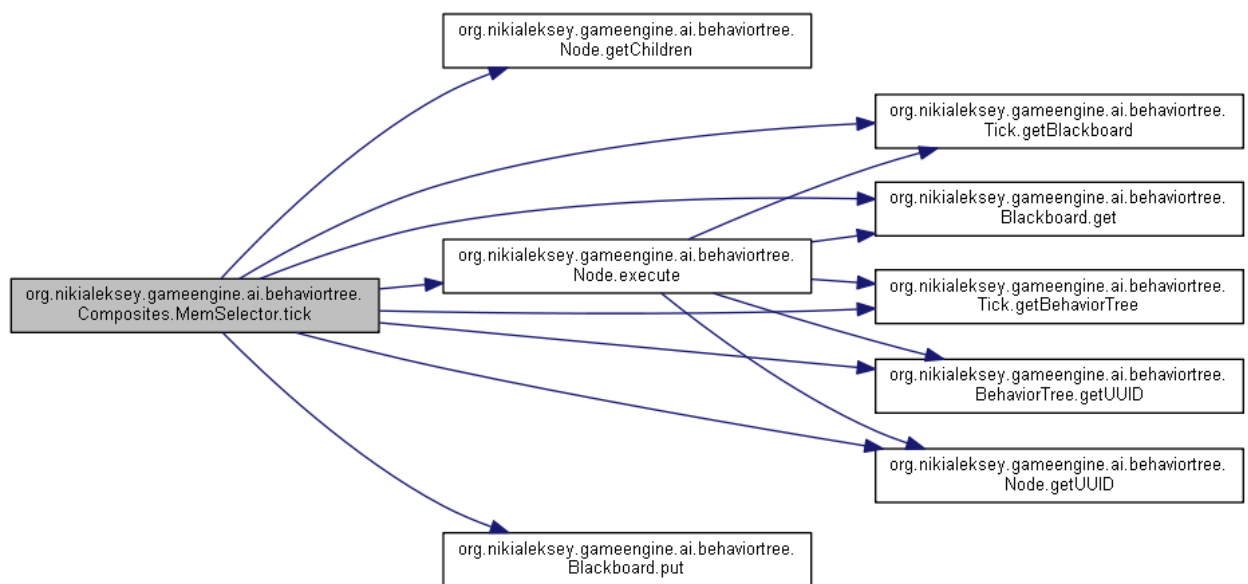


Рисунок Д.16 - Граф вызовов метода MemSelector.tick()

Объявления и описания членов класса находятся в файле:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Composites/MemSelector.java`

Класс Composites.MemSequence

Граф наследования: `Composites.MemSequence` представлен на рисунке Д.17:

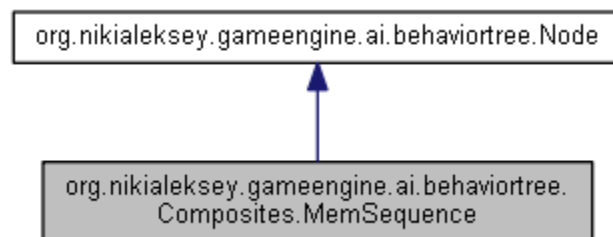


Рисунок Д.17 - Граф наследования Composites.MemSequence

Граф связей класса Composites.MemSequence представлен на рисунке Д.18:

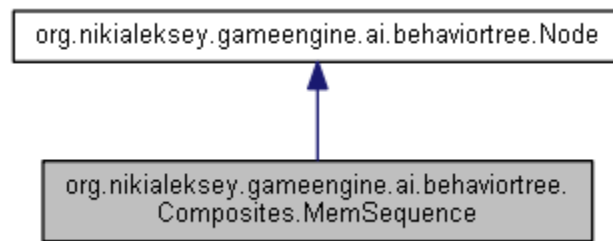


Рисунок Д.18 - Граф связей класса Composites.MemSequence

Открытые члены

- **MemSequence** (Node...nodes)
- void **enter** (Tick tick)
- void **open** (Tick tick)
- **Status** tick (Tick tick)
- void **close** (Tick tick)
- void **exit** (Tick tick)

Подробное описание

Класс представляет композит-последовательность с запоминанием вершины, которая вернула результат RUNNING. На следующем тике данная вершина начнет давать сигнал на исполнение как раз последней запущенной дочерней вершине, а не с начала списка дочерних вершин.

Автор:

Alexey Nikitin

Конструктор(ы)

Composites.MemSequence.MemSequence (Node... *nodes*)

Конструктор.

Аргументы:

- *nodes* – список дочерних вершин

Методы

Status Composites.MemSequence.tick (Tick *tick*)

Передаёт сигнал на исполнение дочерним вершинам до тех пор, пока они возвращают статус SUCCESS. Как только дочерняя вершина вернет статус, отличный от SUCCESS, этот статус будет сразу возвращен этой вершиной и выполнение закончится. Если это был статус RUNNING, то в таком случае будет запомнен в blackboard индекс данной дочерней вершины и в следующий раз передача сигнала на исполнение продолжится именно с этой вершины.

Аргументы:

- *tick* – объект тика

Возвращает:

либо статус дочерней вершины, которая вернула результат, отличный от SUCCESS, либо SUCCESS

Граф вызовов представлен на рисунке Д.19:

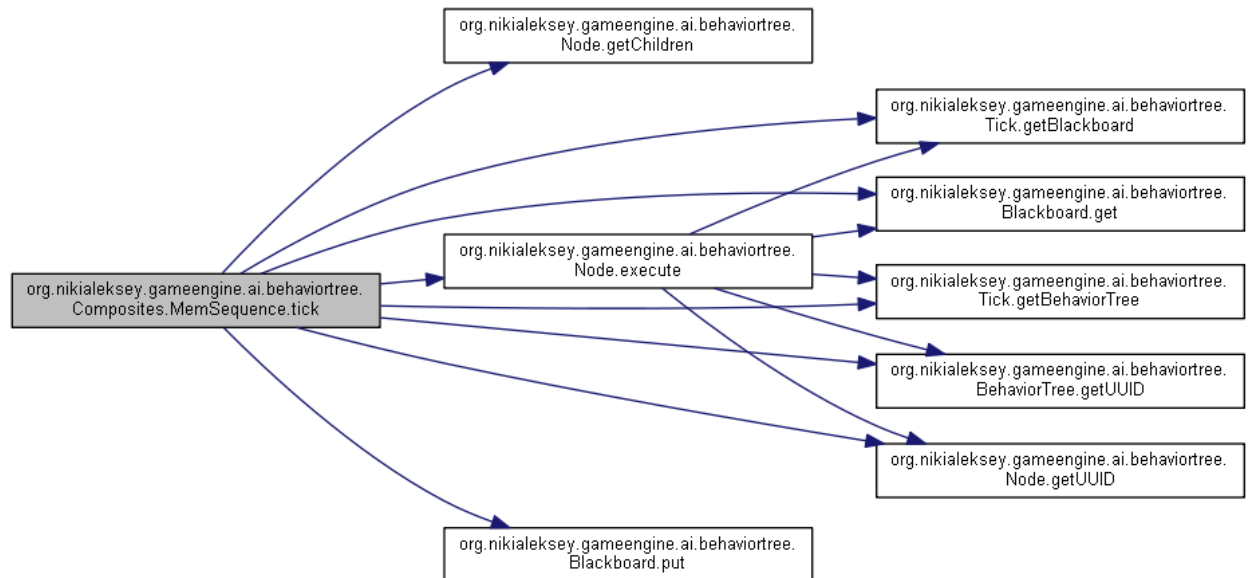


Рисунок Д.19 - Граф вызовов метода `MemSequence.tick()`

Объявления и описания членов класса находятся в файле:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Composites/MemSequence.java`

Класс Node

Граф наследования Node представлен на рисунке Д.20:

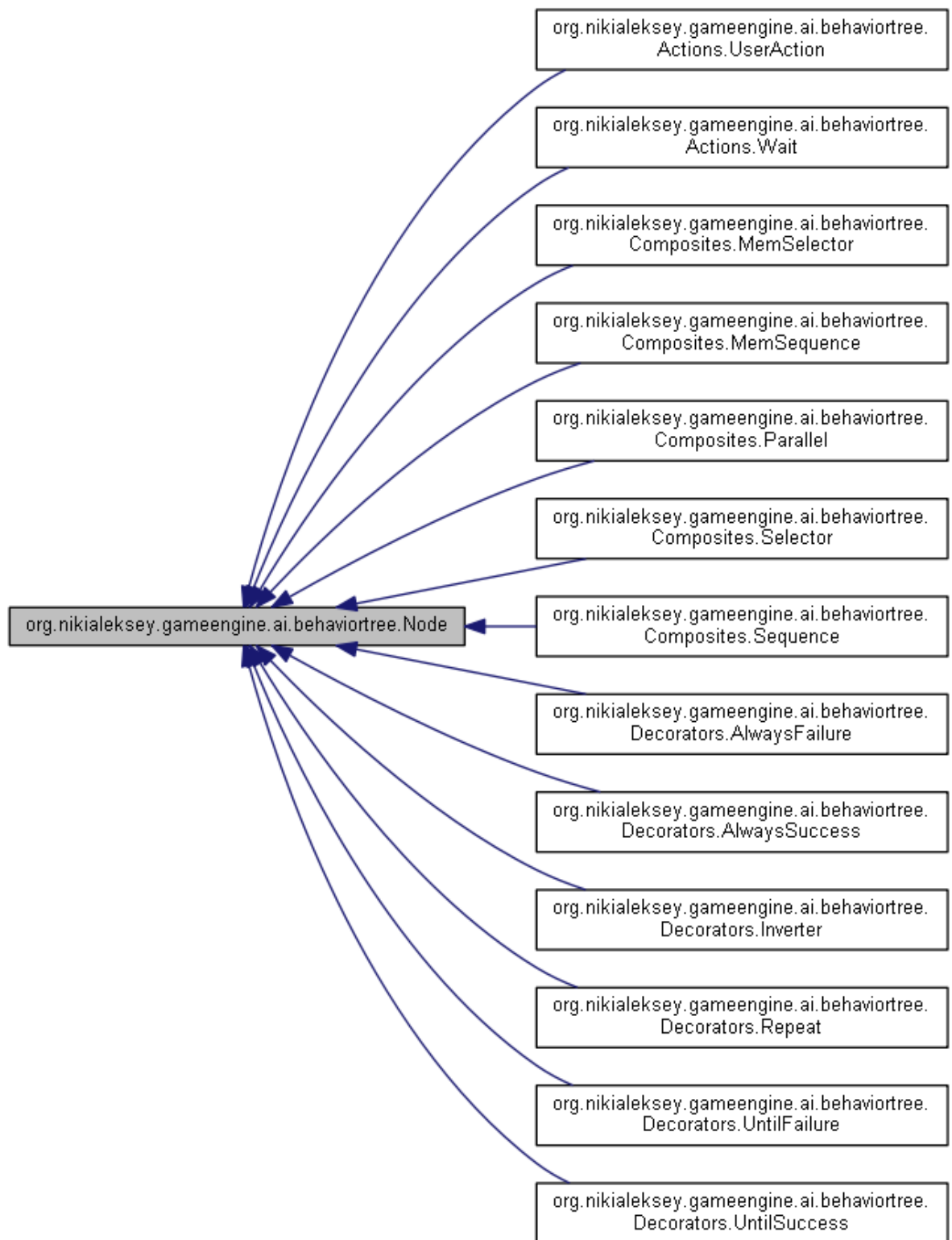


Рисунок Д.20 - Граф наследования Node

Открытые члены

- **Node** (Node...nodes)
- `ArrayList< Node > getChildren ()`

- String **getUUID** ()
- **Status execute** (Tick *tick*)
- abstract void **enter** (Tick *tick*)
- abstract void **open** (Tick *tick*)
- abstract **Status tick** (Tick *tick*)
- abstract void **close** (Tick *tick*)
- abstract void **exit** (Tick *tick*)

Подробное описание

Абстрактный класс вершины дерева повдения. Содержит необходимые методы для выполнения логики вершины.

Автор:

Alexey Nikitin

Конструктор(ы)

Node.Node (Node... *nodes*)

Конструктор.

Аргументы:

- *nodes* – дочерние вершины

Методы

abstract void Node.close (Tick *tick*) [abstract]

Вызывается, при осуществлении закрытия вершины (не вызывается после выполнении логики, если статус

RUNNING

)

Аргументы:

- *tick* – объект тика

abstract void Node.enter (Tick *tick*) [abstract]

Вызывается, при осуществлении входа в вершину

Аргументы:

- *tick* – объект тика

Status Node.execute (Tick *tick*)

Выполняет логику вершины.

Аргументы:

- *tick* – объект тика

Возвращает:

статус, после выполнения логики

Граф вызовов представлен на рисунке Д.21:

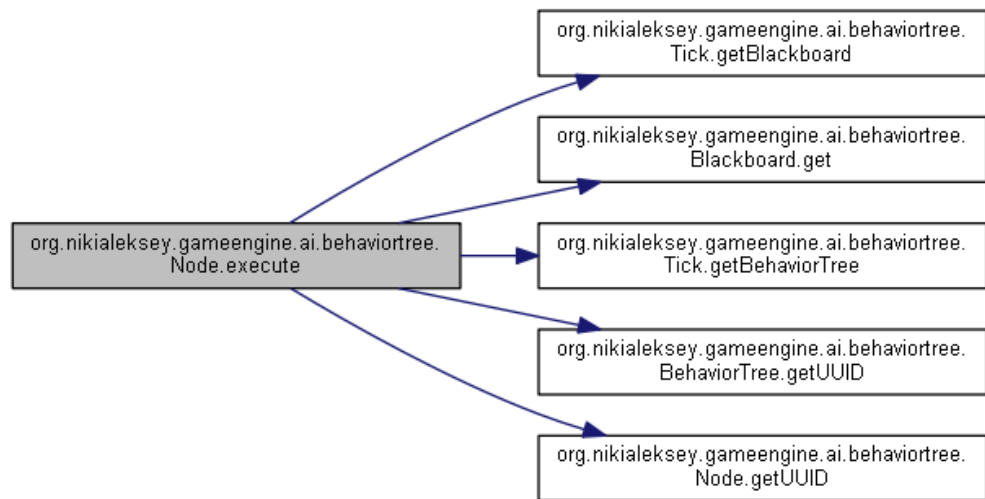


Рисунок Д.21 - Граф вызовов метода `Node.execute()`

Граф вызова функции представлен на рисунке Д.22:

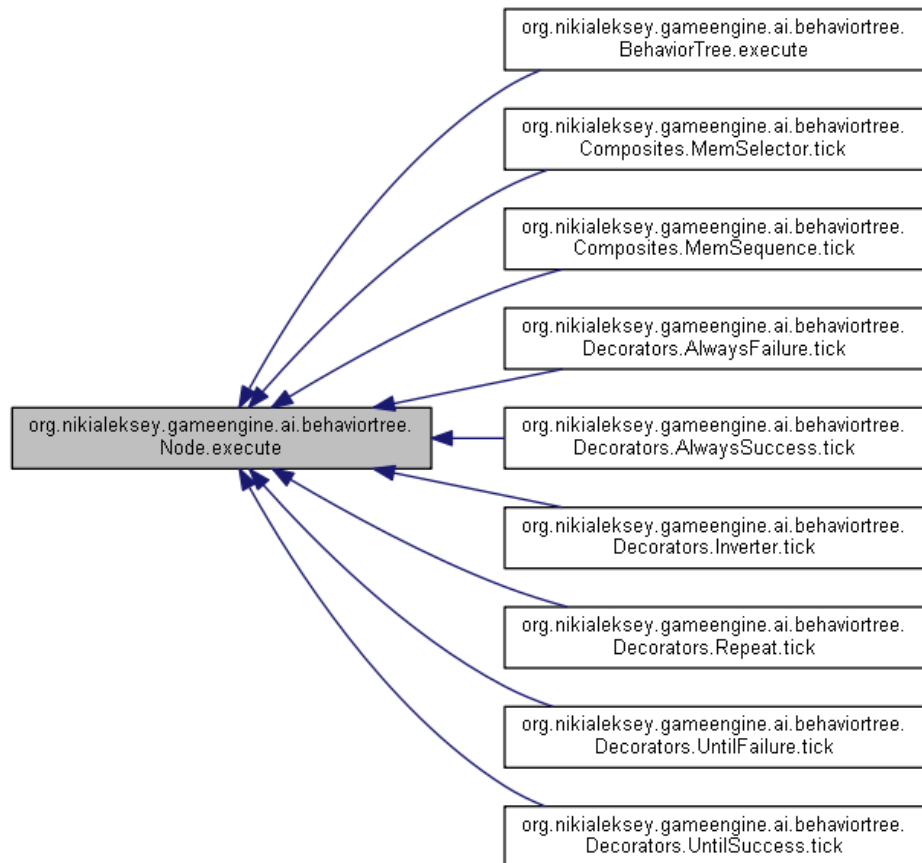


Рисунок Д.22 - Граф вызова метода `Node.execute`

```
abstract void Node.exit (Tick tick) [abstract]
```

Вызывается, при осуществлении выхода из вершины (всегда после выполнения логики, но позже закрытия вершины, если оно было)

Аргументы:

- tick – объект тика

`ArrayList<Node> Node.getChildren ()`

Возвращает список дочерних вершин

Возвращает:

список дочерних вершин

Граф вызова функции представлен на рисунке Д.23:

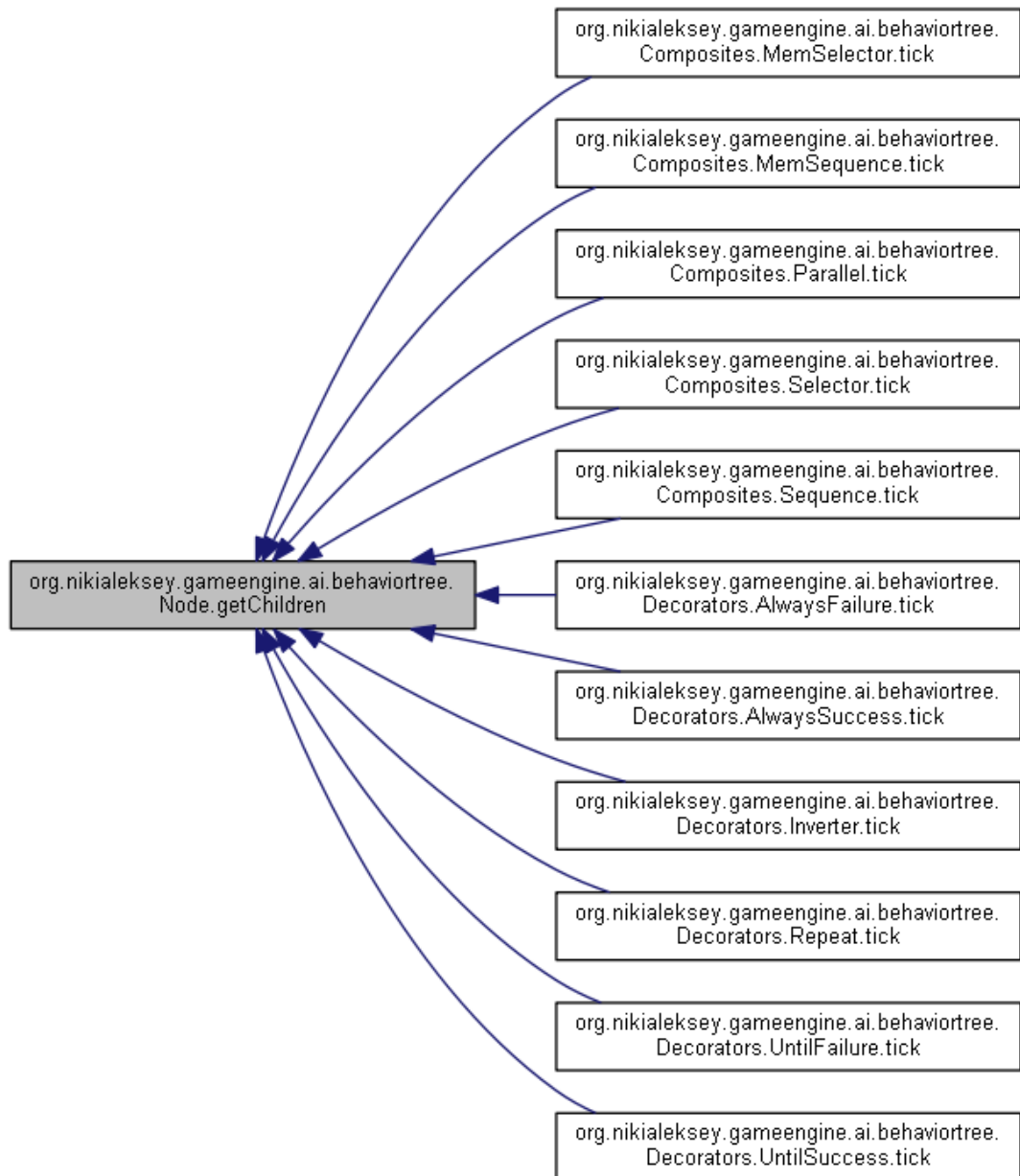


Рисунок Д.23 - Граф вызова метода `Node.getChildren()`

String Node.getUUID ()

Возвращает уникальный идентификатор вершины

Возвращает:

уникальный идентификатор вершины

Граф вызова функции представлен на рисунке Д.24:

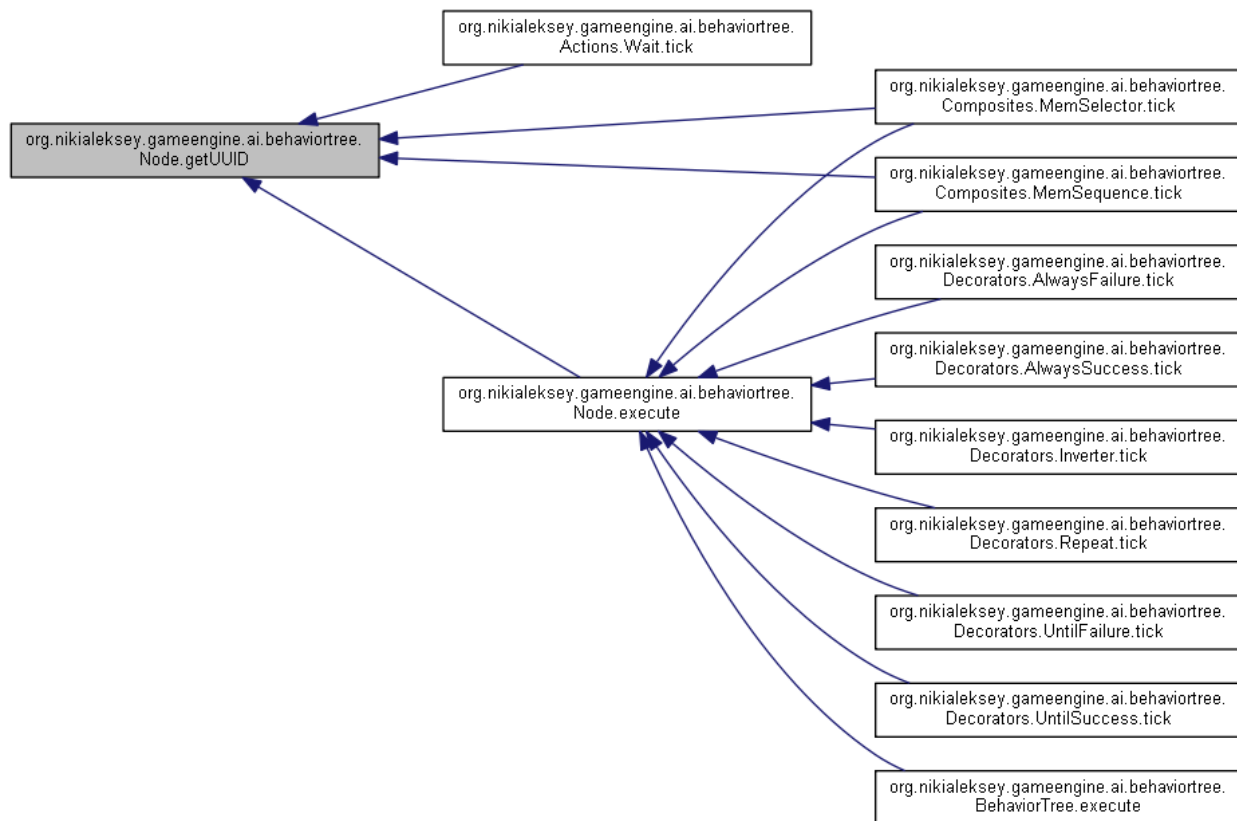


Рисунок Д.24 - Граф вызова метода Node.getUUID()

```
abstract void Node.open (Tick tick) [abstract]
```

Вызывается, при осуществлении открытия вершины (всегда после входа, но не всегда открытая вершина закрывается после выполнения логики)

Аргументы:

- `tick` – объект тика

```
abstract Status Node.tick (Tick tick) [abstract]
```

Содержит код логики вершины, после отработки возвращает один из четырех статусов: `RUNNING`, `WAIT`, `FAILURE`, `SUCCESS`

Аргументы:

- `tick` – объект тика

Возвращает:

один из четырех статусов: RUNNING, WAIT, FAILURE, SUCCESS

Объявления и описания членов класса находятся в файле:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Node.java`

Класс **Composites.Parallel**

Граф наследования **Composites.Parallel** представлен на рисунке Д.25:

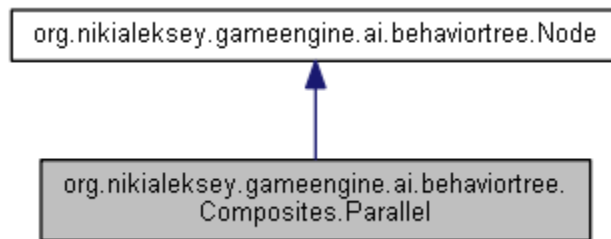


Рисунок Д.25 - Граф наследования **Composites.Parallel**

Граф связей класса **Composites.Parallel** представлен на рисунке Д.26:

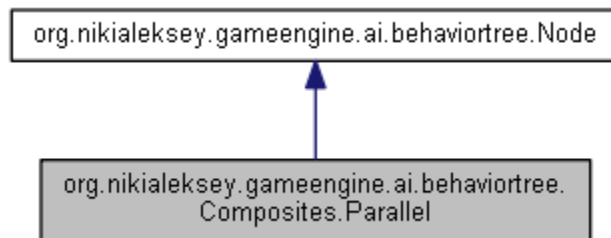


Рисунок Д.26 - Граф связей класса **Composites.Parallel**

Открытые члены

- **Parallel** (`int successCount`, `int failureCount`, `Node...nodes`)
- `void enter (Tick tick)`
- `void open (Tick tick)`
- **Status tick** (`Tick tick`)
- `void close (Tick tick)`
- `void exit (Tick tick)`

Подробное описание

Класс представляет параллельный композит.

Автор:

Alexey Nikitin

Конструктор(ы)

`Composites.Parallel.Parallel (int successCount, int failureCount, Node... nodes)`

Конструктор.

Аргументы:

- `successCount` - количество дочерних вершин, которое должно вернуть SUCCESS, чтобы данный композит вернул SUCCESS
- `failureCount` - количество дочерних вершин, которое должно вернуть FAILURE, чтобы данный композит вернул FAILURE
- `nodes` - список дочерних вершин

Методы

`Status Composites.Parallel.tick (Tick tick)`

Передаёт сигнал на исполнение всем дочерним вершинам одновременно.

Аргументы:

- `tick` – объект тика

Возвращает:

если количество дочерних вершин, вернувших результат SUCCESS больше порогового значения

`successCount`

и, при этом, количество вершин, вернувших результат FAILURE меньше порогового значения

`failureCount`

, то SUCCESS; если количество дочерних вершин, вернувших результат FAILURE больше порогового значения

`failureCount`

и, при этом, количество вершин, вернувших результат SUCCESS меньше порогового значения

`successCount`

, то FAILURE; иначе ERROR

Граф вызовов представлен на рисунке Д.27:



Рисунок Д.27 - Граф вызовов метода `Parallel.tick()`

Объявления и описания членов класса находятся в файле:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Composites/Parallel.java`

Класс `Decorators.Repeat`

Граф наследования `Decorators.Repeat` представлен на рисунке Д.28:

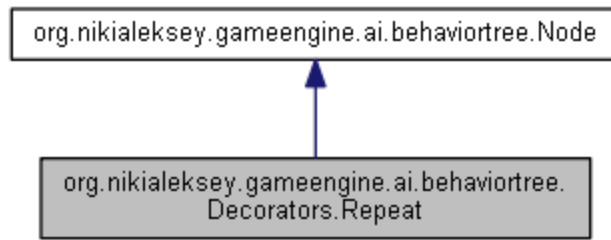


Рисунок Д.28 - Граф наследования Decorators.Repeat

Граф связей класса Decorators.Repeat представлен на рисунке Д.29:

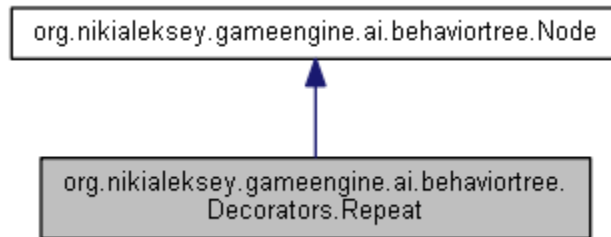


Рисунок Д.30 - Граф связей класса Decorators.Repeat

Открытые члены

- **Repeat** (int repeatCount, **Node** node)
- void **enter** (**Tick** tick)
- void **open** (**Tick** tick)
- **Status** tick (**Tick** tick)
- void **close** (**Tick** tick)
- void **exit** (**Tick** tick)

Подробное описание

Класс представляет декоратор, который передает сигнал на исполнение дочерней вершине заданное количество раз.

Автор:

Alexey Nikitin

Конструктор(ы)

Decorators.Repeat.Repeat (int *repeatCount*, **Node** *node*)

Конструктор.

Аргументы:

- repeatCount - количество передач сигнала на исполнение дочерней вершине
- node - дочерняя вершина

Методы

Status Decorators.Repeat.tick (Tick *tick*)

Передаёт сигнал на исполнение дочерней вершине заданное число раз.

Аргументы:

- *tick* – объект тика

Возвращает:

ERROR, если дочерней вершины нет; последний статус, который вернула дочерняя вершина, если количество повторений было не нулевое; если количество повторений было нулевое, то SUCCESS.

Граф вызовов представлен на рисунке Д.31:

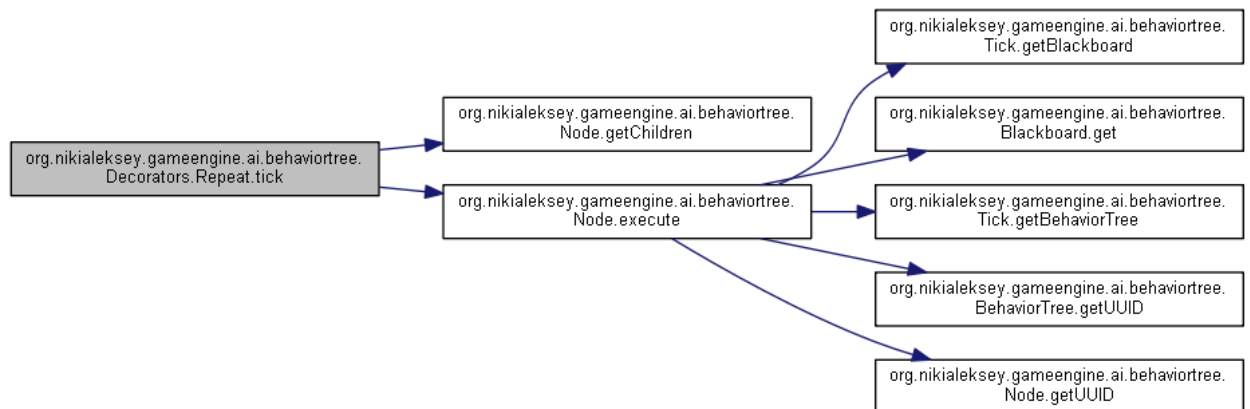


Рисунок Д.31 - Граф вызовов метода Repeat.tick()

Объявления и описания членов класса находятся в файле:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Decorators/Repeat.java`

Класс Composites.Selector

Граф наследования `Composites.Selector` представлен на рисунке Д.32:

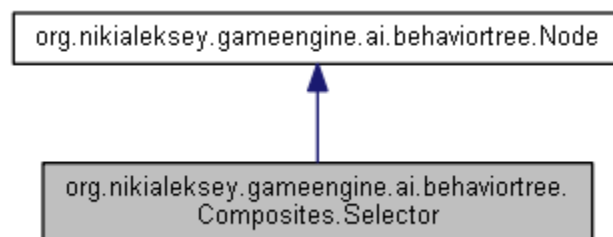


Рисунок Д.32 - Граф наследования Composites.Selector

Граф связей класса `Composites.Selector` представлен на рисунке Д.33:

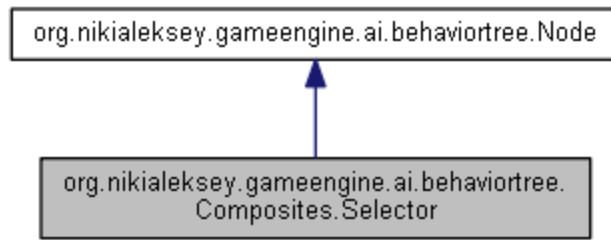


Рисунок Д.33 - Граф связей класса Composites.Selector

Открытые члены

- **Selector** (Node...nodes)
- void **enter** (Tick tick)
- void **open** (Tick tick)
- **Status tick** (Tick tick)
- void **close** (Tick tick)
- void **exit** (Tick tick)

Подробное описание

Класс представляет композит-селектор. Выполняет все свои дочерние вершины по порядку, до тех пор, пока они возвращают результат FAILURE

Автор:

Alexey Nikitin

Конструктор(ы)

Composites.Selector.Selector (Node... *nodes*)

Конструктор.

Аргументы:

- *nodes* - список дочерних вершин

Методы

Status Composites.Selector.tick (Tick *tick*)

Передаёт сигнал на исполнение всем дочерним вершинам, до тех пор пока они возвращают FAILURE

Аргументы:

- *tick* – объект тика

Возвращает:

либо статус той вершины, которая вернула результат, отличный от FAILURE, либо FAILURE

Граф вызовов представлен на рисунке Д.34:



Рисунок Д.34 - Граф вызовов метода Selector.tick()

Объявления и описания членов класса находятся в файле:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Composites/Selector.java`

Класс **Composites.Sequence**

Граф наследования **Composites.Sequence** представлен на рисунке Д.35:

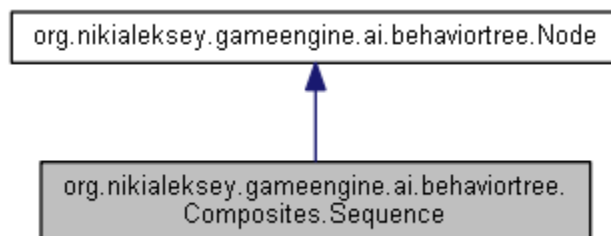


Рисунок Д.35 - Граф наследования **Composites.Sequence**

Граф связей класса **Composites.Sequence** представлен на рисунке Д.36:

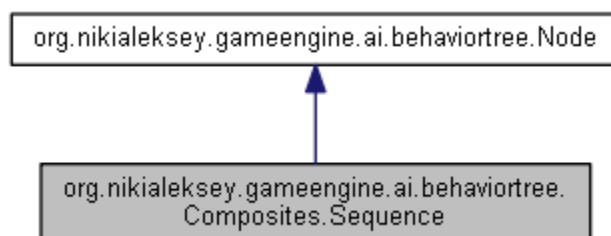


Рисунок Д.36 - Граф связей класса **Composites.Sequence**

Открытые члены

- **Sequence** (Node...nodes)
- **void enter** (Tick tick)
- **void open** (Tick tick)
- **Status tick** (Tick tick)
- **void close** (Tick tick)
- **void exit** (Tick tick)

Подробное описание

Класс представляет композит-последовательность. Выполняет всех своих детей до тех пор, пока они возвращают SUCCESS

Автор:

Alexey Nikitin

Конструктор(ы)

Composites.Sequence.Sequence (Node... *nodes*)

Конструктор.

Аргументы:

- nodes - список дочерних вершин

Методы

Status Composites.Sequence.tick (Tick *tick*)

Передаёт сигнал на исполнение всем своим дочерним вершинам до тех пор, пока они возвращают SUCCESS

Аргументы:

- tick – объект тика

Возвращает:

либо статус дочерней вершины, которая вернула результат, отличный от SUCCESS, либо SUCCESS

Граф вызовов представлен на рисунке Д.37:



Рисунок Д.37 - Граф вызовов метода Sequence.tick()

Объявления и описания членов класса находятся в файле:

- src/org/nikialeksey/gameengine/ai/behaviortree/Composites/Sequence.java

Status Ссылки на перечисление

Открытые атрибуты

- RUNNING
- ERROR
- FAILURE
- SUCCESS

Подробное описание

Класс представляет результаты, которые возвращают вершины после исполнения логики.

Автор:

Alexey Nikitin

Данные класса

Status.ERROR

Ошибка. Результат возвращается, когда произошла ошибка при исполнении логики вершины.

Status.FAILURE

Не успешно. Сигнал возвращается, когда исполнение вершины завершилось не успешно.

Status.RUNNING

Запущено. Результат возвращается, когда вершина не завершила действие.

Status.SUCCESS

Успешно Сигнал возвращается, когда исполнение вершины завершилось успешно.

Документация для этого перечисления сгенерирована из файла:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Status.java`

Класс Tick

Открытые члены

- **Tick** (**BehaviorTree** behaviorTree, **Blackboard** blackboard)
- **Blackboard** getBlackboard ()
- **BehaviorTree** getBehaviorTree ()
- void **enterNode** (**Node** node)
- void **openNode** (**Node** node)
- void **tickNode** (**Node** node)
- void **closeNode** (**Node** node)
- void **exitNode** (**Node** node)

Подробное описание

Класс содержит ссылки на blackboard и behaviorTree. Используется, когда сигнал на исполнение поступает в дерево поведения и пробрасывается дочерним вершинам, чтобы дочерние вершины могли пользоваться blackboard'ом.

Так же удобно здесь писать отладочный код.

Автор:

Alexey Nikitin

Конструктор(ы)

Tick.Tick (**BehaviorTree** *behaviorTree*, **Blackboard** *blackboard*)

Конструктор. Сохраняет ссылки на объект дерева поведения и на общую память.

Аргументы:

- behaviorTree - ссылка на дерево поведения, в котором создали объект Tick
- blackboard - ссылка на общую память

Методы

void **Tick.closeNode** (**Node** *node*)

Исполняется перед закрытием вершины.

Аргументы:

- node - закрываемая вершина

void **Tick.enterNode** (**Node** *node*)

Исполняется при входе в вершину.

Аргументы:

- node - вершина, в которую осуществился вход

`void Tick.exitNode (Node node)`

Исполняется перед выходом из вершины.

Аргументы:

- node - вершина, из которой осуществляется выход

`BehaviorTree Tick.getBehaviorTree ()`

Возвращает ссылку на дерево поведения.

Возвращает:

ссылку на дерево поведения.

Граф вызова функции представлен на рисунке Д.38:

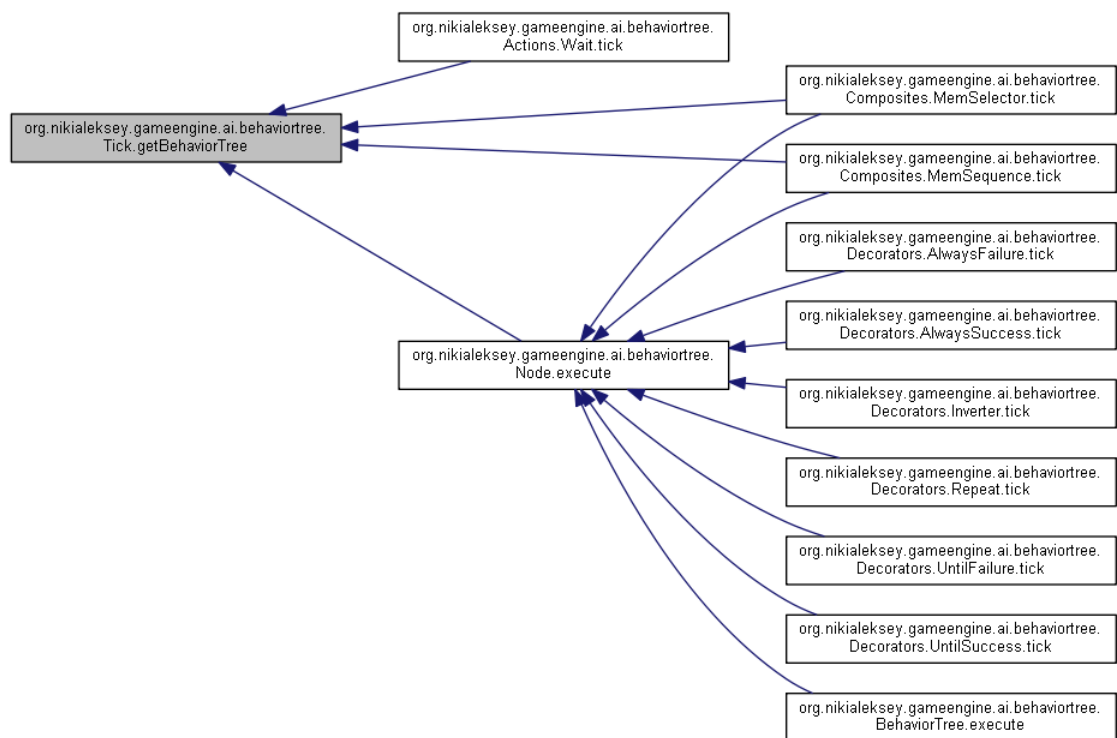


Рисунок Д.38 - Граф вызова метода `Tick.getBehaviorTree()`

`Blackboard Tick.getBlackboard ()`

Возвращает ссылку на объект общей памяти.

Возвращает:

ссылку на объект общей памяти.

Граф вызова функции представлен на рисунке Д.39:

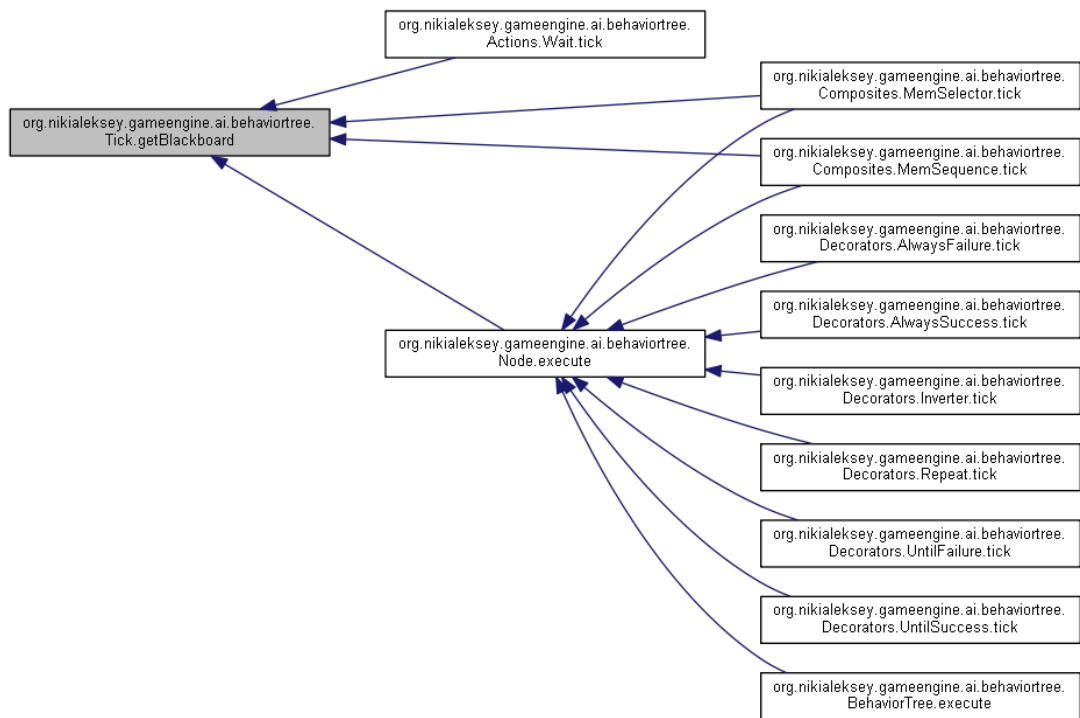


Рисунок Д.39 - Граф вызова метода Tick.getBlackboard()

```
void Tick.openNode (Node node)
```

Исполняется при открытии вершины.

Аргументы:

- node - открываемая вершина

```
void Tick.tickNode (Node node)
```

Исполняется перед выполнением логики вершины.

Аргументы:

- node - вершина, в которую зашли, возможно открыли, и еще не началось выполнение логики

Объявления и описания членов класса находятся в файле:

- src/org/nikialeksey/gameengine/ai/behaviortree/Tick.java

Класс Decorators.UntilFailure

Граф наследования Decorators.UntilFailure представлен на рисунке Д.40:

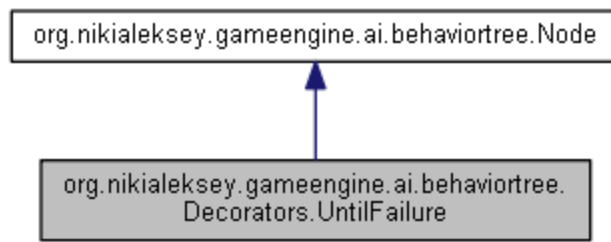


Рисунок Д.40 - Граф наследования Decorators.UntilFailure

Граф связей класса Decorators.UntilFailure представлен на рисунке Д.41:

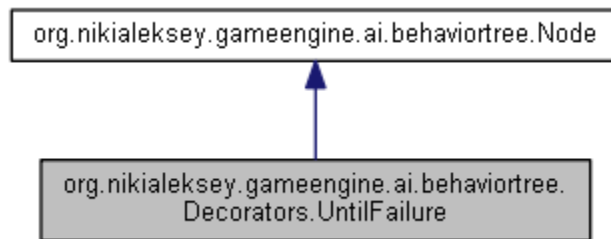


Рисунок Д.41 - Граф связей класса Decorators.UntilFailure

Открытые члены

- **UntilFailure** (Node node)
- void **enter** (Tick tick)
- void **open** (Tick tick)
- **Status** tick (Tick tick)
- void **close** (Tick tick)
- void **exit** (Tick tick)

Подробное описание

Класс представляет декоратор, который передает сигнал на исполнение дочерней вершине до тех пор, пока она не вернет статус FAILURE

Автор:

Alexey Nikitin

Конструктор(ы)

Decorators.UntilFailure.UntilFailure (Node *node*)

Конструктор.

Аргументы:

- *node* - дочерняя вершина

Методы

Status Decorators.UntilFailure.tick (Tick *tick*)

Передаёт сигнал на исполнение дочерней вершине до тех пор, пока она возвращает статус, отличный от FAILURE.

Аргументы:

- *tick* - объект тика

Возвращает:

ERROR, если нет дочерней вершины, FAILURE иначе.

Граф вызовов представлен на рисунке Д.42:

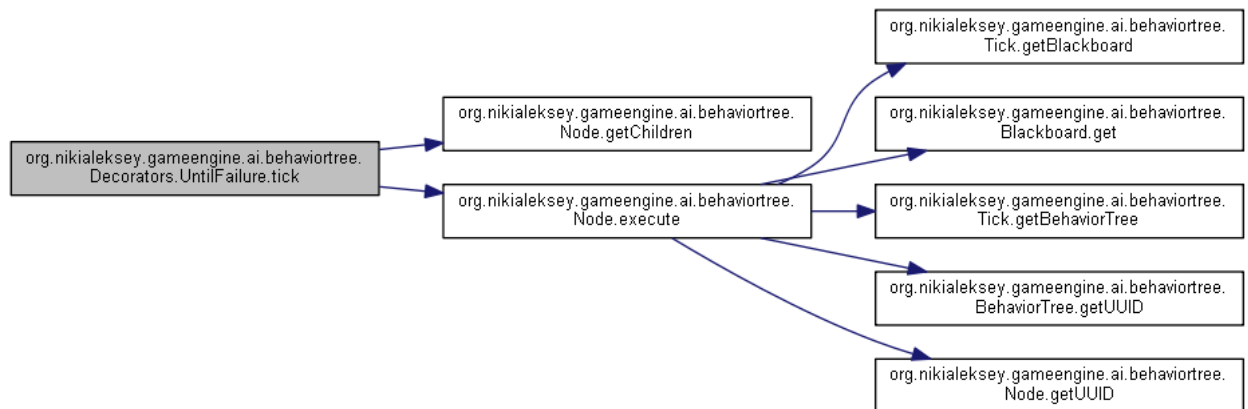


Рисунок Д.42 - Граф вызовов метода UntilFailure.tick()

Объявления и описания членов класса находятся в файле:

- src/org/nikialeksey/gameengine/ai/behaviortree/Decorators/UntilFailure.java

Класс Decorators.UntilSuccess

Граф наследования Decorators.UntilSuccess представлен на рисунке Д.43:

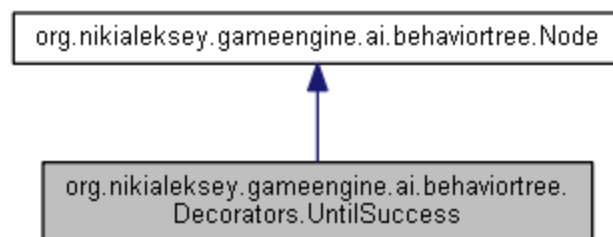


Рисунок Д.43 - Граф наследования Decorators.UntilSuccess

Граф связей класса Decorators.UntilSuccess представлен на рисунке Д.44:

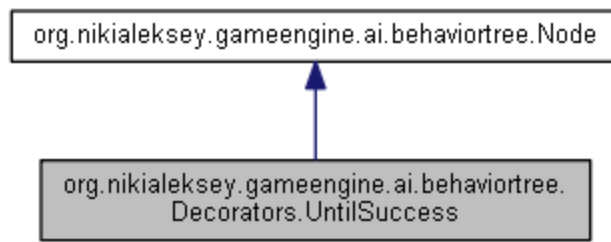


Рисунок Д.44 - Граф связей класса Decorators.UntilSuccess

Открытые члены

- **UntilSuccess** (**Node** node)
- void **enter** (**Tick** tick)
- void **open** (**Tick** tick)
- **Status** tick (**Tick** tick)
- void **close** (**Tick** tick)
- void **exit** (**Tick** tick)

Подробное описание

Класс представляет декоратор, который передает сигнал на исполнение дочерней вершине до тех пор, пока она не вернет статус SUCCESS

Автор:

Alexey Nikitin

Конструктор(ы)

Decorators.UntilSuccess.UntilSuccess (**Node** *node*)

Конструктор.

Аргументы:

- node - дочерняя вершина

Методы

Status Decorators.UntilSuccess.tick (**Tick** *tick*)

Передает сигнал на исполнение дочерней вершине до тех пор, пока она возвращает статус, отличный от SUCCESS.

Аргументы:

- tick - объект тика

Возвращает:

ERROR, если нет дочерней вершины, SUCCESS иначе.

Граф вызовов представлен на рисунке Д.45:

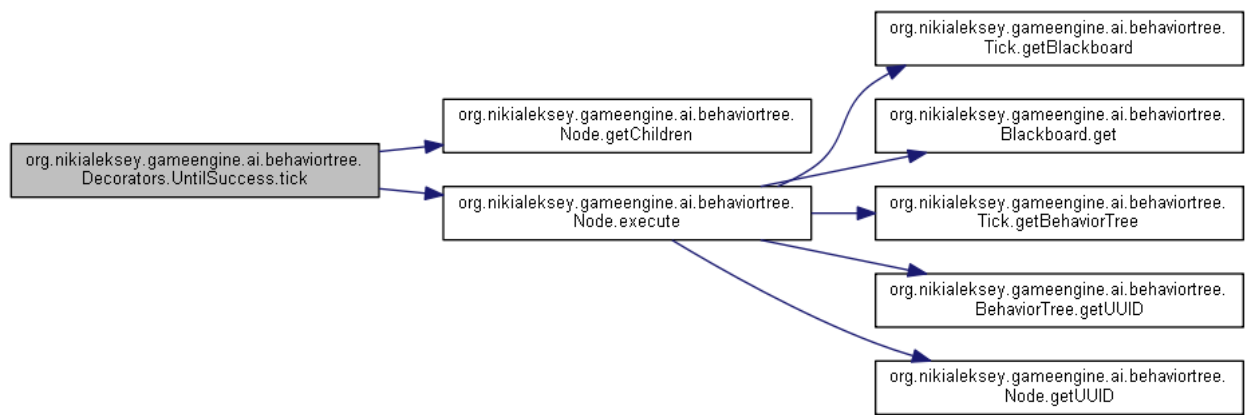


Рисунок Д.45 - Граф вызовов метода UntilSuccess.tick()

Объявления и описания членов класса находятся в файле:

- src/org/nikialeksey/gameengine/ai/behaviortree/Decorators/UntilSuccess.java

Класс Actions.UserAction

Граф наследования Actions.UserAction представлен на рисунке Д.46:

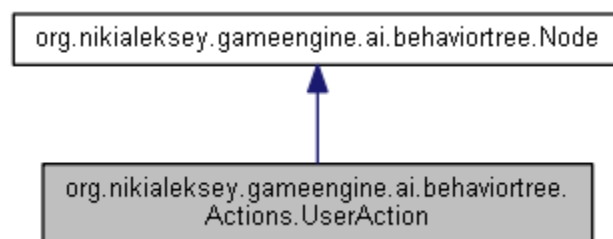


Рисунок Д.46 - Граф наследования Actions.UserAction

Граф связей класса Actions.UserAction представлен на рисунке Д.47:

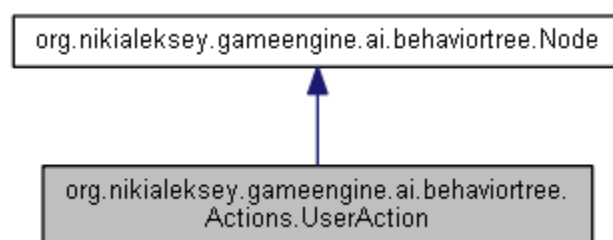


Рисунок Д.47 - Граф связей класса Actions.UserAction

Открытые члены

- **UserAction** (Consumer< **Tick** > action)
- void **enter** (**Tick** tick)
- void **open** (**Tick** tick)
- **Status** tick (**Tick** tick)
- void **close** (**Tick** tick)

- void **exit** (Tick tick)

Подробное описание

Класс представляет лист-действие, определяемое пользователем.

Автор:

Alexey Nikitin

Конструктор(ы)

Actions.UserAction.UserAction (Consumer< Tick > *action*)

Конструктор.

Аргументы:

- *action* - действие, определяемое пользователем

Методы

Status Actions.UserAction.tick (Tick *tick*)

Выполняет пользовательское действие и возвращает SUCCESS.

Аргументы:

- *tick* - объект тика

Возвращает:

SUCCESS

Объявления и описания членов класса находятся в файле:

- src/org/nikialeksey/gameengine/ai/behaviortree/Actions/UserAction.java

Класс Actions.Wait

Граф наследования Actions.Wait представлен на рисунке Д.48:

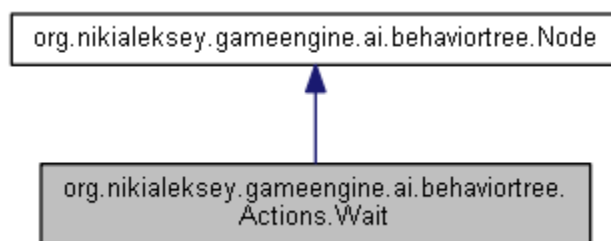


Рисунок Д.48 - Граф наследования Actions.Wait

Граф связей класса Actions.Wait представлен на рисунке Д.49:

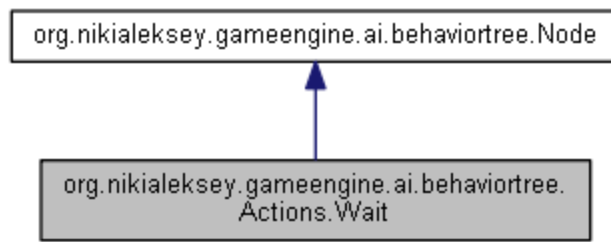


Рисунок Д.49 - Граф связей класса Actions.Wait

Открытые члены

- **Wait** (long milliseconds)
- void **enter** (Tick tick)
- void **open** (Tick tick)
- **Status tick** (Tick tick)
- void **close** (Tick tick)
- void **exit** (Tick tick)

Подробное описание

Класс представляет лист-действие "ожидание".

Автор:

Alexey Nikitin

Конструктор(ы)

Actions.Wait.Wait (long *milliseconds*)

Конструктор.

Аргументы:

- *milliseconds* - количество миллисекунд, которое необходимо подождать

Методы

Status Actions.Wait.tick (Tick *tick*)

Проверяет, сколько времени прошло с первого вызова и если прошло времени больше, чем заданное количество, то вернет SUCCESS.

Аргументы:

- tick - объект тика

Возвращает:

ERROR, если время первого запуска не было записано; RUNNING, если время ожидания еще не прошло; SUCCESS, если время ожидания превышает или совпадает с заданным значением.

Граф вызовов представлен на рисунке Д.50:

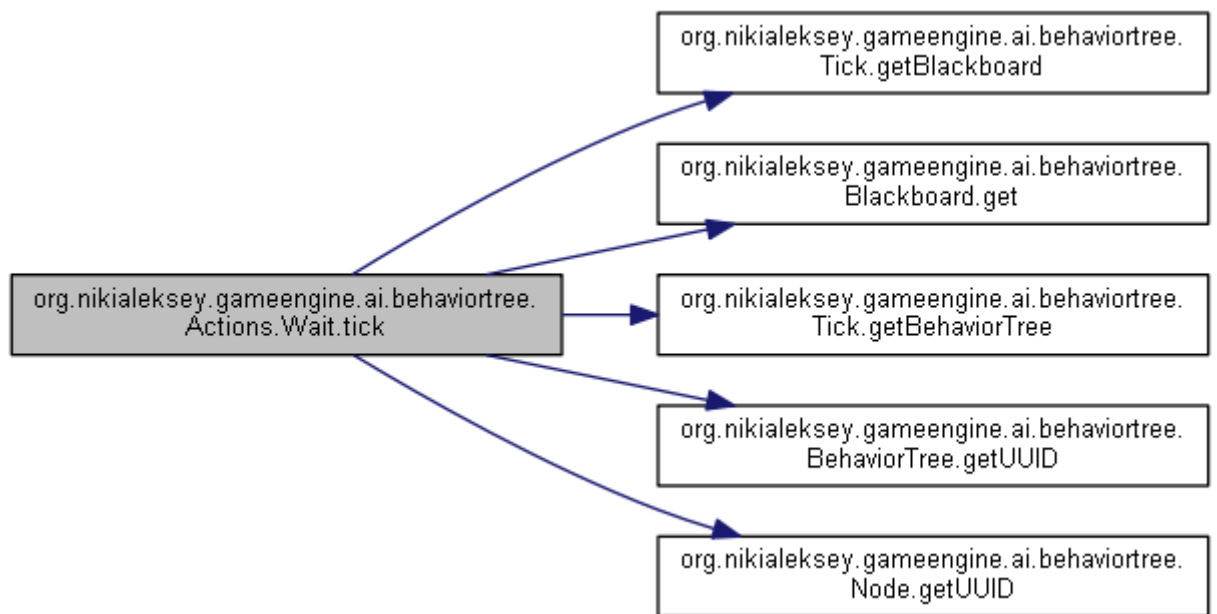


Рисунок Д.50 - Граф вызовов метода `Wait.tick()`

Объявления и описания членов класса находятся в файле:

- `src/org/nikialeksey/gameengine/ai/behaviortree/Actions/Wait.java`

