Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерные технологии»

Б.З. Хасянзянов

Существо *QBeing* для проекта *Электрические Джунгли*

Проектная документация

Проект создан в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2007

Оглавление

Введение 4
1. Правила проекта «Электрические Джунгли» 4
1.1. Обозначения 4
1.2. Понятия и цели 4
1.2.1. Пространство и время 4
1.2.2. Масса и скорость 5
1.2.3. Энергия 5
1.2.4. Информация о внешнем мире 5
1.3. Действия и события 6
1.4. Состязание 7
1.5. Виды игр 7
1.6. Графическое представление игры 8
2. Реализация стратегии существ 9
2.1. Общее описание существа. 9
2.2. Стратегия ведения игры. 9
2.3. Глобальный разум. 9
2.4. Виды существ. 10
2.4.1. Разведчик. 10
2.4.2. Производитель. 10
3. Конечный автомат существа *QBeing* (*A1*) 11
3.1. Описание 11
3.2. Схема связей 12
3.3. Описание состояний 13
3.4. Описание событий 13
3.5. Список выходных воздействий 14
3.6. Список параметров при переходах 15
3.7. Граф переходов 16
4. Запуск программы 16
Заключение 18
Литература 20
Приложение. Исходный текст программы. 21
`BeingEvent.java` 21
`BeingState.java` 21
`EventManager.java` 21
`QBeing.java` 22
`QBeingAutomata.java` 25
`StateMachineContext.java` 37
`StrategyAutomata.java` 38
`StrategyEvent.java` 39

StrategyState.java 40
BehaviourConstants.java 40
BeingType.java 40
BeingUtil.java 41
LocationInfo.java 43
Colony.java 43
ColonyControl.java 46
GlobalStrategy.java 48
UniverseInfo.java 54

Введение

В ходе бакалаврской практики была поставлена задача – создать существо для проекта Электрические Джунгли (http://www.electricjungle.ru). Электрические Джунгли (ЭД) – это состязание, в котором игроки программируют поведение населяющих особый мир виртуальных существ, борющихся за ограниченный ресурс – «энергию». Целью игрока является программирование поведения своих существ таким образом, чтобы его вид как целое максимально преуспел. Разработка должна вестись на языке *Java*.

Для решения данной задачи была использована SWITCH-технология, так как она в полной мере подходит для данного типа задач.

1. Правила проекта «Электрические Джунгли»

1.1. Обозначения

- `X% МЭ` — количество энергии в процентах от максимальной энергии (МЭ) существа;
- `K_someconst(5)` — постоянная, которая определяет значение одного параметра игры (все постоянные определены в исходном тексте игры). В скобках указано текущее значение. Для большинства констант эти значения во время всего конкурса сохранятся, однако возможны и переопределения.

1.2. Понятия и цели

В мире ЭД основная цель — максимальная видовая экспансия. Победившим признается тот игрок, чей вид за отведенное количество ходов добьется максимальной массы – суммарная масса всех особей которого будет наибольшей.

1.2.1. Пространство и время

В ЭД пространство устроено как замкнутое двумерное дискретное поле из клеток, закольцованное по краям (так, что получается тор). Впрочем, в движке поддерживаются и другие топологии и размерности, но победитель будет выявлен на стандартной тороидальной топологии 140x120. Игра состоит из последовательности ходов. За один ход каждому существу дается шанс совершить свое действие. Когда все существа совершат свои действия — начинается следующий ход. В общем случае порядок выполнения действий не определен, разные существа могут совершать действия одновременно.

1.2.2. Масса и скорость

Каждое существо в ЭД обладает двумя базовыми характеристиками: массой и скоростью, а также в каждый момент времени характеризуется энергетическим уровнем. Масса совпадает с максимальной энергией, которой может обладать существо. Если энергетический уровень падает ниже `K_emin(15)`% МЭ — существо умирает и вся его оставшаяся энергия доступна для потребления другими существами. Масса не может быть больше `K_maxmass`(1000) и меньше `K_minmass`(0.1). Скорость не может быть больше `K_maxspeed`(10) и меньше `K_minspeed`(1). Общая масса вида (сумма масс всех живых существ данного вида) определяет количество очков игрока. Масса существа определяет его энергоемкость, энергопотребление за ход и повреждение, наносимое в битве. Скорость определяет максимальное расстояние, на которое может передвигаться существо за один ход. Однако чем больше скорость, тем энергетически дороже передвижение существа. Независимо от скорости существо может получать информацию только о точке, где оно находится, а также о соседних точках.

1.2.3. Энергия

Энергия является основой существования в ЭД. Любое действие (даже просто выживание) требует потребления некоторого количества энергии. В начале игры на поле случайным образом размещаются источники энергии, из которых существа могут черпать энергию (не более 10% МЭ). Энергия в источниках постепенно восполняется. Начальное количество энергии и скорость ее роста случайны и различны для разных источников. На поле имеется `NUM_REGULAR`(130) обычных источников и `NUM_GOLDEN`(3) золотых источников, производящих гораздо больше энергии. Кроме этого, в точке, где рождается первое существо каждого игрока даётся `BORN_BONUS`(100) единиц энергии, и за каждый ход прирастает `BORN_BONUS_GROWTH`(1) единиц энергии. Хотя в различных играх энергия и прирост по-разному распределены на поле, суммарная энергия и ее прирост всегда одинаковы.

1.2.4. Информация о внешнем мире

Существо может получить информацию о внешнем мире при помощи API `BeingInterface` и `PointInfo`. Доступна следующая информация:

- количество энергии самого существа;

- владелец и масса любого существа, находящегося достаточно близко (на той же клетке или на примыкающих клетках);
- количество энергии, скорость ее прироста и максимальная емкость в клетке;
- список всех объектов в данной точке (в частности, других существ);
- суммарная масса всех живых существ в данной точке.

1.3. Действия и события

За один ход существо может совершить одно действие, а также каждое существо получает оповещения о происходящих с ним событиях. Конкретный порядок выполнения действий не специфицирован. Доступны следующие действия:

- `ACTION_MOVE_TO` — перейти в другую клетку. Доступность клетки определяется скоростью существа. Стоит `K_movecost`(1)% скорости.
- `ACTION_EAT` — потребить энергию. Количество энергии, которую можно потребить за ход, но общая энергия не может превышать энергоемкости (массы) и потребленная энергия не может быть больше `K_bite`(10)% МЭ. Бесплатно.
- `ACTION_GIVE` — передать энергию другому существу. Бесплатно. Существа должны быть на одной и той же клетке.
- `ACTION_ATTACK` — Атаковать другое существо, нанося повреждение в `K_fight`(20)% МЭ. При этом теряется `K_fightcost`(1)% собственной МЭ + `K_retaliate`(5)% МЭ атакуемого существа.
- `ACTION_BORN` — породить другое существо, возможно с немного отличающимися (не более чем на `K_minbornvariation`(0.8)/`K_maxbornvariation`(1.2)) массой и скоростью (существо с массой 100 и скоростью 2 может порождать существа с массой от 80 до 120 и скоростью от 1.6 до 2.4). Энергия при этом делится пополам. Рождение возможно только тогда, когда у существа достаточно энергии – не менее `K_toborn`(80%) МЭ. Акт рождения стоит `K_borncost`(20)% МЭ. При этом существу можно передать "генокод" — произвольный *Java* объект.
- `ACTION_MOVE_ATTACK` — совмещено двигаться и атаковать, при этом наносится меньше `K_fightmovepenalty` (0.75) повреждений. Стоимость совпадает с суммарной стоимостью атаки и передвижения.

Также существо получает оповещения о следующих событиях:

- `BEING_BORN` — первое оповещение, которую существо получает после рождения. Можно инициализировать различные параметры, специфичные для данного существа.
- `BEING_DEAD` — последнее оповещение, передается после смерти существа.
- `BEING_ATTACKED` — существо атаковано кем-то, идентификатор атакующего передается как параметр.
- `BEING_ENERGY_GIVEN` — Кто-то передал нам энергию.

1.4. Состязание

Состязание начинается с помещения одного существа каждого игрока в случайную точку пространства, при этом один раз вызывается метод `reinit()`, информирующий о текущих условиях игры и позволяющий заново инициализировать статические переменные. Масса и энергия изначального существа определяется самим игроком и может быть произвольной. Параметры первого существа определяются возвращаемым значением метода `Being.getParams()`. Параметры всех остальных существ определяются параметрами действия `ACTION_BORN`. После этого у каждого существа есть возможность попытаться добиться победы в заданных условиях, как описано далее.

1.5. Виды игр

1. **Блицкриг** — *`SINGLE`*

Одному виду существ дается весь мир. Цель — добиться наивысшего рейтинга путем максимально эффективного использования ресурсов и быстрой разведки за данный временной интервал (200 ходов).

2. **Дуэль** — *`DUEL`*

Главный вид состязания в Электрических Джунглях. Два конкурирующих вида существ сражаются на протяжении 1000 ходов.

3. **Джунгли** — *`JUNGLE`*

До восьми конкурирующих видов сражаются на одном поле 2000 ходов.

1.6. Графическое представление игры

Поле игры приведено на рис. 1.

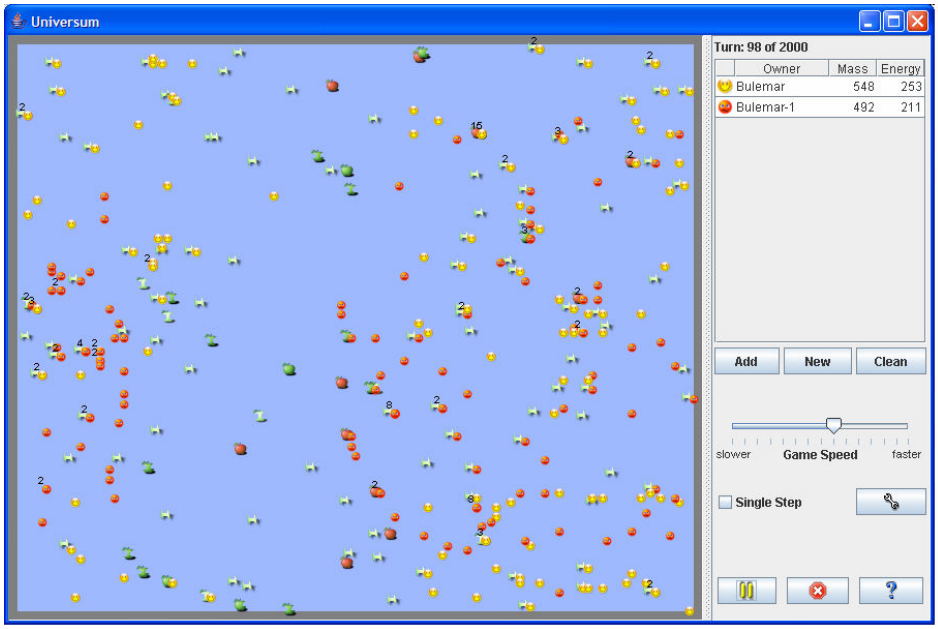


Рис. 1. Поле игры

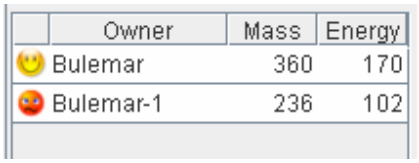
Слева на рис. 1:

 и  – животные из соответствующих команд на игровом поле;

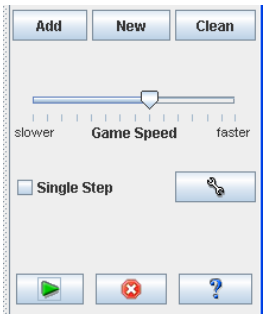
 – источники энергии;  – богатые источники энергии.

Справа на рис. 1:

 – номер текущего хода;



– список существ, участвующих в игре (а так же информация о суммарной массе и энергии существ);



– элементы управления и настройки игры.

2. Реализация стратегии существ

2.1. Общее описание существа

Для проекта Электрические Джунгли было написано существо, которое было названо *QBeing*. Существо реализует наиболее универсальный алгоритм поведения, приемлемый для всех трех видов игр, реализованных в Электронных Джунглях. Экспериментальным путем было получено, что данный алгоритм показывает наилучшие результаты, если применен для существ с массой четырех и скоростью три.

2.2. Стратегия ведения игры

Стратегия существа *QBeing* совмещает в себе алгоритм быстрой разведки территории с целью наиболее раннего захвата ресурсов, алгоритм эффективного распределения ресурсов между особями с целью наиболее быстрого размножения и алгоритм защиты источников энергии от захвата вражескими существами.

Для реализации этой стратегии были разработаны модели поведения для двух видов существ: «Разведчиков» и «Производителей». Задачей «Разведчиков» является быстрая разведка территории и захват новых источников энергии. Основными задачами «Производителей» являются эффективное распределение энергии источника между собой с целью воспроизведения новых существ и защита от нападения врага. Для координации действий «Разведчиков» и «Производителей» разработан класс, служащий одновременно «Глобальным хранилищем» информации об игровом поле и «Глобальным разумом» всей популяции.

2.3. Глобальный разум

Чтобы наиболее эффективно развиваться как вид, существа должны передавать друг другу всю необходимую информацию (расположение источников энергии, разведанные участки, расположение вражеских существ и т.п.). Для этого был создан класс *GlobalStrategy*, основной целью которого был сбор информации об игровом поле. В этом классе реализована логика стратегии поведения существ как целого вида. Этот класс, располагая информацией о «мире» в целом, принимает глобальные решения, затрагивающие целые группы существ. Он координирует разведку, распределяя неразведанные участки между разведчиками. Он же принимает решение об окончании разведки, когда исследована большая часть территории и дает указание разведчикам отправиться к ближайшему наиболее богатому источнику энергии

чтобы присоединиться к колонии. При получении сигнала о нападении врага от какого-либо существа, *GlobalStrategy* решает, нужно ли привлечь существ, находящихся вблизи, к атаке. Если помощь может потребоваться, выбирает группу существ с достаточным количеством энергии и направляет их в атаку. Если энергия существа достигает критически низкой отметки, *GlobalStrategy* направит это существо к ближайшему известному источнику энергии для подпитки.

2.4. Виды существ

2.4.1. Разведчик

Главной целью разведчика является сбор информации о расположении источников энергии на карте. Алгоритм разведки разработан следующим образом. Каждый разведчик проверяет, может ли он за один шаг переместиться на неисследованную позицию карты и, если такие позиции есть, опрашивает «Глобальный Разум» не претендует ли уже какой-либо другой разведчик на исследование этих позиций. Если остаются участки, на которые ни один разведчик не претендует, то существо оповещает «Глобальной Разум», что позиция «занята», и движется к ней. Если не остается неразведанных участков, которые доступны за один шаг, совершается шаг в сторону самого ближнего неразведанного и «свободного» от заявок других разведчиков фрагмента карты.

Как только разведчик находит новый источник энергии, он становится «Производителем» и на этом месте основывает новую колонию. Колония – это группа существ, питающихся от одного источника энергии и распределяющих эту энергию между собой с целью эффективного размножения.

2.4.2. Производитель

Существо, являющееся «Производителем», участвует в производстве новых существ. Группа «Производителей», находящихся в одной точке с источником энергии образует колонию. Задачами колонии «Производителей» является распределение ресурсов между членами колонии и защита колонии от захвата вражескими существами.

Для эффективного распределения энергии среди членов колонии применяется следующий алгоритм. В колонии выделяются наиболее сильные члены. Именно они будут поглощать энергию источника, и заниматься производством существ. Остальные члены колонии будут получать энергию только в том случае, если уровень их энергии достиг критически низкой отметки и существам грозит смерть. Каждый член колонии, не

участвующий в производстве новых существ, старается отдать излишки энергии самым сильным членам. За счет этого достигается большая скорость размножения существ.

Если энергия источника иссякла, и карта еще не разведана, часть «Производителей» превращается в «Разведчиков» и продолжает исследование карты с целью открытия новых источников энергии и создания новых колоний.

Нападение вражеского существа на любого члена колонии расценивается колонией как попытка захвата источника энергии. В этом случае производится оценка сил соперника, и, если сил колонии достаточно для того, чтобы отбить атаку своими силами, то все члены колонии с достаточным уровнем энергии атакуют врага. Если силы врага превосходят силы колонии, то существа, находящиеся вблизи колонии также призываются на помощь.

3. Конечный автомат существа *QBeing* (*A1*)

3.1. Описание

Чтобы выжить, каждое существо должно адекватно реагировать на изменение окружающей обстановки, эффективно обрабатывая как можно больше поступающей извне информации. Автомат *A1*, реализующий логику поведения существа, получает информацию (события) из трех источников. Источники информации автомата перечислены в табл. 1.

Таблица 1

Обозначение	Описание
Engine (*p1*)	Базовый движок игры, предоставляющий такие события, как: • Рождение существа (*BEING_BORN*); • Смерть существа (*BEING_DEAD*); • Нападение вражеского существа (*BEING_ATTACKED*); • Передача энергии от другого существа (*ENERGY_GIVEN*).
GlobalStrategy (*p2*)	Объект, реализующий стратегию всего вида существ. Этот объект содержит в себе всю информацию об исследованном пространстве и реализует алгоритм поведения всего вида. Поэтому может являться источником событий, приводящих к смене тактики целой группы существ.
Colony (*p3*)	Объект, реализующий стратегию поведения существ одной колонии. Этот объект занимается эффективным распределением ресурсов источника энергии в целях наиболее эффективного размножения группы существ.

Поведение каждого существа полностью задается конечным автоматом *A1*, реализованным в классе *QBeingAutomata*.

3.2. Схема связей

Схема связей автомата *A1* изображена на рис. 2. Для построения диаграммы было использовано инструментальное средство *UniMod*.

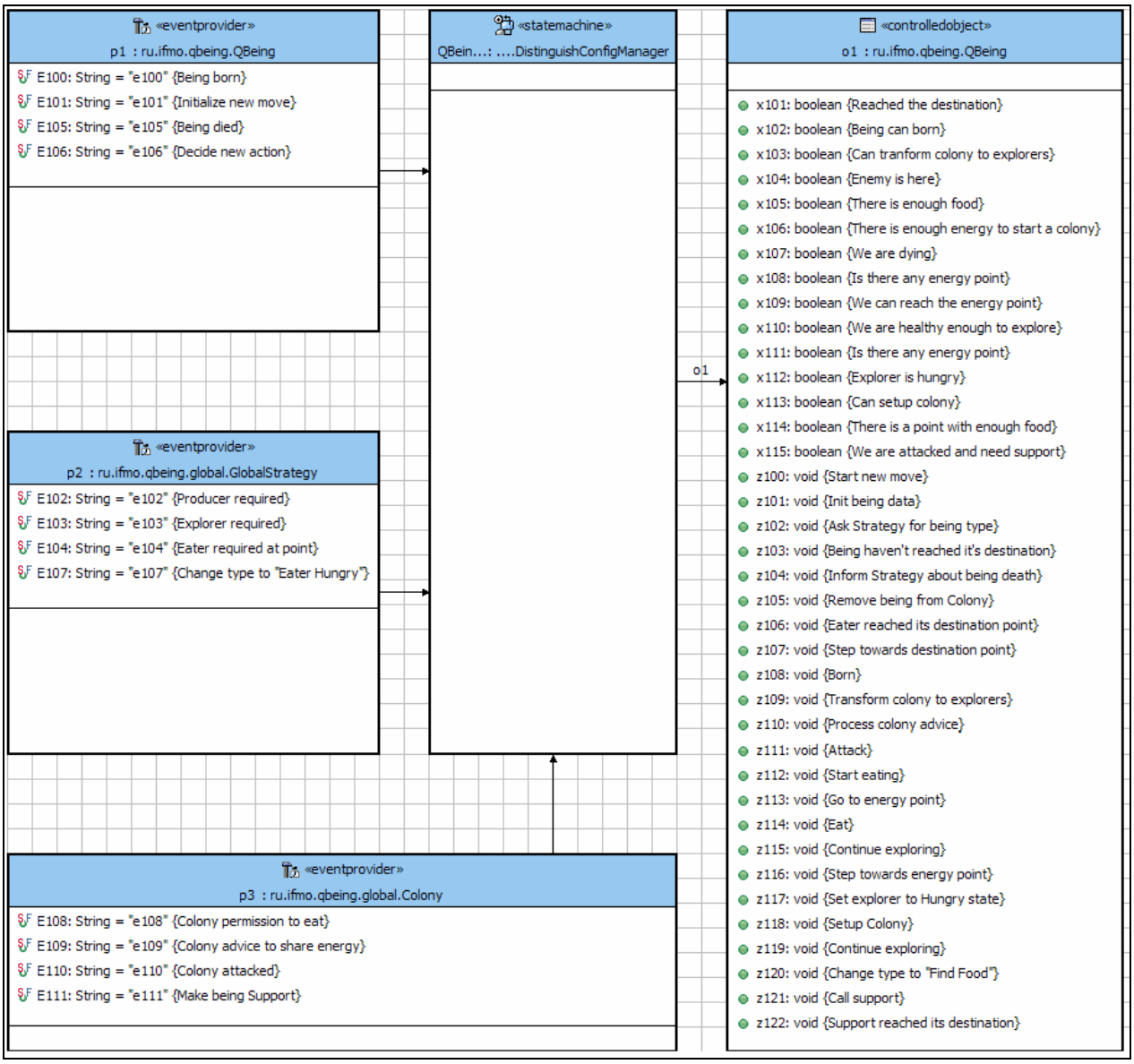


Рис. 2. Схема связей автомата *A1*

3.3. Описание состояний

Состояния автомата *A1* приведены в табл. 2.

Таблица 2

Обозначение	Описание
S_BORN	«Рождение». Начальное состояние существа. Существо ожидание присвоения одного из типов («Разведчик», «Производитель»). Присвоение типа происходит при получении указания от *GlobalStrategy*.
S_EXPLORING	«Разведка». Существо исследует неизвестные участки карты.
S_EXPLORING_HUNGRY	«Разведчик» в поисках энергии для подпитки своих аккумуляторов.
S_PRODUCER	«Производитель»
S_EATER_MOVING_TO_POINT	«Производитель» двигающийся к источнику энергии.
S_EATER_FIND_FEED_PLACE	«Производитель» в поисках энергии для подпитки своих аккумуляторов.
S_SUPPORT	«Поддержка». Существо перемещается к участку, атакованному врагом.

3.4. Описание событий

События, поступающие от источника событий *p1,* движка игры (*Engine*), приведены в табл. 3.

Таблица 3

Обозначение	Описание
E100	Рождение нового существа.
E101	Происходит инициализация перед очередным ходом.
E105	Существо погибло.
E106	Происходит расчет хода.

События, поступающие от источника событий *p2,* стратегии всего вида существ (*GlobalStrategy*), приведены в табл. 4.

Таблица 4

Обозначение	Описание
E102	Требуется «Производитель».
E103	Требуется «Разведчик».
E104	Враг атакует. Требуется поддержка.
E107	Производитель направляется к другому источнику энергии.

События, поступающие от источника событий *p3,* колонии существ (*Colony*), приведены в табл. 5.

Таблица 5

Обозначение	Описание
E108	Колония рекомендует поглотить часть энергии источника.
E109	Колония рекомендует передать излишки энергии другому существу.
E110	Колония атакована врагом.
E111	Существо направляется на битву с врагом (в качестве «Поддержки»).

3.5. Список выходных воздействий

Выходные воздействия клиентского объекта управления (*o1*) приведены в табл. 6.

Таблица 6

Обозначение	Описание
z100	Начать ход.
z101	Инициализация.
z102	Запрос к *GlobalSrategy* о типе существа.
z103	Обработка ситуации, когда существо не достигло пункта назначения.
z104	Обработка события о гибели существа.
z105	Существо покидает колонию.
z106	Существо достигло пункта назначения.
z107	Переместиться в направлении пункта назначения.
z108	Рождение существа.
z109	Расформировать колонию (превратить часть существ колонии в «Разведчиков»).
z110	Расчет действия, направленного на благо колонии в целом.
z111	Атака.
z112	Присоединиться к колонии.
z113	Направиться к источнику энергии.

z114	Поглотить часть энергии источника.
z115	Подзарядка аккумуляторов закончена. Продолжить разведку.
z116	Переместиться в направлении источника энергии.
z117	Направить «Разведчика» на подпитку аккумуляторов.
z118	Превратить «Разведчика», нашедшего источник энергии в «Производителя» и организовать на месте источника новую колонию.
z119	Продолжить разведку территории.
z120	Направить «Производителя» на подпитку аккумуляторов.
z121	Вызвать подмогу.
z122	Существо, направленное на подмогу, достигло точки назначения.

3.6. Список параметров при переходах

Параметры объекта управления существа (*o1*) приведены в табл. 7.

Таблица 7

Обозначение	Описание
x101	Существо достигло точки назначения.
x102	Существо обладает достаточным количеством энергии для рождения нового существа.
x103	Следует ли трансформировать часть существ колонии в «Разведчиков».
x104	Находится ли вражеское существо в непосредственной близости.
x105	Достаточно ли энергии у источника для питания существа.
x106	Достаточно ли энергии у источника для организации новой колонии.
x107	Уровень энергии существа достиг критической отметки.
x108	Имеется ли какой-либо источник энергии, достаточный для подпитки аккумуляторов, в непосредственной близости от существа. (С учетом потребления энергии другими существами колонии).
x109	Хватит ли у существа энергии для того, чтобы достигнуть источника без дополнительной подпитки аккумуляторов.
x110	Достаточно ли у существа энергии для продолжения разведки.
x111	Имеется ли какой-либо источник энергии, достаточный для подпитки аккумуляторов, в непосредственной близости от существа. (Без учета потребления энергии другими существами колонии).
x112	Запас энергии у «Разведчика» подходит к концу.
x113	Возможно ли организовать колонию у ближайшего источника энергии.
x114	Разведка закончена.
x115	Колония атакована врагом. Нужна поддержка.

3.7. Граф переходов

Граф переходов автомата *A1* изображен на рис. 3. Для построения диаграммы было использовано инструментальное средство *UniMod*.

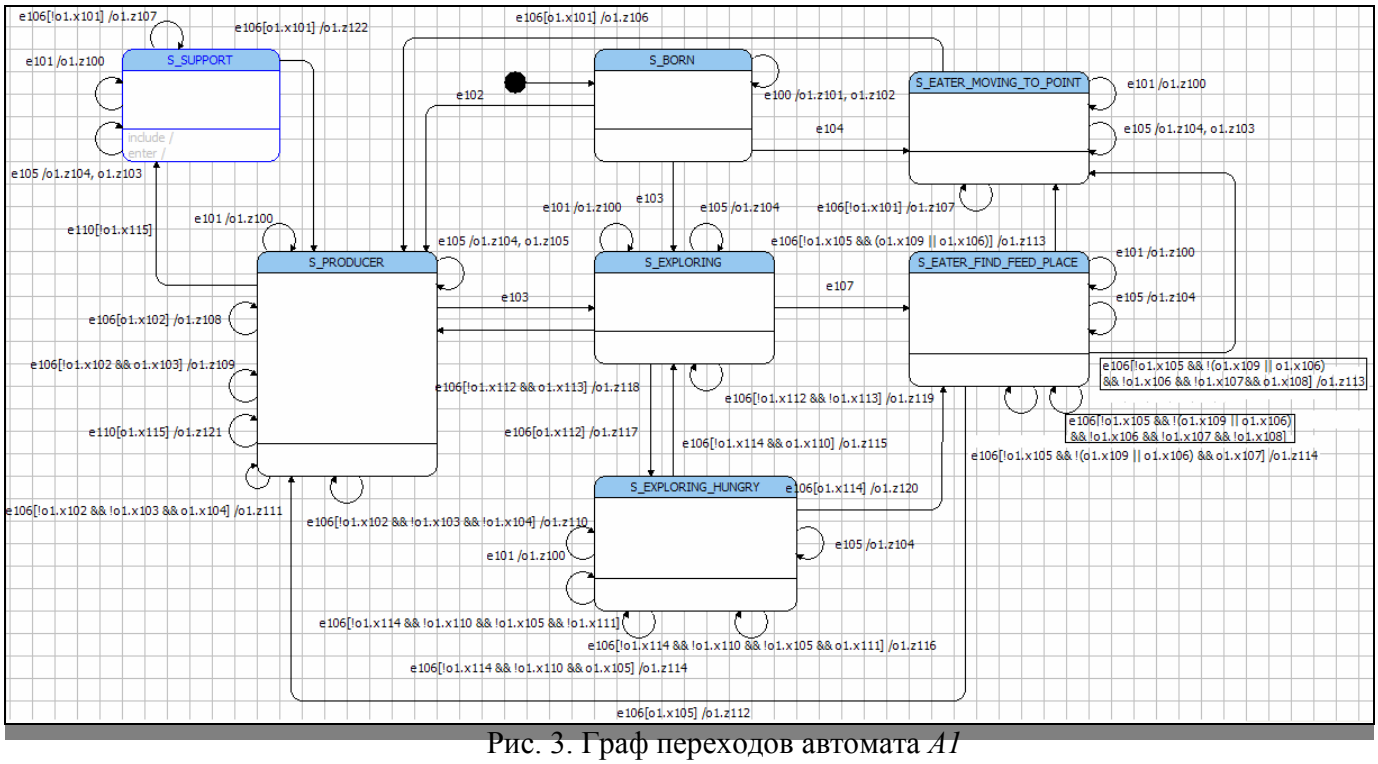


Рис. 3. Граф переходов автомата *A1*

Для реализации автомата *UniMod* использовать было нельзя, так как правила игры запрещают использовать внешние библиотеки.

4. Запуск программы

Для запуска скомпилированной программы необходимо:

1) С сайта Электрических джунглей (http://www.electricjungle.ru/) загрузить Java-архив `ejungle_distr.jar` с исполняемым кодом игры.

2) С сайта проекта скачать Java-архив `QBeing3.jar`, содержащий программу, реализующую алгоритм существа.

3) Запустить игру при помощи команды[1]:

```
java -jar ejungle_distr.jar
```

4) В открывшемся главном окне приложения (рис. 4.) нажать клавишу «`Add`» чтобы добавить существо в игру:

[1] Для запуска необходимо, чтобы на компьютере была установлена *Java* версии 1.5 или более поздней.

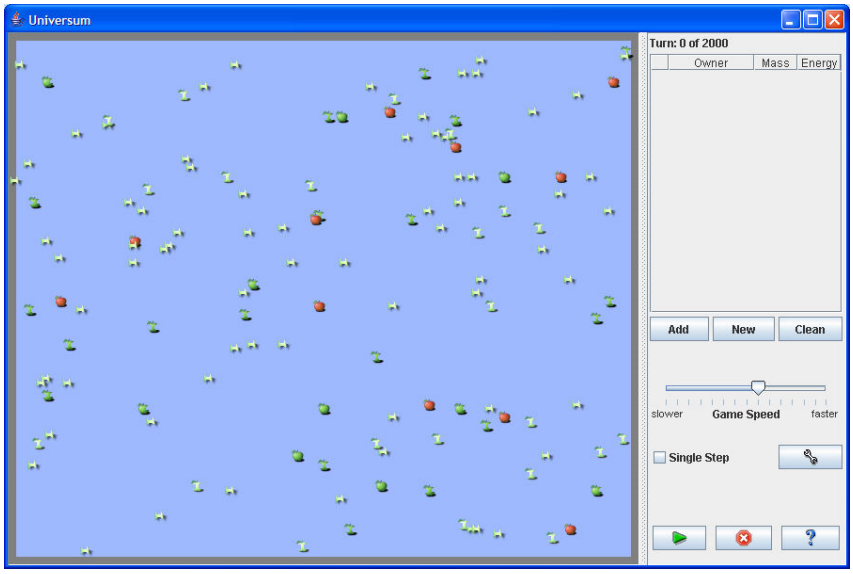


Рис.4. Главное окно приложения

5) В появившемся диалоге (рис. 5) указать путь к Java-архиву `QBeing3.jar`.

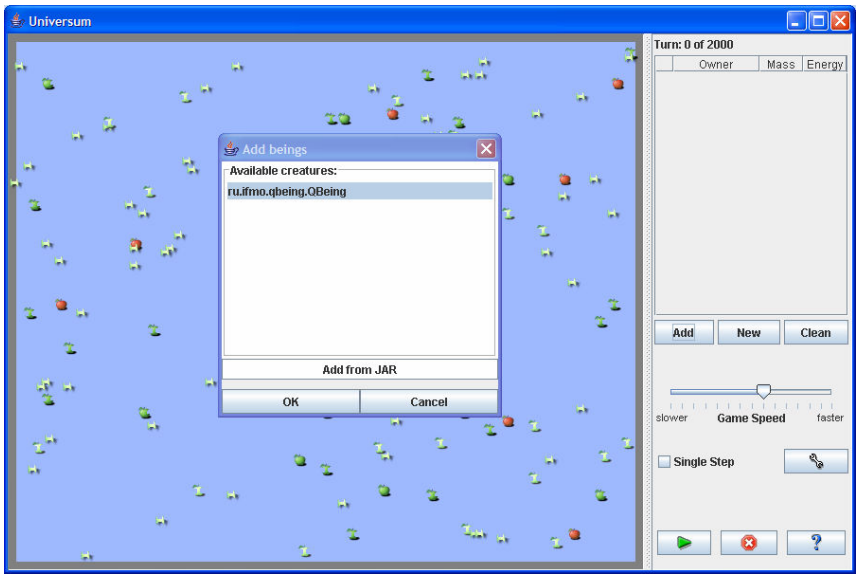


Рис.5. Диалог добавления существа

6) Нажать кнопку  для запуска игры.

Заключение

При разработке существа основное внимание уделялось реализации наиболее универсального алгоритма для двух видов состязаний – одиночных соревнований и дуэлей. Поэтому существа добились равного успеха в обоих конкурсах.

Существо участвует в конкурсе под названием «*QBeing*» (автор «*Bulemar*») и заняло следующие места в рейтинге:

- 52 место в рейтинге «Сильнейших популяций по результатам дуэлей»[2] (рис. 6).

Сильнейшие популяции по результатам дуэлей

Рейтинг	Участник	Существо	Побед
Высшая лига			
1	Executer	universum.beings.HomoSapiensPack.HomoSapiens	163
2	Enotus	universum.beings.Enot1	160
3	riite	universum.beings.Cheburator	152
4	trooper	universum.beings.PypoksMonkey	148
5	Putnik	universum.beings.Life	145
6	trooper	universum.beings.Monkey	143
7	SergE	com.serge.creatures3.Biber	137
8	tivv	com.gtechua.being.b1.Aggressor	135
9	SergE	com.serge.creatures8.Biber	131
10	wp	universum.beings.ThirdBeing	113
1 лига			
50	Goldman	yargroup.EasyLight	117
51	Dyakonov	universum.beings.Sapsan	113
52	Bulemar	ru.ifmo.qbeing.QBeing	112
53	Dyakonov	universum.beings.Voron	103

Рис.6. Результаты в рейтинге «Сильнейших популяций по результатам дуэлей»

В конкурсе «Дуэль» существо вышло в финальный этап соревнований.

[2] Данные рейтинга верны на момент написания работы. Подробнее с рейтингом можно ознакомиться на сайте: http://www.electricjungle.ru в разделе «Рейтинг».

- 24 место в рейтинге «Достижения в одиночных играх» (рис. 7).

Достижения в Одиночных играх

Рейтинг	Участник	Существо	Результат
1	Vovka	universum.beings.vovka.BaseBeing	8822
2	kammerer	ru.kammerer.being.snake.Snake	8618
3	Enotus	universum.beings.Enot1	8613
4	kammerer	ru.kammerer.being.snake.Snake2	8410
5	Griff	universum.beings.griff.BeingA	8284
6	Executer	universum.beings.HomoSapiensPack.HomoSapiens	8270
7	sergonaft	universum.beings.SerZhenCreature	7705
8	giz	ru.ifmo.votinov.ConquerorBeing	7646
9	sergonaft	universum.beings.SerZhenSimplest	7477
10	forlik	universum.beings.forlik.Wasp	7363
22	Antidote	universum.beings.Scout	5725
23	Afonin	universum.beings.robin_bobin.RobinBobin	5700
24	Bulemar	ru.ifmo.qbeing.QBeing	5516
25	e13	piglets.gamma.Piglet	5352

Рис.7. Результаты в рейтинге «Достижения в одиночных играх»

Разработанный интеллект достаточно эффективен при разведке и размножении существ. При этом под эффективностью разведки понимается, что полная разведка территории заканчивается примерно на 50-ых – 70-ых ходах (Результаты справедливы для одиночной игры, когда существам одного вида предоставлено все игровое поле). Быстрая разведка важна в соревнованиях «Дуэль», так как позволяет на ранних стадиях игры захватить как можно больше ресурсов раньше соперника. Под эффективностью размножения понимается возможность воспроизвести как можно большее количество существ, используя как можно меньше энергии.

Полученный в соревновании на данный момент результат можно улучшить, усовершенствовав алгоритм нападения, реализовав не только защиту собственных источников энергии, но и централизованную атаку на наиболее богатые энергией вражеские источники.

Литература

1. *Электрические джунгли*. http://is.ifmo.ru/elejungle/

2. *Шалыто А.А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука. 1998. http://is.ifmo.ru/books/switch/1

3. *Шалыто А.А.* Новая инициатива в программировании. Движение за открытую проектную документацию // Мир ПК. 2003. № 9, с.52–56. http://is.ifmo.ru/works/open_doc

4. *Паньгин* А.А. Электроджунгли. Новый конкурс по программированию на Java // Компьютерные инструменты в образовании. 2006. № 2, с.85–87. http://is.ifmo.ru/elejungle/_CIE-EJ.pdf

Приложение. Исходный текст программы

BeingEvent.java

```
package ru.ifmo.qbeing;

public interface BeingEvent {
    // Being Events
    int E100 = 100; // Being born
    int E101 = 101; // initialize before new move
    int E102 = 102; // set being type to "Producer"
    int E103 = 103; // set being type to "Explorer searching"
    int E104 = 104; // set being type to "Support"
    int E105 = 105; // being dead
    int E106 = 106; // calculate move
    int E107 = 107; // set being type to "Eater find food"
    int E108 = 108; // colony permission to eat
    int E109 = 109; // colony advice to share energy
    int E110 = 110; // Colony attacked
    int E111 = 111; // Make being Support
}
```

BeingState.java

```
package ru.ifmo.qbeing;

public interface BeingState {
    // Being states
    int S_BORN = 10;
    int S_PRODUCER = 11;
    int S_EATER_MOVING_TO_POINT = 12;
    int S_EATER_FIND_FEED_PLACE = 13;
    int S_EXPLORING = 14;
    int S_EXPLORING_HUNGRY = 15;
    int S_SUPPORT = 16;
}
```

EventManager.java

```
package ru.ifmo.qbeing;

import ru.ifmo.qbeing.global.GlobalStrategy;

public class EventManager {
    public static void handleBeingEvent(int event, QBeing dest,
StateMachineContext context) {
        dest.getAutomata().processBeingEvent(event, context);
    }
    public static void handleStrategyEvent(int event, StateMachineContext
context) {
        GlobalStrategy.instance().getAutomata().processStrategyEvent(event,
context);
    }
}
```

QBeing.java

```
package ru.ifmo.qbeing;

import ru.ifmo.qbeing.common.BehaviourConstants;
import ru.ifmo.qbeing.common.BeingType;
import ru.ifmo.qbeing.global.GlobalStrategy;
import ru.ifmo.qbeing.global.UniverseInfo;
import universum.bi.*;

public class QBeing implements Being, Comparable<QBeing> {

    public static int gcount;
    public static int gdead;

    private QBeing parent;
    private float mass;
    private float speed;
    private float energy;
    private int id;

    private Location location;
    private Integer[] others;

    private QBeingAutomata automata;
    private Location destination;
    private Event action;

    public QBeing() {
        others = new Integer[1];
        automata = new QBeingAutomata(this);
    }

    public float getMass() {
        return mass;
    }

    public void setMass(float mass) {
        this.mass = mass;
    }

    public void reinit(UserGameInfo info) {
        GlobalStrategy.initInstance();
        UniverseInfo.initInstance();
        GlobalStrategy.instance().initGameInfo(info);

        gcount = 0;
        gdead = 0;
    }

    public String getName() {
        return "QBeing";
    }

    public String getOwnerName() {
        return "Bulemar";
    }
```

```
    public BeingParams getParams() {
        return new BeingParams(BehaviourConstants.MAX_MASS,
BehaviourConstants.SPEED);
    }

    public synchronized void processEvent(BeingInterface bi, Event e) {
        StateMachineContext ctx;
        switch (e.kind()) {
            case BEING_BORN:
                gcount++;
                ctx = new StateMachineContext();
                ctx.setBeingInterface(bi);
                ctx.setEvent(e);

                EventManager.handleBeingEvent(BeingEvent.E100, this, ctx);
                break;
            case BEING_DEAD:
                gcount--;
                gdead++;

                ctx = new StateMachineContext();
                ctx.setBeingInterface(bi);
                EventManager.handleBeingEvent(BeingEvent.E105, this, ctx);
                bi.log(this, "dead: " + gdead);
                break;
            case BEING_ATTACKED:
                ctx = new StateMachineContext();
                ctx.setBeingInterface(bi);
                ctx.setSender(this);
                float forces = getEnemyForces(bi);
                ctx.put("forces", forces);
                if (getState() == BeingState.S_PRODUCER)
                    EventManager.handleBeingEvent(BeingEvent.E110, this, ctx);
        }
    }

    public Event makeTurn(BeingInterface bi) {
        StateMachineContext machineContext = new StateMachineContext();
        machineContext.setBeingInterface(bi);
        machineContext.setBeing(this);
        EventManager.handleBeingEvent(BeingEvent.E101, this, machineContext);

        return action;
    }

    public QBeing getParent() {
        return parent;
    }

    public void setParent(QBeing parent) {
        this.parent = parent;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getId() {
```

```
    return id;
}

public float getEnergy() {
    return energy;
}

public void setSpeed(float speed) {
    this.speed = speed;
}

public float getSpeed() {
    return speed;
}

public Location getLocation() {
    return location;
}

public void setLocation(Location location) {
    this.location = location;
}

public Integer[] getOthers(BeingInterface bi) {
    others = bi.getPointInfo(this).getEntities(this, others);
    return others;
}

public void setEnergy(float energy) {
    this.energy = energy;
}

public String toString() {
    return this.getClass().getName();
}

public int compareTo(QBeing other) {
    float e1 = getEnergy();
    float e2 = other.getEnergy();
    return -Float.compare(e1, e2);
}

public QBeingAutomata getAutomata() {
    return automata;
}

public void setDestination(Location destination) {
    this.destination = destination;
}

public Location getDestination() {
    return destination;
}

public int getState() {
    return automata.getState();
}

public float beingWillEat() {
```

```
        float willEat = 0;
        if (GlobalStrategy.getType(getState()) == BeingType.TYPE_EXPLORER) {
            if (getEnergy() < BehaviourConstants.NORM_EXP_FOOD * getMass()) {
                willEat = Math.min(getMass() * Constants.K_bite,
BehaviourConstants.NORM_EXP_FOOD * getMass() - getEnergy());
            }
        } else if (GlobalStrategy.getType(getState()) == BeingType.TYPE_EATER &&
getState() == BeingState.S_PRODUCER) {
            willEat = Math.min(getMass() * Constants.K_bite, Constants.K_toborn
* getMass() - getEnergy());
        }
        return willEat;
    }

    public void setAction(Event action) {
        this.action = action;
    }

    private Integer[] entitiesHere = new Integer[1];

    private float getEnemyForces(BeingInterface bi) {
        PointInfo pointInfo = bi.getPointInfo(this);
        entitiesHere = pointInfo.getEntities(this, entitiesHere);
        Object owner = GlobalStrategy.instance().getOwner();
        float force = 0;
        for(Integer id: entitiesHere) {
            if (id == null)
                break;
            if (bi.getOwner(this, id) != owner) {
                force += bi.getMass(this, id);
            }
        }
        return force;
    }

    public float getForce() {
        return getEnergy() - getMass() * Constants.K_emin;
    }
}
```

QBeingAutomata.java

```
package ru.ifmo.qbeing;

import universum.bi.*;

import ru.ifmo.qbeing.global.GlobalStrategy;
import ru.ifmo.qbeing.global.UniverseInfo;
import ru.ifmo.qbeing.common.LocationInfo;
import ru.ifmo.qbeing.common.BehaviourConstants;
import ru.ifmo.qbeing.common.BeingUtil;

import java.util.List;
import java.util.Collection;

public class QBeingAutomata {
    private QBeing being;
```

```
    private int state;
    private StateMachineContext context;

    public QBeingAutomata(QBeing being) {
        this.being = being;
        state = BeingState.S_BORN;
    }

// ------------------ Main Automata event processor ------------------------

    public void processBeingEvent(int event, StateMachineContext context) {
        this.context = context;
        switch (state) {
            case BeingState.S_BORN:
                switch(event) {
                    case BeingEvent.E100:
                        z101(); // init being
                        z102(); // ask GlobalStrategy for required being type
                        break;
                    case BeingEvent.E102:
                        state = BeingState.S_PRODUCER;
                        break;
                    case BeingEvent.E103:
                        state = BeingState.S_EXPLORING;
                        break;
                    case BeingEvent.E104:
                        state = BeingState.S_EATER_MOVING_TO_POINT;
                        break;
                }
                break;
            case BeingState.S_EXPLORING:
                switch(event) {
                    case BeingEvent.E101:
                        z100(); // new move started
                        break;
                    case BeingEvent.E105:
                        z104();
                        break;
                    case BeingEvent.E106:
                        if (x112()) {
                            z117();
                            state = BeingState.S_EXPLORING_HUNGRY;
                            break;
                        }
                        if (x113()) {
                            z118();
                            state = BeingState.S_PRODUCER;
                            break;
                        }
                        z119();
                        break;
                    case BeingEvent.E107:
                        state = BeingState.S_EATER_FIND_FEED_PLACE;
                        break;
                }
                break;
            case BeingState.S_EXPLORING_HUNGRY:
                switch(event) {
                    case BeingEvent.E101:
```

```
                z100(); // new move started
                break;
            case BeingEvent.E105:
                z104(); // inform strategy about being death
                break;
            case BeingEvent.E106:
                if (x114()) {
                    z120();
                    state = BeingState.S_EATER_FIND_FEED_PLACE;
                    break;
                }
                if (x110()) {
                    z115();
                    state = BeingState.S_EXPLORING;
                    break;
                }
                if (x105()) {
                    z114();
                    break;
                }
                if (x111()) {
                    z116();
                    break;
                }
                break;
        }
        break;
    case BeingState.S_PRODUCER:
        switch(event) {
            case BeingEvent.E101:
                z100(); // new move started
                break;
            case BeingEvent.E105:
                z104(); // inform strategy about being death
                z105(); // remove colony supplier
                break;
            case BeingEvent.E103:
                state = BeingState.S_EXPLORING;
                break;
            case BeingEvent.E106:
                if (x102()) {
                    z108();
                    break;
                }
                if (x103()) {
                    z109(); // Transform colony to explorers
                    break;
                }
                if (x104()) {
                    z111(); // Attack
                    break;
                }
                z110();
                break;
            case BeingEvent.E110:
                if (x115()) {
                    z121();
                    break;
                }
```

```
                break;
            case BeingEvent.E111:
                state = BeingState.S_SUPPORT;
                break;
        }
        break;
    case BeingState.S_EATER_MOVING_TO_POINT:
        switch(event) {
            case BeingEvent.E101:
                z100(); // new move started

                break;
            case BeingEvent.E105:
                z104(); // inform strategy about being death
                z103(); // Inform strategy that being haven't reached
                        //it's destination
                break;
            case BeingEvent.E106:
                if (x101()) {
                    z106();
                    state = BeingState.S_PRODUCER;
                    break;
                }
                z107();
                break;
        }
        break;
    case BeingState.S_SUPPORT:
        switch(event) {
            case BeingEvent.E101:
                z100(); // new move started
                break;
            case BeingEvent.E105:
                z104(); // inform strategy about being death
                z103(); // Inform strategy that being haven't reached
                //it's destination
                break;
            case BeingEvent.E106:
                if (x101()) {
                    z122();
                    state = BeingState.S_PRODUCER;
                    break;
                }
                z107();
                break;
        }
        break;
    case BeingState.S_EATER_FIND_FEED_PLACE:
        switch(event) {
            case BeingEvent.E101:
                z100(); // new move started
                break;
            case BeingEvent.E105:
                z104(); // inform strategy about being death
                break;
            case BeingEvent.E106:
                if (x105()) {
                    z112();
                    state = BeingState.S_PRODUCER;
```

```
                        break;
                    }
                    if (x109() || x106()) {
                        z113();
                        state = BeingState.S_EATER_MOVING_TO_POINT;
                        break;
                    }
                    if (x107()) {
                        z114();
                        break;
                    }
                    if (x108()) {
                        z113();
                        state = BeingState.S_EATER_MOVING_TO_POINT;
                        break;
                    }

                    break;
                }
                break;
            }
        }

    // ------------------ End of Main Automata event processor ---------

    // --------------- Automata input actions -------------------------

        /**
         * We have reached the destination point
         * @return true, if we reched destination
         */
        private boolean x101() {
            BeingInterface bi = context.getBeingInterface();
            QBeing me = context.getBeing();
            return bi.getLocation(me).equals(me.getDestination());
        }

        /**
         * Being can born
         * @return true, if being can born
         */
        private boolean x102() {
            QBeing me = context.getBeing();
            BeingInterface bi = context.getBeingInterface();
            return bi.getEnergy(me) >= Constants.K_toborn * me.getMass();
        }

        /**
         * Lack of food, should tranform the colony to explorers
         * @return true, if food amount is not enough
         */
        private boolean x103() {
            return lackOfFood() && !GlobalStrategy.instance().isExploringFinished();
        }

        /**
         * Is there an enemy we should attack?
         * @return true, if the enemy is located, and we are healthy enough to
    attack.
```

```
     */
    private boolean x104() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        Integer other = isForeignHere(me, bi);
        float e = bi.getEnergy(me);
        context.put("enemy", other);
        return other != null && e >= BehaviourConstants.MIN_EATER_ATTACK_ENERGY
* me.getMass();
    }

    /**
     * We have enough energy at the current location.
     * @return true, if there is enough food.
     */
    private boolean x105() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        return UniverseInfo.instance().isEnoughEnergy(me, new
LocationInfo(bi.getPointInfo(me), me), BehaviourConstants.MIN_FEED_ENERGY);
    }

    /**
     * Is there an energy point with enogh energy to start a colony we can reach
     * @return true, if there is one
     */
    private boolean x106() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        Location loc = UniverseInfo.instance().findReachableEnergyPoint(me, bi,
BehaviourConstants.MIN_START_COLONY_ENERGY);
        context.put("location", loc);
        return loc != null;
    }

    /**
     * We are dying
     * @return true, if we have low energy
     */
    private boolean x107() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        return bi.getEnergy(me) < BehaviourConstants.MIN_HEALTH * me.getMass();
    }

    /**
     * Is there an energy point with some energy we can reach
     * @return true, if there is one
     */
    private boolean x108() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        Location loc = UniverseInfo.instance().findReachableEnergyPoint(me, bi,
BehaviourConstants.MIN_FEED_ENERGY);
        context.put("location", loc);
        return loc != null;
    }

    /**
```

```
     * We can reach the energy point with energy
     * @return true, if we can
     */
    private boolean x109() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        Location loc = UniverseInfo.instance().findReachableEnergyPoint(me, bi,
BehaviourConstants.MIN_FEED_ENERGY);
        if (loc != null) {
            float avail = UniverseInfo.instance().getAvailEnergyAtLoc(me, loc);
            me.setDestination(loc);
            if (avail > BehaviourConstants.CHANGE_FEED_PLACE_REQUIRED) {
                context.put("location", loc);
                return true;
            }
        }
        return false;
    }

    /**
     * We re not hugry anymore
     * @return true, if we are healthy enough to explore.
     */
    private boolean x110() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        return bi.getEnergy(me) >= BehaviourConstants.NORM_EXP_FOOD *
me.getMass();
    }

    /**
     * Is there an energy point with some energy we can reach (Greedy)
     * @return true, if there is one
     */
    private boolean x111() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        Location loc = UniverseInfo.instance().findReachableEnergyPoint(me, bi,
BehaviourConstants.MIN_FEED_ENERGY);
        if (loc == null) {
            loc = UniverseInfo.instance().findReachableEnergyPoint(me, bi,
BehaviourConstants.MIN_FEED_ENERGY, true);
        }
        context.put("location", loc);
        return loc != null;
    }

    /**
     * Explorer is hungry.
     * @return
     */
    private boolean x112() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        return bi.getEnergy(me) < BehaviourConstants.MIN_EXP_FOOD *
me.getMass();
    }

    /**
```

```
     * Can start colony.
     * @return
     */
    private boolean x113() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        PointInfo pi = bi.getPointInfo(me);
        return UniverseInfo.instance().isEnoughEnergy(me, new LocationInfo(pi,
me),
                BehaviourConstants.MIN_START_COLONY_ENERGY) &&
GlobalStrategy.instance().startColonyHere(pi, me);
    }

    /**
     * Explorer should be set to Eater.
     * @return true, if there is a point with enough food
     */
    private boolean x114() {
        return x110() && GlobalStrategy.instance().isExploringFinished();
    }

    /**
     * We need support
     * @return true, if support is needed
     */
    private boolean x115() {
        Float forces = (Float) context.get("forces");
        context.setBeing(being);
        QBeing me = context.getBeing();
        float supportNeeded = forces * BehaviourConstants.ENERGY_FROM_MASS -
GlobalStrategy.instance().getColonyForce(me);
        if (supportNeeded > 0) {
            context.put("supportNeeded", supportNeeded);
            return true;
        }
        return false;
    }
//  ------------ End of Automata input actions -------------

//  ------------ Automata output actions -------------------

    /**
     * Update being energy, location, etc.
     */
    private void z100() {
        being.setAction(null);
        BeingInterface bi = context.getBeingInterface();
        being.setLocation(bi.getLocation(being));
        GlobalStrategy.instance().updateEnergy(being, bi.getEnergy(being));
        UniverseInfo.instance().updateKnownLocations(being, bi);
        StateMachineContext ctx = new StateMachineContext();
        ctx.setBeingInterface(bi);
        EventManager.handleStrategyEvent(StrategyEvent.E200, ctx);
        EventManager.handleBeingEvent(BeingEvent.E106, being, context);
    }

    /**
     * Init being
     */
```

```
    private void z101() {
        BeingInterface bi = context.getBeingInterface();
        Event e = context.getEvent();
        if (e.param() != null) {
            BeingParams bp = (BeingParams) e.param();
            being.setMass(bp.M);
            being.setSpeed(bp.S);
            being.setEnergy(bi.getEnergy(being));
            being.setParent((QBeing) bp.parameter);
            being.setId(bi.getId(being));
            GlobalStrategy.instance().setOwner(bi.getOwner(being,
bi.getId(being)));
        }
    }

    /**
     * Ask GlobalStrategy for required being type
     */
    private void z102() {
        BeingInterface bi = context.getBeingInterface();
        StateMachineContext ctx = new StateMachineContext();
        ctx.setSender(being);
        ctx.setBeingInterface(bi);
        EventManager.handleStrategyEvent(StrategyEvent.E201, ctx);
    }

    /**
     * Inform strategy that being haven't reached it's destination
     */
    private void z103() {
        UniverseInfo.instance().removeBeingFollowingLocation(being,
being.getDestination());
        UniverseInfo.instance().removeSupportBeing(being,
being.getDestination());
    }

    /**
     * Inform strategy about being death
     */
    private void z104() {
        StateMachineContext context = new StateMachineContext();
        context.setBeing(being);
        EventManager.handleStrategyEvent(StrategyEvent.E202, context);
    }

    /**
     * Inform strategy of being death.
     */
    private void z105() {
        BeingInterface bi = context.getBeingInterface();
        GlobalStrategy.instance().colonySuplierRemoved(bi.getLocation(being),
being);
    }

    /**
     * Eater reached its destination point
     */
    private void z106() {
        QBeing me = context.getBeing();
```

```
        BeingInterface bi = context.getBeingInterface();
        UniverseInfo.instance().removeBeingFollowingLocation(me,
me.getDestination());

GlobalStrategy.instance().changeState(BeingState.S_EATER_MOVING_TO_POINT,
BeingState.S_PRODUCER, me, bi);
    }

    /**
     * Step towards destination point
     */
    private void z107() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        being.setAction(GlobalStrategy.instance().moveTo(me, bi,
me.getDestination()));
    }

    /**
     * Born
     */
    private void z108() {
        QBeing me = context.getBeing();
        BeingParams bp = me.getParams();
        bp.parameter = me;
        being.setAction(new Event(EventKind.ACTION_BORN, bp));
    }

    /**
     * Transform colony to explorers
     */
    private void z109() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        GlobalStrategy.instance().transformColonyToExplorers(me, bi);
    }

    /**
     * Process colony advice
     */
    private void z110() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        being.setAction(GlobalStrategy.instance().colonyAction(bi, me));
    }

    /**
     * Attack enemy.
     */
    private void z111() {
        being.setAction(new Event(EventKind.ACTION_ATTACK,
context.get("enemy")));
    }

    /**
     * Start eating.
     */
    private void z112() {
        QBeing me = context.getBeing();
```

```
        BeingInterface bi = context.getBeingInterface();

GlobalStrategy.instance().changeState(BeingState.S_EATER_FIND_FEED_PLACE,
BeingState.S_PRODUCER, me, bi);
    }

    /**
     * Go to energy point
     */
    private void z113() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        Location loc = (Location) context.get("location");
        UniverseInfo.instance().setBeingFollowingLocation(me, loc);
        me.setDestination(loc);

GlobalStrategy.instance().changeState(BeingState.S_EATER_FIND_FEED_PLACE,
BeingState.S_EATER_MOVING_TO_POINT, me, bi);
    }

    /**
     * Eat.
     */
    private void z114() {
        QBeing me = context.getBeing();
        being.setAction(new Event(EventKind.ACTION_EAT, me.getMass()));
    }

    /**
     * Continue exploring.
     */
    private void z115() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        GlobalStrategy.instance().changeState(BeingState.S_EXPLORING_HUNGRY,
BeingState.S_EXPLORING, me, bi);
    }

    /**
     * Explorer Step towards energy point
     */
    private void z116() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        being.setAction(GlobalStrategy.instance().moveTo(me, bi, (Location)
context.get("location")));
    }

    /**
     * Set Explorer to hungry state
     */
    private void z117() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        GlobalStrategy.instance().changeState(BeingState.S_EXPLORING,
BeingState.S_EXPLORING_HUNGRY, me, bi);
    }

    /**
```

```
     * Start colony
     */
    private void z118() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        GlobalStrategy.instance().changeState(BeingState.S_EXPLORING,
BeingState.S_PRODUCER, me, bi);
    }

    /**
     * Continue exploring
     */
    private void z119() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        Location nextLocation;
        // move to one of the neighbour point to set colony
        Location enoughEnergyLocation =
UniverseInfo.instance().findEnoughEnergy(me, bi.getNeighbourInfo(me),
BehaviourConstants.MIN_START_COLONY_ENERGY);
        if (enoughEnergyLocation != null) {
            nextLocation = enoughEnergyLocation;
        } else {
            // continue search
            nextLocation = calculateNextRadiusLocation(bi, me);
        }
        being.setAction(GlobalStrategy.instance().moveTo(me, bi, nextLocation));
    }

    /**
     * Set Eater being to "Find food" state
     */
    private void z120() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        GlobalStrategy.instance().changeState(BeingState.S_EXPLORING_HUNGRY,
BeingState.S_EATER_FIND_FEED_PLACE, me, bi);
    }

    /**
     * Call support
     */
    private void z121() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        Float supportNeeded = (Float) context.get("supportNeeded");

        GlobalStrategy.instance().getSupport(bi, me.getLocation(),
supportNeeded);
    }

    /**
     * Support reached its destination point
     */
    private void z122() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        UniverseInfo.instance().removeSupportBeing(me, me.getDestination());
```

```
        GlobalStrategy.instance().changeState(BeingState.S_SUPPORT,
BeingState.S_PRODUCER, me, bi);
    }
//  ------------ End of Automata output actions -------------

//  ------------- Misc. Methods ------------
    /**
     * Returns current state
     * @return state
     */
    public int getState() {
        return state;
    }

    private boolean lackOfFood() {
        QBeing me = context.getBeing();
        BeingInterface bi = context.getBeingInterface();
        PointInfo pi = bi.getPointInfo(me);
        float avail = UniverseInfo.instance().getAvailEnergy(me, new
LocationInfo(pi, me));
        return avail < BehaviourConstants.MIN_FEED_ENERGY ;
    }

    private Integer isForeignHere(QBeing me, BeingInterface bi) {
        Object owner = GlobalStrategy.instance().getOwner();
        Integer[] others = me.getOthers(bi);
        for (Integer other : others) {
            if (other == null) {
                break;
            }
            if (owner != bi.getOwner(me, other)) {
                return other;
            }
        }
        return null;
    }

    private Location calculateNextRadiusLocation(BeingInterface bi, QBeing me) {
        Location resultLocation;
        List<Location> reachable = bi.getReachableLocations(me);
        Location center = UniverseInfo.instance().getExploreCenters(me);
        Collection<Location> notVisited =
UniverseInfo.instance().filterVisited(bi, reachable);
        if (notVisited.isEmpty()) {
            resultLocation = BeingUtil.getRandomFarest(bi, center, reachable);
        } else {
            resultLocation = BeingUtil.getRandomNearest(bi, center, notVisited);
        }
        return resultLocation;
    }
}
```

StateMachineContext.java

```
package ru.ifmo.qbeing;

import universum.bi.BeingInterface;
```

```
import universum.bi.Event;

import java.util.HashMap;

public class StateMachineContext extends HashMap<String, Object> {
    public StateMachineContext() {
    }

    public StateMachineContext(QBeing me, BeingInterface bi) {
        setBeing(me);
        setBeingInterface(bi);
    }

    public void setBeing(QBeing me) {
        put("being", me);
    }

    public QBeing getBeing() {
        return (QBeing) get("being");
    }

    public BeingInterface getBeingInterface() {
        return (BeingInterface) get("bi");
    }

    public Event getEvent() {
        return (Event) get("e");
    }

    public void setBeingInterface(BeingInterface bi) {
        put("bi", bi);
    }

    public void setEvent(Event e) {
        put("e", e);
    }

    public void setSender(QBeing sender) {
        put("sender", sender);
    }

    public QBeing getSender() {
        return (QBeing) get("sender");
    }
}
```

StrategyAutomata.java

```
package ru.ifmo.qbeing;

import ru.ifmo.qbeing.global.GlobalStrategy;
import universum.bi.BeingInterface;

public class StrategyAutomata {
    private int state;
    private GlobalStrategy strategy;
    private StateMachineContext context;
```

```
    public StrategyAutomata(GlobalStrategy strategy) {
        state = StrategyState.S_EXPLORING;
        this.strategy = strategy;
    }

    public void processStrategyEvent(int event, StateMachineContext context) {
        this.context = context;
        switch (state) {
            case StrategyState.S_EXPLORING:
                switch (event) {
                    case StrategyEvent.E200:
                        z200();
                        break;
                    case StrategyEvent.E201:
                        z201();
                        break;
                    case StrategyEvent.E202:
                        z202();
                        break;
                }
                break;
        }
    }

    private void z202() {
        strategy.beingDead(context.getBeing());
    }

    /**
     * Make calculations before each move
     */
    private void z200() {
        strategy.checkStep(context.getBeingInterface());
    }

    /**
     * Advise being state
     */
    private void z201() {
        QBeing me = context.getSender();
        BeingInterface bi = context.getBeingInterface();
        strategy.setInitialState(me, bi);
        strategy.incBeingCount();
    }
}
```

StrategyEvent.java

```
package ru.ifmo.qbeing;

public interface StrategyEvent {
    // Strategy Events
    int E201 = 201; // Ask strategy about required being type
    int E200 = 200; // Make calculations before being move
    int E202 = 202; // being dead
}
```

StrategyState.java

```
package ru.ifmo.qbeing;

public interface StrategyState {
    // Strategy states
    int S_EXPLORING = 20;
}
```

BehaviourConstants.java

```
package ru.ifmo.qbeing.common;

public interface BehaviourConstants {
    float MIN_EXP_FOOD = 0.3f; // %
    float NORM_EXP_FOOD = 0.5f;

    float MIN_HEALTH = 0.151f; // %
    float MIN_FEED_ENERGY = 0.1f; // %
    float MIN_START_COLONY_ENERGY = 0.05f; // %
    float EQUAL_FEED_ENERGY = 0.1f;

    // Exploration constants
    float MAP_EXPLORED_RATE_SINGLE = 0.99f;
    float MAP_EXPLORED_RATE_DUEL = 0.99f;
    float MAP_EXPLORED_RATE_JUNGLE = 0.95f;


    float MIN_EATER_PERCENT = 0.01f;
    float EQUAL_DISTANCE = 1f;
    float MIN_EATER_ATTACK_ENERGY = 0.3f;

    int MAX_EXPLORE_STEPS_SINGLE = 200;
    int MAX_EXPLORE_STEPS_DUEL = 150;
    int MAX_EXPLORE_STEPS_JUNGLE = 200;

    int MAX_EXPLORER_COUNT = 1000;
    float MIN_FOOD_FOR_EATER = 0.1f;
    float CHANGE_FEED_PLACE_REQUIRED = 0.2f;

    float KEEP_EATERS_WHEN_LACK_OF_FOOD = 0.1f;

    float MIN_ENERGY_TO_GIVE = 0.2f;
    float GIVE_TO_OTHER_IF_REQUIRED_LESS = 0.01f;
    float MAX_GIVE_ENERGY = 0.04f;
    float FEED_NOT_TO_DIE = 0.1f;

    int MAX_MASS = 4;
    int SPEED = 3;

    float ENERGY_FROM_MASS = 0.6f;
    float SUPPORT_ENERGY = 0.5f;
}
```

BeingType.java

```
package ru.ifmo.qbeing.common;
```

```
public enum BeingType {
    TYPE_EXPLORER,
    TYPE_EATER
}
```

BeingUtil.java

```
package ru.ifmo.qbeing.common;

import ru.ifmo.qbeing.QBeing;
import universum.bi.BeingInterface;
import universum.bi.Constants;
import universum.bi.Location;
import universum.util.Util;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

@SuppressWarnings("deprecation")
public class BeingUtil {
    public static Location getRandomNearest(BeingInterface bi, Location center,
Collection<Location> points) {
        return getLocationFarestOrNearestRandom(bi, center, points, false);
    }

    public static Location getRandomFarest(BeingInterface bi, Location center,
Collection<Location> points) {
        return getLocationFarestOrNearestRandom(bi, center, points, true);
    }

    public static Location getLocationEqual(BeingInterface bi, Location center,
Collection<Location> points, Location locEqual) {
        float findDist = bi.distance(center, locEqual);
        Location perfectLoc = null;
        float minDiff = 0;
        boolean bFirst = true;
        for (Location loc : points) {
            float dist = bi.distance(center, loc);
            if (dist - findDist < minDiff || bFirst) {
                minDiff = dist - findDist;
                perfectLoc = loc;
            }
            bFirst = false;
        }
        return perfectLoc;
    }

    public static Location getLocationFarestOrNearestRandom(BeingInterface bi,
Location center, Collection<Location> points, boolean farest) {
        boolean bFirst = true;
        float min = 0;
        float max = 0;
        List<Location> minLoc = new ArrayList<Location>();
        List<Location> maxLoc = new ArrayList<Location>();
        for (Location loc : points) {
            float dist = bi.distance(center, loc);
```

```
            if (dist > max || bFirst) {
                max = dist;
                maxLoc.clear();
                maxLoc.add(loc);
            } else {
                if (Math.abs(dist - max) < BehaviourConstants.EQUAL_DISTANCE) {
                    maxLoc.add(loc);
                }
            }
            if (dist < min || bFirst) {
                min = dist;
                minLoc.clear();
                minLoc.add(loc);
            } else {
                if (Math.abs(dist - min) < BehaviourConstants.EQUAL_DISTANCE) {
                    minLoc.add(loc);
                }
            }
            bFirst = false;
        }
        if (farest) {
            if (maxLoc.isEmpty()) {
                return null;
            }
            return maxLoc.get(Util.rnd(maxLoc.size()));
        } else {
            if (minLoc.isEmpty()) {
                return null;
            }
            return minLoc.get(Util.rnd(minLoc.size()));
        }
    }

    // - ftr: fake step count
    public static int getStepsCount(BeingInterface bi, Location loc1, Location
loc2) {
        return (int) bi.distance(loc1, loc2);
    }

    public static float getEnergyToReach(QBeing me, BeingInterface bi, Location
loc) {
        return (float) Math.ceil(bi.distance(bi.getLocation(me), loc) /
me.getSpeed()) * Constants.K_movecost * me.getSpeed();
    }

    public static Location stepToward(BeingInterface bi, QBeing me, Location to)
{
        Location from = bi.getLocation(me);
        float speed = me.getSpeed();
        int dx = to.getX() - from.getX();
        int dy = to.getY() - from.getY();
        int dirx = dx == 0 ? 0 : (dx > 0 ? 1 : -1);
        int diry = dy == 0 ? 0 : (dy > 0 ? 1 : -1);

        int ddx = Math.abs(dx);
        int ddy = Math.abs(dy);
        if (ddx > Constants.getWidth() - ddx) {
            ddx = Constants.getWidth() - ddx;
            dirx = -dirx;
```

```
        }
        if (ddy > Constants.getHeight() - ddy) {
            ddy = Constants.getHeight() - ddy;
            diry = -diry;
        }

        if (ddx <= speed && ddy <= speed) {
            return to;
        }

        int s = (int) speed;
        ddx = Math.min(ddx, s) * dirx;
        ddy = Math.min(ddy, s) * diry;

        int x = from.getX() + ddx;
        int y = from.getY() + ddy;
        if (x < 0)
            x += Constants.getWidth();
        if (x >= Constants.getWidth())
            x -= Constants.getWidth();
        if (y < 0)
            y += Constants.getHeight();
        if (y >= Constants.getHeight())
            y -= Constants.getHeight();

        return bi.createLocation(x, y);
    }
}
```

LocationInfo.java

```
package ru.ifmo.qbeing.common;

import ru.ifmo.qbeing.QBeing;
import universum.bi.Location;
import universum.bi.PointInfo;

public class LocationInfo {
    public float count;
    public float maxCount;
    public float growthRate;
    public Location location;

    public LocationInfo(PointInfo pi, QBeing me) {
        this.count = pi.getCount(me);
        this.maxCount = pi.getMaxCount(me);
        this.growthRate = pi.getGrowthRate(me);
        this.location = pi.getLocation();
    }
}
```

Colony.java

```
package ru.ifmo.qbeing.global;

import ru.ifmo.qbeing.*;
import ru.ifmo.qbeing.common.BehaviourConstants;
```

```
import ru.ifmo.qbeing.common.LocationInfo;
import universum.bi.*;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

@SuppressWarnings("deprecation")
public class Colony implements Comparable<Colony> {
    private LocationInfo pi;
    private Set<QBeing> supliers;

    public void addSupplier(QBeing being) {
        supliers.add(being);
    }

    public boolean removeSupplier(QBeing being) {
        supliers.remove(being);
        return supliers.size() == 0;
    }

    public Colony(LocationInfo pi, QBeing being) {
        this.pi = pi;
        this.supliers = new HashSet<QBeing>();
        this.supliers.add(being);
    }

    public Set<QBeing> getSupliers() {
        return supliers;
    }

    public Event colonyAction(BeingInterface bi, QBeing me) {
        Event ev = null;
        float energy = bi.getPointInfo(me).getCount(me);
        float consumed = 0;
        ArrayList<QBeing> ts = new ArrayList<QBeing>(this.supliers);
        Collections.sort(ts);
        QBeing toGive = null;
        float giveEnergy = 0;

        if (GlobalStrategy.instance().heal) {
            for (int i = ts.size() - 1; i >= 0; i--) {
                if (consumed > energy) {
                    break;
                }
                QBeing being = ts.get(i);
                if (being.getEnergy() < BehaviourConstants.MIN_HEALTH *
being.getMass()) {
                    if (me == being) {
                        break;
                    }
                    float energyRequired = being.getMass() *
BehaviourConstants.FEED_NOT_TO_DIE;
                    if (toGive == null) {
                        toGive = being;
                        giveEnergy = energyRequired;
                    }
                    consumed += energyRequired;
```

```
                }
            }
        }
        if (toGive == null && ev == null)
            for (QBeing being : ts) {
                if (being.getState() == BeingState.S_PRODUCER) {
                    if (consumed > energy) {
                        break;
                    }
                    if (being == me && me.getEnergy() < Constants.K_toborn *
me.getMass()) {
                        float give = Math.min(me.getMass() - me.getEnergy(),
me.getMass() * Constants.K_bite);
                        updateSupplierEnergy(me, me.getEnergy() + give);

                        StateMachineContext ctx = new StateMachineContext(me,
bi);
                        ctx.put("amount", give);
                        EventManager.handleBeingEvent(BeingEvent.E108, me, ctx);

                        ev = new Event(EventKind.ACTION_EAT, me.getMass() -
me.getEnergy());
                        break;
                    }
                    float energyRequired = being.getMass() * Constants.K_toborn
- being.getEnergy();
                    if (energyRequired >=
BehaviourConstants.GIVE_TO_OTHER_IF_REQUIRED_LESS * me.getMass() && toGive ==
null) {
                        toGive = being;
                        giveEnergy = calcGiveMaxEnergy(me, bi, energyRequired);
                    }
                    consumed += Math.min(giveEnergy, being.getMass() *
Constants.K_bite);
                }
            }
        if (ev == null && toGive != null) {
            if (bi.getEnergy(me) >= BehaviourConstants.MIN_ENERGY_TO_GIVE *
me.getMass()) {
                float give = Math.min(giveEnergy,
BehaviourConstants.MAX_GIVE_ENERGY * me.getMass());
                updateSupplierEnergy(toGive, toGive.getEnergy() + give);
                updateSupplierEnergy(me, me.getEnergy() - give);

                StateMachineContext ctx = new StateMachineContext(me, bi);
                ctx.put("amount", give);
                ctx.put("other", toGive.getId());
                EventManager.handleBeingEvent(BeingEvent.E109, me, ctx);

                ev = new Event(EventKind.ACTION_GIVE, toGive.getId(), give);
            }
        }
        return ev;
    }

    private float calcGiveMaxEnergy(QBeing me, BeingInterface bi, float
energyRequired) {
        int turnsToLive = GlobalStrategy.instance().getMaxTurns() -
bi.getTurnsCount();
```

```
        float energyForTurns = (Constants.K_masscost * me.getMass() *
(turnsToLive + 1));
        float energyToGive = me.getEnergy() - energyForTurns;
        return Math.min(energyToGive, energyRequired);
    }


    public String toString() {
        return "[" + supliers.size() + "] (" + pi.location.toString() + ")";
    }

    public void updateSupplierEnergy(QBeing me, float energy) {
        me.setEnergy(energy);
    }

    public Location getLocation() {
        return pi.location;
    }

    public int compareTo(Colony o) {
        int s1 = getSupliers().size();
        int s2 = o.getSupliers().size();
        return s1 == s2 ? 0 : (s1 > s2 ? 0 : 1);
    }

    public boolean equals(Object obj) {
        return getLocation().equals(((Colony) obj).getLocation());
    }
}
```

ColonyControl.java

```
package ru.ifmo.qbeing.global;

import ru.ifmo.qbeing.QBeing;
import ru.ifmo.qbeing.common.LocationInfo;
import universum.bi.*;

import java.util.*;

public class ColonyControl {
    protected Map<Location, Colony> colonies;
    private List<Location> golden;

    public ColonyControl() {
        colonies = new HashMap<Location, Colony>();
    }

    public boolean setupColony(PointInfo pi, QBeing me) {
        Location location = pi.getLocation();
        if (colonies.containsKey(location)) {
            return false;
        }
        colonies.put(location, new Colony(new LocationInfo(pi, me), me));
        return true;
    }

    public void removeColonySupplier(Location loc, QBeing me) {
```

```
        Colony ci = colonies.get(loc);
        if (ci != null) {
            if (ci.removeSupplier(me)) {
                colonies.remove(ci.getLocation());
            }
        }
    }

    public void addColonySupplier(QBeing me, BeingInterface bi) {
        Colony ci = colonies.get(bi.getLocation(me));
        if (ci != null) {
            ci.addSupplier(me);
        } else {
            setupColony(bi.getPointInfo(me), me);
        }
    }

    public QBeing[] getSuppliers(Location loc) {
        Colony ci = colonies.get(loc);
        QBeing[] suppliers = null;
        if (ci != null) {
            suppliers = ci.getSupliers().toArray(new QBeing[1]);
        }
        return suppliers;
    }

    public Set<QBeing> getSuppliersList(Location loc) {
        Colony ci = colonies.get(loc);
        if (ci != null) {
            return ci.getSupliers();
        }
        return null;
    }

    public Event colonyAction(BeingInterface bi, QBeing me) {
        Colony colony = colonies.get(bi.getLocation(me));
        if (colony != null) {
            return colony.colonyAction(bi, me);
        } else {
            return null;
        }
    }

    public void updateupdateSupplierEnergy(QBeing me) {
        Colony ci = colonies.get(me.getLocation());
        if (ci != null) {
            ci.updateSupplierEnergy(me, me.getEnergy());
        }
    }

    public List<Location> getGoldenLocations() {
        if (golden == null) {
            golden = new ArrayList<Location>();
            calcGolden();
        }
        return golden;
    }

    private void calcGolden() {
```

```
        List<Colony> largestColonies = new ArrayList<Colony>();
        for (Colony c : colonies.values()) {
            if (!largestColonies.contains(c)) {
                largestColonies.add(c);
                Collections.sort(largestColonies);
                if (largestColonies.size() > Constants.NUM_GOLDEN) {
                    largestColonies.remove(largestColonies.size() - 1);
                }
            }
        }
        for (Colony c : largestColonies) {
            golden.add(c.getLocation());
        }
    }
}
```

GlobalStrategy.java

```
package ru.ifmo.qbeing.global;

import ru.ifmo.qbeing.*;
import ru.ifmo.qbeing.common.*;
import universum.bi.*;

import java.util.*;

public class GlobalStrategy {
    protected static GlobalStrategy instance;

    protected ColonyControl colonyControl;

    protected int beingCount;
    protected Map<BeingType, Integer> beingTypeCount;
    protected Map<BeingType, Set<QBeing>> beingsByType;

    protected boolean exploringFinished = false;
    private Object owner;
    private int maxTurns;
    private int maxExplorerSteps;
    private int lastKnownLocationsUpdatedTurn;
    private float exploredRate;
    public boolean heal;

    private StrategyAutomata automata;

    public GlobalStrategy() {
        automata = new StrategyAutomata(this);
        colonyControl = new ColonyControl();
        beingsByType = new HashMap<BeingType, Set<QBeing>>();
        beingCount = 0;
        beingTypeCount = new HashMap<BeingType, Integer>();
    }

    public static synchronized void initInstance() {
        instance = new GlobalStrategy();
    }

    public static synchronized GlobalStrategy instance() {
```

```
        if (instance == null) {
            initInstance();
        }
        return instance;
    }

    public void checkExploringFinished(BeingInterface bi) {
        if (exploringFinished)
            return;
        if (bi.getTurnsCount() > maxExplorerSteps ||
UniverseInfo.instance().getExploredLocationCount() >= getExploredRate() *
Constants.getWidth() * Constants.getHeight()) {
            convertExplorersToEaters(bi);
            exploringFinished = true;
        }
    }

    private void convertExplorersToEaters(BeingInterface bi) {
        Set<QBeing> beingsSet = getBeingsByType(BeingType.TYPE_EXPLORER);
        QBeing[] beings = beingsSet.toArray(new QBeing[beingsSet.size()]);
        for (QBeing being : beings) {
            changeState(being.getState(), BeingState.S_EATER_FIND_FEED_PLACE,
being, bi);
            EventManager.handleBeingEvent(BeingEvent.E107, being, new
StateMachineContext(being, bi));
        }
    }

    private Integer getBeingTypeCount(BeingType type) {
        Integer i = beingTypeCount.get(type);
        if (i == null)
            i = 0;
        return i;
    }

    private Set<QBeing> getBeingsByType(BeingType type) {
        Set<QBeing> beings = beingsByType.get(type);
        if (beings == null) {
            beings = new HashSet<QBeing>();
            beingsByType.put(type, beings);
        }
        return beings;
    }

    public synchronized int getBeingsByTypeCount(BeingType type) {
        return getBeingsByType(type).size();
    }

    private void incBeingTypeCount(QBeing me, int state) {
        BeingType type = getType(state);
        beingTypeCount.put(type, getBeingTypeCount(type) + 1);
        getBeingsByType(type).add(me);
    }

    public static BeingType getType(int state) {
        switch(state) {
            case BeingState.S_EATER_FIND_FEED_PLACE:
            case BeingState.S_EATER_MOVING_TO_POINT:
            case BeingState.S_PRODUCER:
```

```
            return BeingType.TYPE_EATER;
        case BeingState.S_EXPLORING:
        case BeingState.S_EXPLORING_HUNGRY:
            return BeingType.TYPE_EXPLORER;
    }
    return null;
}

private void decBeingTypeCount(QBeing me, int state) {
    if (state != 0) {
        BeingType type = getType(state);
        beingTypeCount.put(type, getBeingTypeCount(type) - 1);
        getBeingsByType(type).remove(me);
    }
}

public synchronized void beingDead(QBeing me) {
    UniverseInfo.instance().removeBeingFromLocation(me, me.getLocation());
    decBeingTypeCount(me, me.getState());
    beingCount--;
}

public boolean startColonyHere(PointInfo pi, QBeing me) {
    return colonyControl.setupColony(pi, me);
}

public synchronized void colonySuplierRemoved(Location loc, QBeing me) {
    colonyControl.removeColonySupplier(loc, me);
}

private void setColonySupplier(QBeing me, BeingInterface bi) {
    colonyControl.addColonySupplier(me, bi);
}

public synchronized void setOwner(Object owner) {
    if (this.owner == null)
        this.owner = owner;
}

public synchronized Object getOwner() {
    return owner;
}

public synchronized void checkStep(BeingInterface bi) {
    checkGameState(bi, bi.getTurnsCount());
}

private void checkGameState(BeingInterface bi, int turnsCount) {
    if (turnsCount >= maxExplorerSteps) {
        convertExplorersToEaters(bi);
        exploringFinished = true;
    }
    if (lastKnownLocationsUpdatedTurn != turnsCount) {
        UniverseInfo.instance().updateEnergyAtKnownLocations();
        lastKnownLocationsUpdatedTurn = turnsCount;
    }
}

public synchronized void initGameInfo(UserGameInfo info) {
```

```
        maxTurns = info.maxTurns;
        switch (info.kind) {
            case SINGLE:
                maxExplorerSteps = BehaviourConstants.MAX_EXPLORE_STEPS_SINGLE;
                exploredRate = BehaviourConstants.MAP_EXPLORED_RATE_SINGLE;
                heal = true;
                break;
            case DUEL:
                maxExplorerSteps = BehaviourConstants.MAX_EXPLORE_STEPS_DUEL;
                exploredRate = BehaviourConstants.MAP_EXPLORED_RATE_DUEL;
                heal = true;
                break;
            case JUNGLE:
                maxExplorerSteps = BehaviourConstants.MAX_EXPLORE_STEPS_JUNGLE;
                exploredRate = BehaviourConstants.MAP_EXPLORED_RATE_JUNGLE;
                heal = true;
                break;
        }
    }

    public synchronized void transformColonyToExplorers(QBeing me,
BeingInterface bi) {
        if (!exploringFinished) {
            QBeing[] suppliers = colonyControl.getSuppliers(bi.getLocation(me));
            if (suppliers != null) {
                int i = (int)
(Math.ceil(BehaviourConstants.KEEP_EATERS_WHEN_LACK_OF_FOOD *
suppliers.length));
                if (i == 0)
                    i = 1;
                if (getBeingTypeCount(BeingType.TYPE_EXPLORER) +
suppliers.length > BehaviourConstants.MAX_EXPLORER_COUNT) {
                    i = Math.max(i, suppliers.length -
(BehaviourConstants.MAX_EXPLORER_COUNT -
getBeingTypeCount(BeingType.TYPE_EXPLORER)));
                }
                while (i < suppliers.length) {
                    QBeing being = suppliers[i];
                    changeState(being.getState(), BeingState.S_EXPLORING, being,
bi);
                    EventManager.handleBeingEvent(BeingEvent.E103, being, new
StateMachineContext(being, bi));
                    i++;
                }
            }
        }
    }

    public synchronized float getExploredRate() {
        return exploredRate;
    }

    public synchronized Event colonyAction(BeingInterface bi, QBeing me) {
        return colonyControl.colonyAction(bi, me);
    }

    public synchronized void changeState(int oldState, int newState, QBeing me,
BeingInterface bi) {
        decBeingTypeCount(me, oldState);
```

```
        incBeingTypeCount(me, newState);
        if (me.getState() == BeingState.S_PRODUCER) {
            colonySuplierRemoved(me.getLocation(), me);
            colonySuplierRemoved(bi.getLocation(me), me);
        }
        if (newState == BeingState.S_PRODUCER) {
            setColonySupplier(me, bi);
        }
        if (newState == BeingState.S_EXPLORING) {
            UniverseInfo.instance().setExplorerCenter(me, bi.getLocation(me));
        }
    }

    public synchronized Event moveTo(QBeing me, BeingInterface bi, Location
location) {
        UniverseInfo.instance().updateBeingLocation(me, bi, bi.getLocation(me));
        UniverseInfo.instance().updateBeingLocation(me, bi, location);
        me.setLocation(location);
        return new Event(EventKind.ACTION_MOVE_TO, BeingUtil.stepToward(bi, me,
location));
    }

    public synchronized Set<QBeing> getAllBeings() {
        Set<QBeing> set = new HashSet<QBeing>();
        for (Set<QBeing> gbs : beingsByType.values()) {
            set.addAll(gbs);
        }
        return set;
    }

    public synchronized void updateEnergy(QBeing me, float energy) {
        me.setEnergy(energy);
        colonyControl.updateupdateSupplierEnergy(me);
    }

    public synchronized int getMaxTurns() {
        return maxTurns;
    }

    public StrategyAutomata getAutomata() {
        return automata;
    }

    public synchronized void incBeingCount() {
        beingCount++;
    }

    public synchronized void setInitialState(QBeing me, BeingInterface bi) {
        checkExploringFinished(bi);
        float avail = UniverseInfo.instance().getAvailEnergy(me, new
LocationInfo(bi.getPointInfo(me), me));
        if (avail > BehaviourConstants.MIN_FOOD_FOR_EATER * me.getMass()) {
            EventManager.handleBeingEvent(BeingEvent.E102, me, new
StateMachineContext(me, bi));
            setColonySupplier(me, bi);
        } else if (!exploringFinished) {
            if (avail < BehaviourConstants.MIN_FOOD_FOR_EATER * me.getMass()) {
                EventManager.handleBeingEvent(BeingEvent.E103, me, new
StateMachineContext(me, bi));
```

```
                transformColonyToExplorers(me, bi);
            } else if (getBeingTypeCount(BeingType.TYPE_EATER) < beingCount *
BehaviourConstants.MIN_EATER_PERCENT ||
                    getBeingTypeCount(BeingType.TYPE_EXPLORER) >=
BehaviourConstants.MAX_EXPLORER_COUNT) {
                EventManager.handleBeingEvent(BeingEvent.E102, me, new
StateMachineContext(me, bi));
                setColonySupplier(me, bi);
            } else {
                EventManager.handleBeingEvent(BeingEvent.E102, me, new
StateMachineContext(me, bi));
            }
        } else {
            Location loc = UniverseInfo.instance().findReachableEnergyPoint(me,
bi, BehaviourConstants.CHANGE_FEED_PLACE_REQUIRED);
            if (loc != null) {
                me.setDestination(loc);
                EventManager.handleBeingEvent(BeingEvent.E104, me, new
StateMachineContext(me, bi));
            } else {
                EventManager.handleBeingEvent(BeingEvent.E102, me, new
StateMachineContext(me, bi));
                setColonySupplier(me, bi);
            }
        }
        changeState(0, me.getState(), me, bi);
    }


    public boolean isExploringFinished() {
        return exploringFinished;
    }

    public synchronized float getColonyForce(QBeing me) {
        Set<QBeing> suppliers =
colonyControl.getSuppliersList(me.getLocation());
        float force = 0;
        if (suppliers != null) {
            for (QBeing being: suppliers) {
                force += being.getForce();
            }
        }
        force += UniverseInfo.instance().getSupport(me.getLocation());
        return force;
    }

    public synchronized void getSupport(final BeingInterface bi, final Location
location, Float supportNeeded) {
        Set<QBeing> eaters = getBeingsByType(BeingType.TYPE_EATER);
        List<QBeing> potentialFighters = new ArrayList<QBeing>();
        for (QBeing being: eaters) {
            if (!being.getLocation().equals(location)) {
                if (being.getEnergy() >= being.getMass() *
BehaviourConstants.SUPPORT_ENERGY) {
                    potentialFighters.add(being);
                }
            }
        }
        Collections.sort(potentialFighters, new Comparator<QBeing>() {
```

```
            public int compare(QBeing o1, QBeing o2) {
                float d1 = bi.distance(o1.getLocation(), location);
                float d2 = bi.distance(o2.getLocation(), location);
                return Float.compare(d1, d2);
            }
        });
        float force = 0;
        for (QBeing b: potentialFighters) {
            if (force >= supportNeeded) {
                break;
            }
            force += b.getForce();
            makeBeingWarrior(bi, b, location);
        }
    }

    private void makeBeingWarrior(BeingInterface bi, QBeing b, Location
location) {
        UniverseInfo.instance().setSupportBeing(b, location);
        StateMachineContext ctx = new StateMachineContext(b, bi);
        b.setDestination(location);
        EventManager.handleBeingEvent(BeingEvent.E111, b, ctx);
    }
}
```

UniverseInfo.java

```
package ru.ifmo.qbeing.global;

import universum.bi.*;
import ru.ifmo.qbeing.common.LocationInfo;
import ru.ifmo.qbeing.common.BeingUtil;
import ru.ifmo.qbeing.common.BehaviourConstants;
import ru.ifmo.qbeing.QBeing;

import java.util.*;

public class UniverseInfo {
    protected Map<Location, LocationInfo> knownLocations;
    protected Map<Location, LocationInfo> foodLocations;
    protected Map<QBeing, Location> beingLocations;
    protected Map<QBeing, Location> exploreCenters;
    protected Map<Location, Set<QBeing>> beingsByLocation;
    protected Map<Location, Set<QBeing>> beingFollowingLocation;
    protected Map<Location, Set<QBeing>> beingSupportLocation;

    protected static UniverseInfo instance = null;

    protected UniverseInfo() {
        knownLocations = new HashMap<Location, LocationInfo>();
        foodLocations = new HashMap<Location, LocationInfo>();
        beingLocations = new HashMap<QBeing, Location>();
        exploreCenters = new HashMap<QBeing, Location>();
        beingsByLocation = new HashMap<Location, Set<QBeing>>();
        beingFollowingLocation = new HashMap<Location, Set<QBeing>>();
        beingSupportLocation = new HashMap<Location, Set<QBeing>>();
    }
```

```
    public static void initInstance() {
        instance = new UniverseInfo();
    }

    public static UniverseInfo instance() {
        if (instance == null) {
            initInstance();
        }
        return instance;
    }

    public int getExploredLocationCount() {
        return knownLocations.keySet().size();
    }

    public synchronized void updateKnownLocations(QBeing me, BeingInterface bi)
{
        List<PointInfo> neighbourInfo = bi.getNeighbourInfo(me);
        updateKnownLocation(me, bi.getPointInfo(me));
        for (PointInfo pi : neighbourInfo) {
            updateKnownLocation(me, pi);
        }
    }

    public synchronized void updateKnownLocation(QBeing me, PointInfo pi) {
        knownLocations.put(pi.getLocation(), new LocationInfo(pi, me));
        if ((pi.getMaxCount(me) > 0 && pi.getGrowthRate(me) > 0.001) ||
pi.getMaxCount(me) == -1) {
            foodLocations.put(pi.getLocation(), new LocationInfo(pi, me));
        }
    }

    public synchronized void updateBeingLocation(QBeing me, BeingInterface bi,
Location newLocation) {
        Location oldLoc = beingLocations.get(me);
        removeBeingFromLocation(me, oldLoc);

        if (newLocation != null) {
            addBeingToLocationMap(beingsByLocation, me, newLocation);
            beingLocations.put(me, newLocation);
        }
    }

    public synchronized void setExplorerCenter(QBeing me, Location location) {
        exploreCenters.put(me, location);
    }

    public synchronized Location getExploreCenters(QBeing me) {
        return exploreCenters.get(me);
    }

    public synchronized Collection<Location> filterVisited(BeingInterface bi,
List<Location> reachable) {
        HashSet<Location> loc = new HashSet<Location>();
        for (Location l : reachable) {
            if (!knownLocations.containsKey(l) && !findVisibleLocation(bi, l)) {
                loc.add(l);
            }
        }
```

```
        return loc;
    }

    private synchronized boolean findVisibleLocation(BeingInterface bi, Location
l) {
        for (Location loc : beingLocations.values()) {
            if (bi.distance(loc, l) <= 1) {
                return true;
            }
        }
        return false;
    }

    public synchronized void removeBeingFromLocation(QBeing me, Location oldLoc)
{
        beingLocations.remove(me);
        removeBeingFromLocationMap(beingsByLocation, me, oldLoc);
    }

    private void addBeingToLocationMap(Map<Location, Set<QBeing>> locMap, QBeing
me, Location newLocation) {
        Set<QBeing> beings = locMap.get(newLocation);
        if (beings == null) {
            beings = new HashSet<QBeing>();
        }
        beings.add(me);
        locMap.put(newLocation, beings);
    }

    private void removeBeingFromLocationMap(Map<Location, Set<QBeing>> locMap,
QBeing me, Location oldLoc) {
        if (oldLoc != null) {
            Set<QBeing> beings = locMap.get(oldLoc);
            if (beings != null) {
                beings.remove(me);
            }
        }
    }

    public synchronized float getAvailEnergy(QBeing me, LocationInfo pi) {
        Location loc = pi.location;
        float count = pi.count;
        float energyAlreadyConsumed = getConsumedAt(me, loc);
        return (count - energyAlreadyConsumed) < 0 ? 0 : count -
energyAlreadyConsumed;
    }

    private float getConsumedAt(QBeing me, Location loc) {
        float consumed = 0;
        Set<QBeing> beings = beingsByLocation.get(loc);
        if (beings != null) {
            for (QBeing being : beings) {
                if (being != me)
                    consumed += being.beingWillEat();
            }
        }
        Set<QBeing> beingsArriving = beingFollowingLocation.get(loc);
        if (beingsArriving != null) {
            for (QBeing being : beingsArriving) {
```

```
                if (being != me)
                    consumed += being.beingWillEat();
            }
        }
        return consumed;
    }

    public synchronized boolean isEnoughEnergy(QBeing me, LocationInfo pi, float
energyEnough) {
        return getAvailEnergy(me, pi) >= energyEnough;
    }

    public synchronized Location findEnoughEnergy(QBeing me, List<PointInfo>
neighbourInfo, float energyEnough) {
        boolean first = true;
        float max = 0;
        Location bestLoc = null;
        for (PointInfo pi : neighbourInfo) {
            float count = getAvailEnergy(me, new LocationInfo(pi, me));
            if (count >= energyEnough) {
                if (count > max || first) {
                    max = count;
                    bestLoc = pi.getLocation();
                }
                first = false;
            }
        }
        return bestLoc;
    }

    public synchronized Location findReachableEnergyPoint(QBeing me,
BeingInterface bi, float energyAmount) {
        return findReachableEnergyPoint(me, bi, energyAmount, false);
    }

    public synchronized Location findReachableEnergyPoint(QBeing me,
BeingInterface bi, float energyAmount, boolean greedy) {
        Location myLocation = bi.getLocation(me);
        float energyForMoving = me.getEnergy() - Constants.K_emin *
me.getMass();
        int myMaxSteps = (int) Math.floor(energyForMoving /
(Constants.K_movecost * me.getSpeed()));

        float maxEnergy = 0;
        float stepsForBest = 0;
        Location bestLocation = null;
        boolean first = true;
        for (Map.Entry<Location, LocationInfo> entry : foodLocations.entrySet())
{
            float energyAvail = greedy ? entry.getValue().count :
getAvailEnergy(me, entry.getValue());
            if (energyAvail >= energyAmount * me.getMass()) {
                // - ftr: improve (add move cost)
                int steps = (int) (Math.ceil((float) BeingUtil.getStepsCount(bi,
myLocation, entry.getKey())));
                if (steps <= myMaxSteps && steps * Constants.K_movecost <
energyAvail) {
                    if (maxEnergy < energyAvail ||
```

```
                            (Math.abs(maxEnergy - energyAvail) <=
BehaviourConstants.EQUAL_FEED_ENERGY
                                    && steps < stepsForBest) ||
                            first) {
                        stepsForBest = steps;
                        maxEnergy = energyAvail;
                        bestLocation = entry.getKey();
                    }
                    first = false;
                }
            }
        }
        return bestLocation;
    }

    public synchronized void updateEnergyAtKnownLocations() {
        for (LocationInfo li : knownLocations.values()) {
            li.count += li.growthRate;
        }
    }

    public synchronized void setBeingFollowingLocation(QBeing me, Location loc)
{
        addBeingToLocationMap(beingFollowingLocation, me, loc);
    }

    public synchronized void removeBeingFollowingLocation(QBeing me, Location
loc) {
        removeBeingFromLocationMap(beingFollowingLocation, me, loc);
    }

    public synchronized float getAvailEnergyAtLoc(QBeing me, Location loc) {
        LocationInfo li = knownLocations.get(loc);
        return getAvailEnergy(me, li);
    }

    public float getSupport(Location location) {
        float force = 0;
        Set<QBeing> beings = beingSupportLocation.get(location);
        if (beings != null) {
            for (QBeing being : beings) {
                force += being.getForce();
            }
        }
        return force;
    }

    public void setSupportBeing(QBeing b, Location location) {
        addBeingToLocationMap(beingSupportLocation, b, location);
    }

    public void removeSupportBeing(QBeing b, Location location) {
        removeBeingFromLocationMap(beingSupportLocation, b, location);
    }
}
```