“Механизм построения процесса изготовления игрового продукта на Java без привязки к устройству”

Автор: И.А.Мазница
Версия: 1.06 (6 Декабрь, 2007)

г.Санкт-Петербург
2007г.

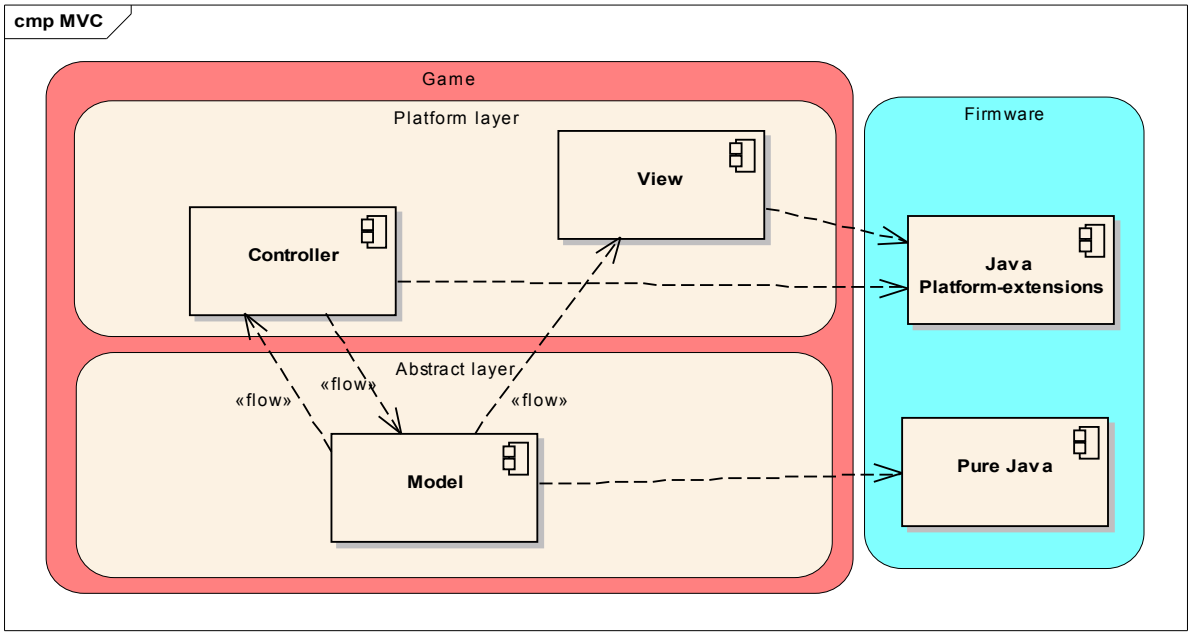
Содержание

Введение

Данный документ описывает механизм построения игрового приложения на языке Java, пригодного к последующему портированию на различные платформы, поддерживающие Java Virtual Machine.

Логическая организация ПО

Предлагается строить игровое ПО по паттерну MVC (Model-View-Controller), что позволит безболезненно подменять платформенно-зависимые компоненты, не затрагивая основную игровую логику.

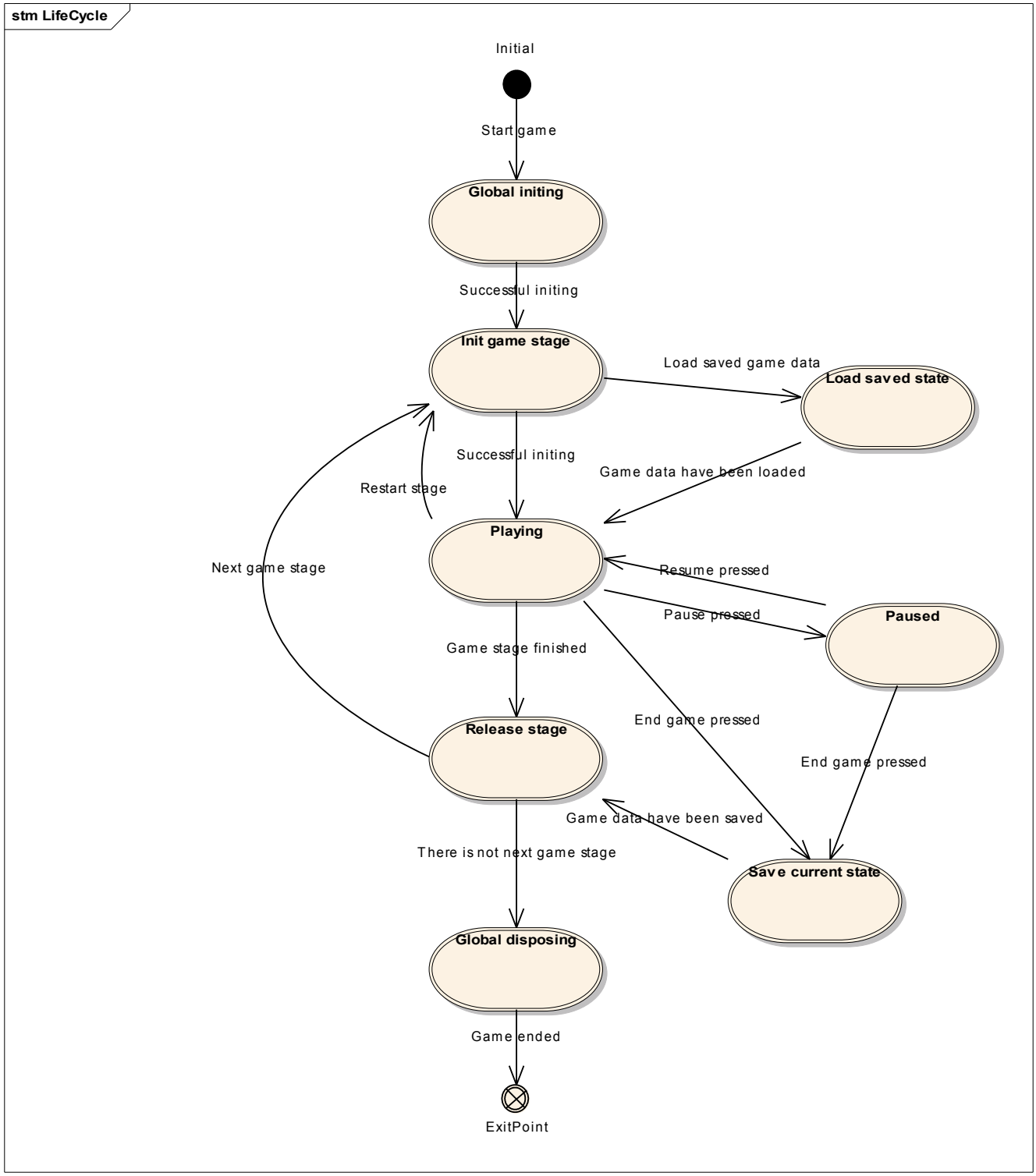


Так как потенциально каждая игра может быть разбита на ряд составляющих компонентов, то появляется возможность к вынесению ряда платформенно-зависимых действий в отдельные компоненты, которые могут безболезненно подменяться и дорабатываться третьей стороной, для портирования. Так же подобный подход упрощает процесс контроля изготовления изделия и его последующую доработку. В данном случае, мы имеем только два компонента, зависящих непосредственно от плаформы, на которой запускается игра, а именно Controller (отвечает за взаимодействие с пользователем) и View (отвечает за отображение состояния модели на экране).

Далее мы будем рассматривать только компонент Model, как основной компонент, содержащий бизнес-логику приложения.

Логические игровые состояния

Игровой компонент, проходит ряд состояний. В нижеприведенной диаграмме не показаны исключения и действия при ошибках, рассматривается идеальный случай.



Сразу стоит указать, что игровой компонент является пассивным и не может создавать управляющие потоки. Всё его функционирование обеспечивается контейнером, который осуществляет контроль и логическую последовательность вызова состояний игрового компонента. Данный контейнер так же осуществляет все платформенно-зависимые операции и взаимодействие с аппаратно-зависимыми слоями, надежно изолируя игровой компонент от непосредственной работы с ними.

Состояние Global Initing

Работа компонента начинается с глобальной инициализации, в которой он может занять системные ресурсы и произвести разовые вычисления. Данное состояние компонент проходит только один раз в своем жизненном цикле. Данное состояние может быть обработано как при непосредственном старте игры из главного меню, так и при общей загрузке игры.

Состояние Init game stage

Данное состояние отрабатывает загрузку текущего игрового уровня. Хотелось бы сразу уточнить, что данное состояние есть и у игр в которых только один игровой уровень. Данное состояние может встречаться неодин раз в жизненном цикле игры. При обработке данного состояния, производится загрузка и инициализация игровых ресурсов, относящихся непосредственно к данному игровому уровню. После отработки данного состояния, контейнер игры может перевести её в состояние загрузки сохраненных игровых данных предыдущей игровой сессии (Load saved state) или перейти в режим игры (Playing).

Состояние Load saved state

Компонент в данном состоянии производит загрузку данных из внешнего хранилища. Фактически десериализуя игровой модуль, хотя это опционально и зависит от игры. После отработки данного состояния, модуль переходит в режим игры (Playing).

Состояние Playing

Данное состояние является основным и в нём осуществляется отработка основного игрового цикла. Вся бизнес логика игрового процесса отрабатывается в этом состоянии. В случае успешного или неуспешного окончания игрового процесса для игрока, компонент переводится в состояние освобождения ресурсов игрового уровня (Release stage). Так же компонент может быть переведен в режим паузы (Paused). При команде на прерывание игрового процесса, компонент переходит в состояние сохранения игровых данных (Save current state). В ходе игрового процесса, игрок может выйти на режим “безнадежности” игрового процесса (как пример - неправильный ход, без возможности других решений) и ему требуется давать возможность перезапуска игрового уровня, что переводит компонент в состояние инициализации игрового уровня (Init game stage).

Состояние Paused

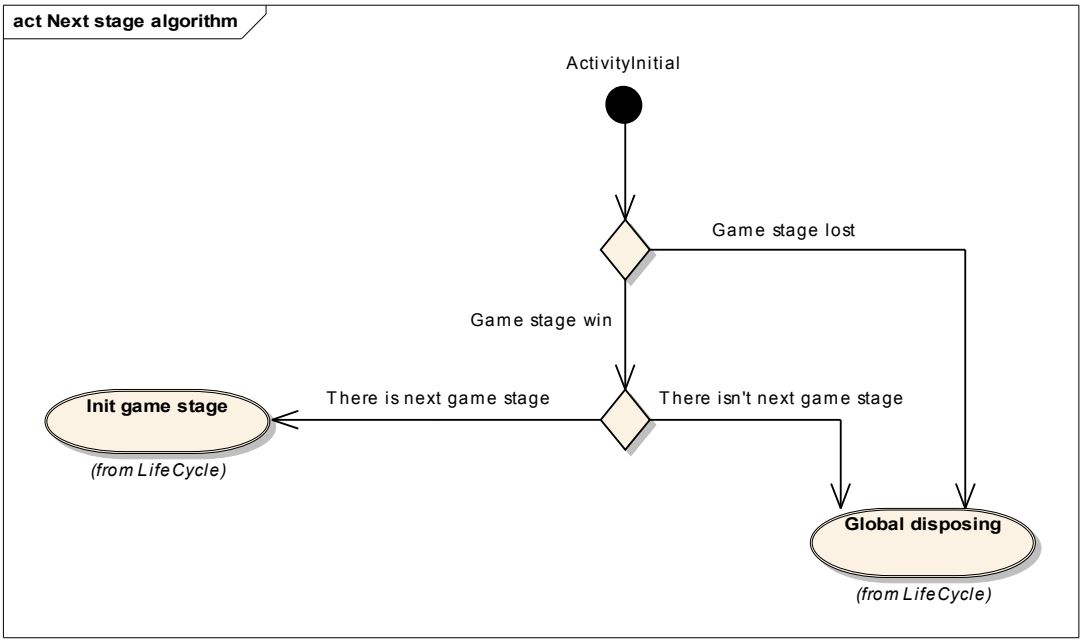
Данное состояние компонента достаточно условное и описывает его “неактивность”, однако при переходе в данное состояние и выходе из него, компонент может осуществлять некоторые действия, зависящие от характера игрового процесса (как пример фиксацию времени игрового таймера в играх на время). Игровой процесс, находящийся в данном состоянии, может быть прерван пользователем или контейнером, перейдя при этом в режим сохранения игровых данных (Save current state). При команде продолжения игрового процесса, компонент возвращается в состояние игры (Playing).

Состояние Save current state

Данное состояние компонента активизируется при принудительном прекращении игрового процесса для последующего восстановления и продолжения игры. В нем можно произвести фактически сериализацию компонента в поток, но это зависит от игры. Из данного состояния, компонент переходит в состояние освобождение ресурсов игрового уровня (Release Stage).

Состояние Release stage

В данном состоянии компонент освобождает ресурсы, занятые им для текущего игрового уровня. Если игра не была прервана, а была закончена согласно внутренним бизнес-процессам и имеет много игровых уровней, то проверяется условие перехода на следующий уровень и компонент переводится в состояние инициализации следующего игрового уровня (Init game stage), но если игра была прервана, то осуществляется переход в состояние глобального освобождения ресурсов (Global disposing).



Состояние Global disposing

Данное состояние является завершающим в работе компонента, в нем происходит полная деинициализация и освобождение ресурсов. По выходе из данного состояния, компонент не может продолжать дальнейшее функционирование.

Сохранение-загрузка состояния компонента

Данные операции, компонент должен производить без привязки к аппаратным особенностям платформы, из-за чего вся нагрузка ложится на контейнер, который просто передает InputStream компоненту для загрузки состояния и OutputStream для выгрузки оного.

Все вопросы о месте хранения и способах кодирования, решает контейнер. В целом, данный подход позволяет организовать как хранение одного состояния, так и много состояний, но в условиях мобильных и телевизионных устройств, крайне сомнительно, что пользователь будет иметь интерес к многоуровневому хранению игровых состояний, но в большей степени будет заинтересован в бытром прекращении игры и в продолжении при повторной загрузки прерванного процесса.

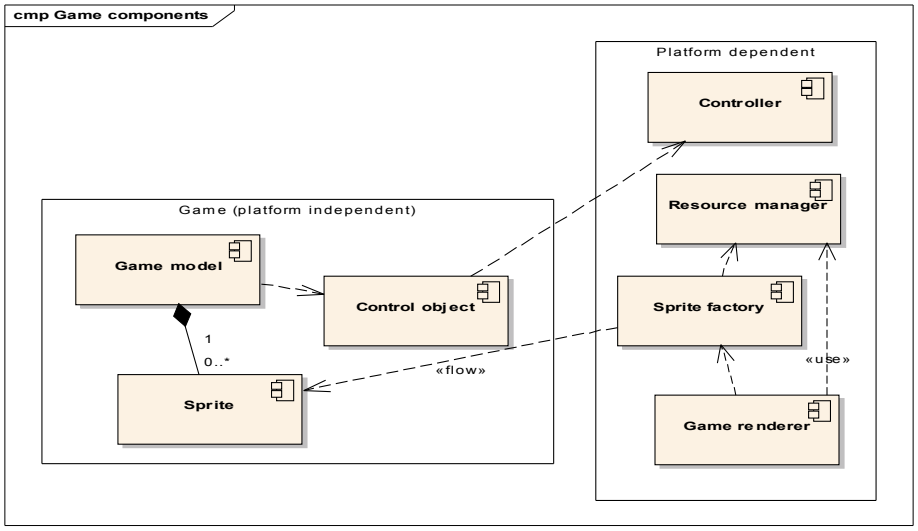
Внутриигровая координатная система

Поскольку игровой компонент описывает только бизнес-процессы игры, т.е. является её математической моделью и реализует бизнес-процессы в ней протекающие, то он никаким образом не соотносится с платформой, на которой запущен. Так как игровой процесс, как правило, является визуальным, а устройства на которых он может быть запущен, имеют разное графическое разрешение, то тут требуется подход, который обеспечит безошибочное и быстрое выполнение одного и того же компонента (!) на платформах с разными устройствами вывода. Для этого предлагается использовать механизм фиксированной точки 16 бит, что позволит отображать игровой процесс на устройствах с маленьким и большим разрешением экрана, без потери точности игрового процесса. Как эталонное, в этом случае, выбирается устройство со средним или наибольшим графическим разрешением из запланированных. Так же в этом формате отрабатываются и все внутреигровые размерности объектов. Процесс настройки игрового компонента под разрешение целевой платформы может осуществляться как в runtime (путем передачи коэффициентов масштабирования), так и в процессе компиляции (путем настройки статических констант).

Использование механизма фиксированной точки, позволит использовать наиболее быстрый механизм Java, а именно работу с целыми числами 32 битной разрядности (int), так же это обеспечит совместимость игрового компонента со всеми видами Java, от ME до SE.

Абстрактные сущности процесса

Для полноценной реализации игрового процесса, требуется не только компонент реализующий бизнес-логику, но и вспомогательные компоненты, облегчающие процесс написания.



Для полноценного построения абстрактной математической модели визуальной игры, предлагается ввести следующие платформеннонезависимые сущности:

- ***Game model*** – игровой компонент, реализующий бизнес-логику игрового процесса
- ***Control object*** – контейнер данных, осуществляющий хранение и передачу данных от контроллера в игровую модель.
- ***Sprite*** – спрайт, функционирующий в пространстве игры и представляющий собой прямоугольную координатную областьс неким графическим содержимым, так же данный компонент содержит методы, для обработки взаимодействия с другими спрайтами.

Для взаимодействия с аппаратными слоями, для каждой платформы будут имплементированы следующие компоненты:

- ***Controller*** – основной объект, в нашем случае осуществляет прием команд от пользователя и может являться контейнером, осуществляющим и контроллирующим жизненный цикл Game model. Зачастую, контроллер может являться экземпляром основного стартового класса для заданной платформы (MIDlet, Xlet, Applet и т.д.)
- ***Sprite factory*** – фабрика, которая позволяет игровому компоненту создавать спрайты для последующего использования в игровом процессе. Такое централизованное генерирование спрайтов, позволяет оптимизировать загрузку и хранение ресурсов, связанных со спрайтами, а так же позволяет изолировать программиста от особенностей реализации механизма спрайтов для конкретной платформы.
- ***Resource manager*** – компонент, осуществляющий работу с системными ресурсами, их хранение, загрузку и выгрузку (если требуется). Может быть связан со Sprite factory если их совместная работа приведет к оптимизации загрузки игровых ресурсов.
- ***Game renderer*** – компонент, осуществляющий отображение игровых объектов представленных спрайтами и переводящий их координаты и взаимное расположение из абстрактных игровых координат в физическое представление, базирующееся на возможностях устройства отображения, присущего данной платформе. Данный компонент активно взаимодействует с Resource manager, который хранит графические ресурсы.

Интерфейсы и абстрактные классы фреймворка

Класс Gamelet

Данный абстрактный класс должен быть унаследован игровым компонентом, реализующим игровую логику. Программист должен имплементировать абстрактные функции данного класса и разместить в них игровую логику.

```
package tv.zodiac;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.util.Random;

/**
 * Класс описывает игровой геймлет
 *
 * @author Igor A. Maznitsa
 */
```

```
public abstract class Gamelet
{
//----------------Раздел констант-----------------------
    /**
     * Флаг, показывающий, что игра может сохранять свои игровые состояния и загружать их
     */
    public static final int FLAG_CANBESAVED = 0x01;

    /**
     * Флаг, показывающий, что игра разрешает перезапускать игровые фазы
     */
    public static final int FLAG_STAGECANBERESTARTED = 0x02;

    /**
     * Флаг, показывающий, что игра имеет много игровых уровней-состояний
     */
    public static final int FLAG_MANYSTAGES = 0x04;

    /**
     * Флаг, показывающий, что проигрыш игрока на уровне не прекращает игру
     */
    public static final int FLAG_STAGESCONTINUEAFTERLOST = 0x08;

    /**
     * Флаг, показывает, что геймлет в неинициализированном состоянии
     */
    public static final int STATE_UNINITED = 0;

    /**
     * Флаг, показывает, что геймлет в инициализированном состоянии
     */
    public static final int STATE_INITED = 1;

    /**
     * Флаг, показывает, что геймлет в состоянии инициализированной игровой сессии
     */
    public static final int STATE_SESSIONINITED = 2;

    /**
     * Флаг, показывает, что геймлет в состоянии инициализированной игровой фазы
     */
    public static final int STATE_STAGEINITED = 3;

    /**
     * Флаг, показывает, что геймлет в состоянии инициализированной но поставленной на паузу игровой фазы
     */
    public static final int STATE_STAGEPAUSED = 4;

    /**
     * Флаг, показывает, что игра в рабочем состоянии
     */
    public static final int GAMESTATE_PLAYED = 0;

    /**
     * Флаг, показывает, что игра в законченном состоянии с выигрышем игрока
     */
    public static final int GAMESTATE_PLAYERWIN = 1;

    /**
     * Флаг, показывает, что игра в законченном состоянии с проигрышем игрока
     */
    public static final int GAMESTATE_PLAYERLOST = 2;

    /**
     * Флаг, показывает, что игра в законченном состоянии, была "ничья"
     */
    public static final int GAMESTATE_DRAWGAME = 3;

//-----------------------------------------------------

    /**
     * Переменная хранит показатель текущего состояния геймлета
     */
    protected int i_State;

    /**
     * Переменная хранит показатель текущего состояния игрового процесса
     */
    protected int i_GameState;

    /**
```

```
     * Переменная хранит идентификатор текущей игровой сессии
     */
    protected int i_SessionID;

    /**
     * Переменная хранит идентификатор текущей игровой фазы
     */
    protected int i_StageID;


    /**
     * Функция позволяет получить текущее значение состояния геймлета
     *
     * @return текущее состояние Gamelet как int
     */
    public final int getState()
    {
        return i_State;
    }

    /**
     * Функция позволяет получить текущий идентификатор игрового уровня
     *
     * @return идентификатор игрового уровня как int
     */
    public final int getStageID()
    {
        return i_StageID;
    }

    /**
     * Функция позволяет получить текущий идентификатор игровой сессии
     *
     * @return идентификатор игровой сессии как int
     */
    public final int getSessionID()
    {
        return i_SessionID;
    }

    /**
     * Функция позволяет получить текущее состояние игрового процесса
     *
     * @return состояние игрового процесса как int
     */
    public final int getGameState()
    {
        return i_GameState;
    }

    /**
     * Инициализация геймлета, до вызова данной функции, запрещены любые операции с геймлетом
     *
     * @param _parent контейнер для данного геймлета
     * @throws Throwable порождается если была ошибка инициализации или геймлет не в состоянии STATE_UNINITED
     */
    protected final void init(GameContainer _parent) throws Throwable
    {
        if (i_State != STATE_UNINITED)
            throw new Throwable();
        _cInit(_parent);
        i_State = STATE_INITED;
    }

    /**
     * Инициализация игровой сессии
     *
     * @param _parent    игровой контейнер геймлета
     * @param _sessionID уникальный идентификатор инициализируемой игровой сессии
     * @throws Throwable порождаетсе если проблемы с инициализацией сессии или геймлет в состоянии, в котором нельзя
производить данную операцию
     */
    protected final void initSession(GameContainer _parent, int _sessionID) throws Throwable
    {
        if (i_State != STATE_INITED) throw new Throwable("" + i_State);
        i_SessionID = _sessionID;
        _cInitSession(_parent, _sessionID);
        i_State = STATE_SESSIONINITED;
        i_GameState = GAMESTATE_PLAYED;
    }
```

```
    /**
     * Перезапуск игровой сессии по команде игрока
     *
     * @throws Throwable порождается если состояние геймлета не позволяет осуществлять перезапуск или геймлет не
поддерживает данную функцию
     */
    protected final void restartCurrentStage() throws Throwable
    {
        if ((getGameFlags() & FLAG_STAGECANBERESTARTED) == 0)
            throw new Throwable();
        switch (i_State)
        {
            case STATE_STAGEINITED:
            case STATE_STAGEPAUSED:
            {
                _cRestartCurrentStage();
            }
            ;
            break;
            default:
                throw new Throwable();
        }
    }

    /**
     * Инициализация игровой фазы
     *
     * @param _parent  игровой контейнер геймлета
     * @param _stageID уникальный идентификатор игровой фазы
     * @throws Throwable порождается, если нет возможности инициализировать фазу или геймлет в состоянии в котором
нельзя производить данную операцию
     * @return true если игра имеет игровую фазу и false если игра не имеет заданную игровую фазу
     */
    protected final boolean initStage(GameContainer _parent, int _stageID) throws Throwable
    {
        if (i_State != STATE_SESSIONINITED)
            throw new Throwable();
        i_StageID = _stageID;
        if (_cInitStage(_parent, _stageID))
        {
            i_State = STATE_STAGEINITED;
            i_GameState = GAMESTATE_PLAYED;
            return true;
        }
        else
            return false;
    }

    /**
     * Возвращает размер блока данных, требуемых для сохранения данных
     *
     * @return количество байтов, требуемых для сохранения состояния
     */
    protected final int getDataBlockSzie()
    {
        int i_size = 8;
        i_size += _cGetDataBlockSize();
        return i_size;
    }

    /**
     * Записывает игровое состояние геймлета
     *
     * @param _outStream поток, в который производится выгрузка данных
     * @throws Throwable порождается если произошла ошибка или геймлет в неправильном состоянии
     */
    protected final void saveState(DataOutputStream _outStream) throws Throwable
    {
        switch (i_State)
        {
            case STATE_STAGEINITED:
            case STATE_STAGEPAUSED:
            {

            }
            ;
            break;
            default:
                throw new Throwable();
        }
```

```
        _outStream.writeInt(i_SessionID);
        _outStream.writeInt(i_StageID);

        _cSaveStage(_outStream);
    }

    /**
     * Загружает игровое состояние геймлета
     *
     * @param _parent   класс-родитель геймлета
     * @param _inStream поток, из которого производится загрузка данных
     * @throws Throwable порождается если произошла ошибка или геймлет в неправильном состоянии
     */
    protected final void loadState(Class _parent, DataInputStream _inStream) throws Throwable
    {
        final int i_sessionID = _inStream.readInt();
        final int i_stageID = _inStream.readInt();

        switch (i_State)
        {
            case STATE_INITED:
            {
                initSession(_parent, i_sessionID);
                initStage(_parent, i_stageID);
            }
            ;
            break;
            case STATE_SESSIONINITED:
            {
                disposeSession();
                initSession(_parent, i_sessionID);
                initStage(_parent, i_stageID);
            }
            ;
            break;
            case STATE_STAGEPAUSED:
            case STATE_STAGEINITED:
            {
                disposeStage();
                disposeSession();
                initSession(_parent, i_sessionID);
                initStage(_parent, i_stageID);
            }
            ;
            break;
            default:
                throw new Throwable();
        }
        _cLoadStage(_inStream);
    }

    /**
     * Дает команду геймлету поставить игровой процесс на паузу
     *
     * @throws Throwable порождается если произошла ошибка или геймлет в неправильном состоянии
     */
    protected final void pause() throws Throwable
    {
        switch (i_State)
        {
            case STATE_STAGEINITED:
            {
                i_State = STATE_STAGEPAUSED;
                _cPauseStage();
            }
            ;
            break;
            case STATE_STAGEPAUSED:
                return;
            default:
                throw new Throwable();
        }
    }

    /**
     * Дает команду геймлету выйти с режима пауза в нормальный режим
     *
     * @throws Throwable порождается если произошла ошибка или геймлет в неправильном состоянии
     */
    protected final void resume() throws Throwable
    {
```

```
        switch (i_State)
        {
            case STATE_STAGEPAUSED:
            {
                i_State = STATE_STAGEINITED;
                _cResumeStage();
            }
            ;
            break;
            case STATE_STAGEINITED:
                return;
            default:
                throw new Throwable();
        }
    }

    /**
     * Освобождает ресурсы игровой фазы и переводит геймлет в состояние инициализированной игровой сессии
     *
     * @throws Throwable порождается если произошла ошибка или геймлет в неправильном состоянии
     */
    protected final void disposeStage() throws Throwable
    {
        switch (i_State)
        {
            case STATE_STAGEPAUSED:
            case STATE_STAGEINITED:
            {
                i_State = STATE_SESSIONINITED;
                _cDisposeStage();
            }
            ;
            break;
            default:
            {
                throw new Throwable();
            }
        }
    }

    /**
     * Освобождает ресурсы игровой сессии и переводит геймлет в инициализированное состояние
     *
     * @throws Throwable порождается если произошла ошибка или геймлет в неправильном состоянии
     */
    protected final void disposeSession() throws Throwable
    {
        switch (i_State)
        {
            case STATE_STAGEINITED:
            case STATE_STAGEPAUSED:
            {
                disposeStage();
            }
            case STATE_SESSIONINITED:
            {
                i_State = STATE_INITED;
                _cDisposeSession();
            }
            ;
            break;
            default:
                throw new Throwable();
        }
    }

    /**
     * Освобождает ресурсы занятые геймлетом и переводит геймлет в неинициализированное состояние
     *
     * @throws Throwable порождается если произошла ошибка или геймлет в "неправильном" состоянии
     */
    protected final void release() throws Throwable
    {
        switch (i_State)
        {
            case STATE_UNINITED:
                throw new Throwable();
            case STATE_INITED:
            {
                i_State = STATE_UNINITED;
                _cRelease();
```

```
            }
            ;
            break;
            case STATE_SESSIONINITED:
            {
                disposeSession();
                i_State = STATE_UNINITED;
                _cRelease();
            }
            ;
            break;
            case STATE_STAGEPAUSED:
            case STATE_STAGEINITED:
            {
                disposeStage();
                disposeSession();
            }
            ;
            break;
        }
    }

    /**
     * Генератор псевдослучайных чисел
     */
    protected final Random p_RNDGen = new Random(System.currentTimeMillis());

    /**
     * Коэффициент для генератора случайных чисел
     */
    protected final int RND_CONST = 0x58E312A7;

    /**
     * Текущее значение генератора случайных чисел
     */
    protected int i_RndValue = (int) (System.currentTimeMillis() & 0x7FFFFFFF);


    /**
     * Функция генерирует и возвращает псевдослучайное числовое значение в заданном пределе (включительно).
     *
     * @param _limit предел генерируемого числового значения (включительно)
     * @return сгенерированное псевдослучайное значение int в заданном пределе.
     */
    public final int getRndInt(int _limit)
    {
        i_RndValue = ((i_RndValue ^ RND_CONST) * RND_CONST) % 0x7FFFFFFF;
        _limit++;
        _limit = (int) (((long) i_RndValue * (long) _limit) >>> 31);
        return _limit;
    }

    /**
     * Обработка игровой итерации
     * @param _container ссылка на игровой контейнер
     * @param _controlObject объект, содержащий информацию, контроллирующую игровой процесс
     * @return возвращет текущее состояние игрового процесса
     * @throws Throwable порождается если было исключение в процессе отработки итерации
     */
    public abstract int processIteration(GameConainer _container, ControlObject _controlObject) throws Throwable;

    /**
     * Возвращает текущие игровые очки игрока
     *
     * @return игровые очки игрока как int
     */
    public abstract int getPlayerScore();

    /**
     * Обработка инициализации геймлета
     *
     * @param _parent контейнер геймлета
     * @throws Throwable порождается если было невозможно произвести инициализацию геймлета
     */
    public abstract void _cInit(GameContainer _parent) throws Throwable;

    /**
     * Обработка инициализации игровой фазы
     *
     * @param _parent  игровой контейнер геймлета
     * @param _stageID уникальный идентификатор фазы
```

```
     * @throws Throwable порождается если было невозможно произвести инициализацию фазы
     * @return true если имеется игровая фаза и false если игра больше не имеет игровых фаз
     */
    public abstract boolean _cInitStage(GameContainer _parent, int _stageID) throws Throwable;

    /**
     * Обработка инициализации игровой сессии
     *
     * @param _parent    игровой контейнер геймлета
     * @param _sessionID уникальный идентификатор сессии
     * @throws Throwable порождается если было невозможно произвести инициализацию сессии
     */
    public abstract void _cInitSession(GameContainer _parent, int _sessionID) throws Throwable;

    /**
     * Обработка деинициализации текущей игровой сессии
     */
    public abstract void _cDisposeSession();

    /**
     * Обработка постановки игрового процесса на паузу
     */
    public abstract void _cPauseStage();

    /**
     * Обработка перехода игрового процесса из состояния паузы в состояние работы
     */
    public abstract void _cResumeStage();

    /**
     * Обработка деинициализации текущей фазы игры
     */
    public abstract void _cDisposeStage();

    /**
     * Обработка глобальной деинициализации геймлета
     */
    public abstract void _cRelease();

    /**
     * Обработка перезапуска текущего игрового уровня по запросу игрока
     */
    public abstract void _cRestartCurrentStage();

    /**
     * Запрос на размер игровых данных в байтах, требуемый для сохранения состояния игровой фазы
     *
     * @return размер блока в байтах
     */
    public abstract int _cGetDataBlockSize();

    /**
     * Загрузка игрового состояния из потока
     *
     * @throws Throwable генерируется если произвошла ошибка загрузки
     */
    public abstract void _cLoadStage(DataInputStream _inStream) throws Throwable;

    /**
     * Запись игрового состояния в поток
     *
     * @throws Throwable генерируется если произошла ошибка записи
     */
    public abstract void _cSaveStage(DataOutputStream _outStream) throws Throwable;

    /**
     * Функцмя возвращает комбинацию флагов, позволяющих контейнеру понять возможности и потребности игрового процесса
     *
     * @return комбинация флагов как int
     */
    public abstract int getGameFlags();

    /**
     * Функция возвращает строковый уникальный идентификатор игры
     *
     * @return строковый идентификатор как String, не может быть null
     */
    public abstract String getGameID();

    /**
     * Возвращает по запросу контейнера время в миллисекундах между игровыми итерациями. Опрашивается перед каждой
```

```
итерацией.
     *
     * @return возвращает время между итерациями в миллисекундах
     */
    public abstract long getPrefferedTimeDelayBetweenIterations();

    /**
     * Возвращает массив спрайтов, в котором они хранятся в порядке от самого дальнего (0 индекс) до самого ближнего
(последний индекс).
     * @return массив объектов Sprite, который может содержать null ячейки
     */
    public abstract Sprite[] getSpritesToShow();

    /**
     * Функция вызывается при изменении в наблюдаемом спрайте
     * @param _sprite указатель на наблюдаемый спрайт, который был изменен
     */
    public abstract void onSpriteChanged(Sprite _sprite);
}
```

Класс ControlObject

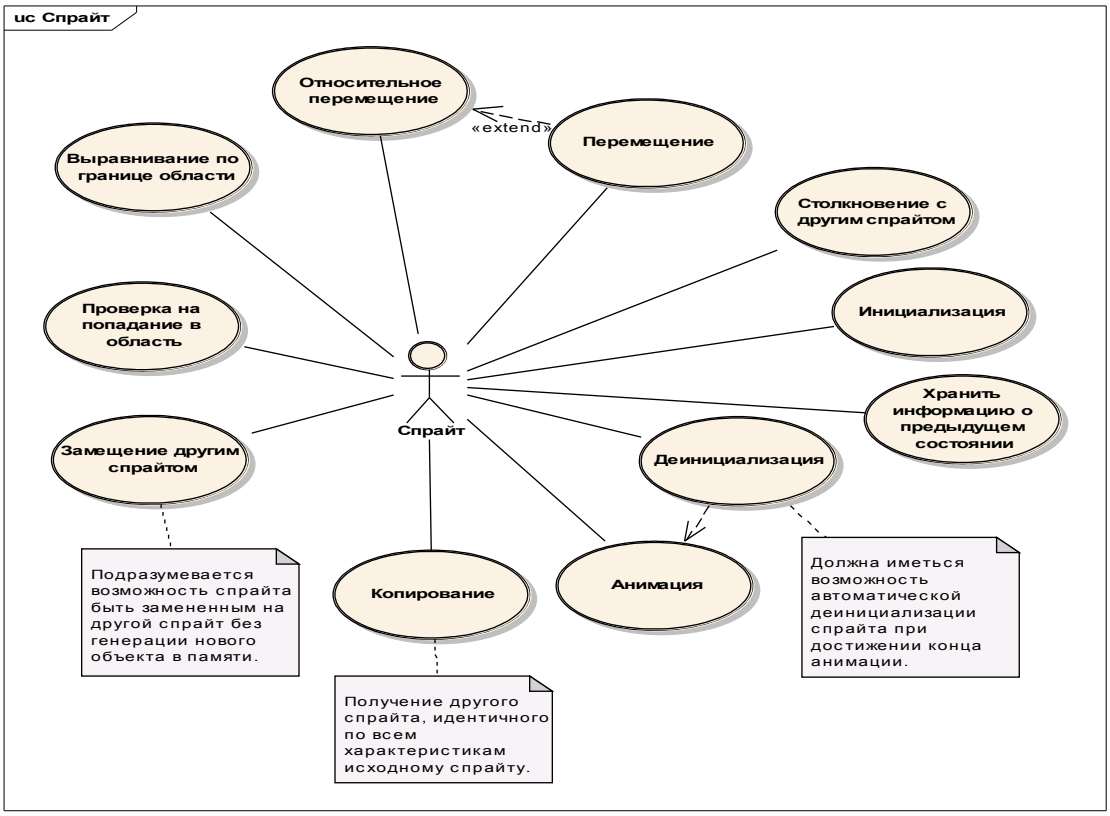
Объект данного класса, используется Gamelet для получения состояния управляющих кнопок.

```
package tv.zodiac;

/**
 * Класс описывает объект, содержащий флаги нажатых управляющих кнопок
 *
 * @author Igor Maznitsa
 */
public class ControlObject
{
    /**
     * Состояние когда все кнопки не нажаты
     */
    public static final int BUTTON_NONE = 0;

    /**
     * Нажата кнопка манипулятора "ВЛЕВО"
     */
    public static final int BUTTON_LEFT = 1;

    /**
     * Нажата кнопка манипулятора "ВПРАВО"
     */
    public static final int BUTTON_RIGHT = 2;

    /**
     * Нажата кнопка манипулятора "ВВЕРХ"
     */
    public static final int BUTTON_UP = 4;

    /**
     * Нажата кнопка манипулятора "ВНИЗ"
     */
    public static final int BUTTON_DOWN = 8;

    /**
     * Нажата кнопка манипулятора "ОГОНЬ"
     */
    public static final int BUTTON_FIRE = 16;

    /**
     * Нажата кнопка манипулятора "Сервисная кнопка 1"
     */
    public static final int BUTTON_SERVICE1 = 32;

    /**
     * Нажата кнопка манипулятора "Сервисная кнопка 2"
     */
    public static final int BUTTON_SERVICE2 = 64;

    /**
     * Нажата кнопка манипулятора "Сервисная кнопка 3"
```

```
     */
    public static final int BUTTON_SERVICE3 = 128;

    /**
     * Нажата кнопка манипулятора "Сервисная кнопка 4"
     */
    public static final int BUTTON_SERVICE4 = 256;


    /**
     * Переменная содержит флаги нажатых кнопок
     */
    protected int i_PressedButtons;

    /**
     * Функция возвращает состояние кнопок
     *
     * @return флаги нажатых кнопок как int
     */
    public int getPressedButtons()
    {
        return i_PressedButtons;
    }

    /**
     * Функция осуществляет сброс всех флагов нажатых кнопок
     */
    public void reset()
    {
        i_PressedButtons = BUTTON_NONE;
    }
}
```

Интерфейс Sprite

Так как имеется много вариантов имплементации спрайтов и работы с ними (в том числе и “аппаратная” реализация на некоторых платформах), то предлагается не делать жесткую реализацию, а написать интерфейс, который должен имплементироваться разработчиком и через него он должен работать со спрайтами в игровом процессе. Так же это позволяет портировать существующие игры с возникновением меньшего количества несостыковок с существующими внутриигровыми механизмами.



Спрайт рассматривается как прямоугольная зона, которая содержит другую вложенную прямоугольную зону (hotspot), которая используется для проверок взаимодействия спрайта с другими спрайтами. Все координатные отсчеты в спрайте идут от так называемой главной точки (main point). Её наличие позволяет более точно ориентировать спрайты друг относительно друга при отрисовке и наложении, а так же при изменении типа спрайта. Например - если у нас взрывается космический корабль неправильной формы, то мы имеем возможность разместить главную точку на месте топливного бака, а у спрайта взрыва главную точку в эпицентре и при наложении спрайтов получится, что взрыв будет рисоваться на изображении корабля с эпицентром на баке с горючим. Так же главная точка может использоваться для выравнивания спрайта по границам прямоугольной области, например при реализации шарика летающего внутри прямоугольника с разрешением выхода границ шарика за его пределы.

Для описания заданной функциональности приводится интерфейс Sprite. Хотелось бы отметить, что в нем используются координаты с фиксированной точкой 16 бит, что позволяет спрайту меньше привязываться к графическому пространству конкретного устройства. Это позволяет сделать процесс более переносимым между платформами.

```
import java.io.*;

public interface Sprite
{
    /**
     * Возвращает ширину спрайта в фиксированной точке 16 бит
     */
    public int getWidth();

    /**
     * Возвращает высоту спрайта в фиксированной точке 16 бит
     */
    public int getHeight();
```

```
    /**
     * Возвращает ширину "горячей зоны" спрайта в фиксированной точке 16 бит
     */
    public int getHospotWidth();

    /**
     * Возвращает высоту "горячей зоны" спрайта в фиксированной точке 16 бит
     */
    public int getHospotHeight();

    /**
     * Возвращает смещение "горячей зоны" спрайта по X в фиксированной точке 16 бит от верхнего левого края
     */
    public int getHospotX();

    /**
     * Возвращает смещение "горячей зоны" спрайта по Y в фиксированной точке 16 бит от верхнего левого края
     */
    public int getHospotY();

    /**
     * Возвращает координату X верхней левой точки спрайта в фиксированной точке 16 бит
     */
    public int getTopLeftX();

    /**
     * Возвращает координату Y верхней левой точки спрайта в фиксированной точке 16 бит
     */
    public int getTopLeftY();

    /**
     * Возвращает координату X верхней левой точки спрайта, отражающую предыдущее положение спрайта (до последнего
изменения) в фиксированной точке 16 бит
     */
    public int getPrevTopLeftX();

    /**
     * Возвращает координату Y верхней левой точки спрайта, отражающую предыдущее положение спрайта (до последнего
изменения) в фиксированной точке 16 бит
     */
    public int getPrevTopLeftY();

    /**
     * Возвращает координату X точки привязки спрайта в фиксированной точке 16 бит
     */
    public int getMainX();

    /**
     * Возвращает координату Y точки привязки спрайта в фиксированной точке 16 бит
     */
    public int getMainY();

    /**
     * Функция возвращает флаг, показывающий, что анимационная последовательность спрайта стоит на первом фрейме.
     * @return true если активен первый фрейм в анимационной последовательности, false если нет
     */
    public boolean isFirstFrame();

    /**
     * Функция возвращает флаг, показывающий, что анимационная последовательность спрайта стоит на последнем фрейме
     * @return true если активен последний фрейм, false если нет
     */
    public boolean isLastFrame();

    /**
     * Функция возвращает количество фреймов в спрайте
     * @return количество фреймов в спрайте как int
     */
    public int getFramesNumber();

    /**
     * Функция возвращает значение текущего фрейма анимации спрайта
     * @return номер текущего кадра анимации спрайта как int (0 кадр - первый)
     */
    public int getCurrentFrameNumber();

    /**
     * Функция задает значение текущего фрейма анимации спрайта
     * @param _frameNumber задаваемый номер текущего кадра анимации спрайта как int (0 кадр - первый)
     */
```

```
    public void setCurrentFrameNumber(int _frameNumber);

    /**
     * Запись состояния спрайта в поток
     */
    public void saveToStream(DataOutputStream _outStream) throws Throwable;

    /**
     * Восстановление состояния спрайта из потока
     */
    public void loadFromStream(DataInputStream _inStream) throws Throwable;

    /**
     * Выравнивание спрайта по главной точке другого спрайта (совмещение главных точек)
     */
    public void alignMainPoint(Sprite _sprite);

    /**
     * Возвращает true если спрайт на паузе
     */
    public boolean isPaused();

    /**
     * Устанавливает или снимает режим пацзы для спрайта (пауза приостанавливает процесс анимации для спрайта)
     */
    public void setPause(boolean _pause);

    /**
     * Возвращает идентификатор типа спрайта (например SPACE_SHIP)
     */
    public int getType();

    /**
     * Возвращает идентификатор состояние спрайта (например SPACE_SHIP_DESTROYED)
     */
    public int getState();

    /**
     * Проверяет на столкновение горячей зоны спрайта с горячей зоной другого спрайта и если есть столкновение то true
     */
    public boolean checkCollision(Sprite _sprite);

    /**
     * Возвращает опциональные данные, хранимые спрайтом (например количество энергии, патронов)
     */
    public int getOptionalData();

    /**
     * Устанавливает опциональные данные, хранимые спрайтом
     */
    public void setOptionalData(int _value);

    /**
     * Смещает главноую точку спрайта по координатам X и Y в значениях с фиксированной точкой 16 бит. При этом
пересчитываются координаты экранного отображения, горячей зоны и верхнего левого края
     */
    public void moveMainPointXY(int _i16deltaX, int _i16deltaY);

    /**
     * Выставляет главную точку спрайта в заданные координаты X и Y в значениях с фиксированной точкой 16 бит. При
этом пересчитываются координаты экранного отображения, горячей зоны и верхнего левого края
     */
    public void setMainPointXY(int _i16newMainX, int _i16newMainY);

    /**
     * Возвращает флаг отображеия спрайта на экране
     * @return true если спрайт должен отображаться и false если не должен
     */
    public boolean isShown()

    /**
     * Устанавливает флаг отображения спрайта, т.е. если этот флаг сброшен, значит спрайт не подлежит отрисовке
     * @param _flag флаг отображения спрайта, true если флаг выставляется и false если флаг сбрасывается
     */
    public void setShown(boolean _flag)

    /**
     * Выравниваем спрайт по границам области
     * @param _i16x1 координата X верхней левой точки границы (фикс. точка 16 бит)
     * @param _i16y1 координата Y верхней левой точки границы (фикс. точка 16 бит)
     * @param _i16x2 координата X нижней правой точки границы (фикс. точка 16 бит)
```

```
     * @param _i16y2 координата Y нижней правой точки границы (фикс. точка 16 бит)
     * @return true если было выравнивание, иначе false
     */
    public boolean alignToArea(int _i16x1, int _i16y1, int _i16x2, int _i16y2);

    /**
     * Деактивизация спрайта, после данной операции, спрайт не может быть использован.
     * После выполнения данной функции, спрайт может быть переинициализирован SpriteFactory для дальнейшего
использования.
     */
    public void release();

    /**
     * Возвращает флаг активности спрайта, что позволяет узнать деактивирован ли он через realise().
     * @return true если спрайт реализован, false если активен
     */
    public boolean isReleased();

    /**
     * Выставляет значение флага, показывающего что спрайт должен уведомлять о своем изменении Gamelet
     * @param _flag true если должен нотифицировать о себе наблюдателя и false если не должен
     */
    public void setObservable(boolean _flag);


    /**
     * Возвращает значение флага, наблюдаемости спрайта. По умолчанию false (не наблюдаемый).
     */
    public boolean isObservable();

}
```

Класс SpriteFactory

Программист не имеет непосредственной возможности к инициализации спрайта и для этого воодится класс SpriteFactory, позволяющий игре получать одиночные и групповые спрайты. Так как, тем самым, мы переводим процесс инициализации спрайтов в четко определенное место, то мы имеем возможность увязывать SpriteFactory с менеджером ресурсов и оптимизировать последним загрузку графических файлов, под затребованные типы спрайтов. Класс SpriteFactory зависит от платформы, на которой реализуется (может быть написан и общий, платформеннонезависимый) и позволяет оптимизировать процессы инициализации спрайтов, а так же изолировать программиста от непосредственного доступа к платформозависимым механизмам.

```
public abstract class SpriteFactory
{
    /**
     * Возвращает экземпляр фабрики спрайтов для заданного Gamelet
     * @param _gamelet указатель на Gamelet для которого требуется получить фабрику
     * @return объект-фабрика как SpriteFactory
     */
    public static SpriteFactory getSpriteFactory(Gamelet _gamelet)
    {
        ...
    }

    /**
     * Генерация одиночного спрайта заданного типа и состояния.
     * @param _type тип спрайта
     * @param _state состояние спрайта
     * @return объект Sprite
     */
    public abstract Sprite makeSprite(int _type,int _state);

    /**
     * Генерация массива спрайтов заданного типа и состояния.
     * @param _type  тип спрайтов
     * @param _state состояние спрайтов
     * @param _number количество спрайтов в массиве
     * @return массив содержащий инициализированные объекты типа Sprite
```

```
     */
    public abstract Sprite [] makeSprites(int _type,int _state,int _number);

    /**
     * Реинициализация спрайта. Позволяет повторно использовать существующий активный или реализованный спрайт.
     * @param _sprite Объект Sprite для реинициализации
     * @param _newType тип нового спрайта
     * @param _newState состояние нового спрайта
     * @param _saveFramePosition если true то у нового спрайта сохраняется номер спрайта от предыдущего иначе значение
инициализируется 0
     */
    public abstract void reinitSprite(Sprite _sprite, int _newType,int _newState, boolean _saveFramePosition);

    /**
     * Функция позволяет клонировать заданный спрайт
     * @param _source исходный спрайт
     * @return объект Sprite, обладающий всеми признаками исходного спрайта
     */
    public abstract Sprite cloneSprite(Sprint _source);

    /**
     * Функция делает попытку клонирования спрайта в заданный массив спрайтов.
     * Ищет в массиве деактивированный спрайт и замещает его на клон исходного спрайта.
     * @param _source исходный спрайт
     * @param _array рабочий массив спрайтов,
     * @return true если был найден реализованный спрайт и произведена его замена, false если не был найден
реализованный спрайт
     */
    public abstract boolean cloneSpriteToArray(Sprite _source,Sprite [] _array);
}
```

Интерфейс GameContainer

Данный интерфейс должен имплементироваться игровым контейнером и позволяет геймлету взаимодействовать с контейнером и через него с аппаратными частями платформы.

```
public interface GameContainer
{
      /**
       * Функция сигнализирует контейнеру о наступлении некоего события в игровом процессе
       * @param _actionID индекс события
       * @return опциональное значение возвращаемое контейнером. Может быть использовано для возврата резальтата
выбора пользователя.
       */
      public boolean processAction(int _actionID);

      /**
       * Функция для получения потока ввода из статического ресурса относящегося к игре (например карту уровня)
       * @param _resourceID строковый идентификатор ресурса
       * @return InputStream с данными из заданного ресурса если найден и null если ресурс не найден
       */
      public InputStream getStaticGameResource(String _resourceID);

}
```

Как пример, данный механизм может быть задействован для озвучки игры. При смерти игрока, игра вызывает container.processAction(PLAYER_DEAD), что вызовет проигрывание звука смерти.

Ограничения при разработке математической модели игры

При написании игровой модели (класс наследованный от Gamelet и реализующий бизнес-логику игрового процесса) , программист обязан руководствоваться нижеприведенными ограничениями и указаниями:

1. Запрещается использовать вещественные типы данных float и double
2. Запрещается вносить в игровую модель, реализующую бизнес-правила игрового

процесса, игровые меню, внутриигровые меню, информационные экраны, тексты подсказок, хелпы и т.п. не относящееся непосредственно к игровому процессу. Игровая модель должна реализовывать только бизнес-логику игрового процесса и все показы подобного рода данных, например хелп, из игрового процесса – недопустимы. Если программист хочет отобразить некий экран с текстовой информацией на каком то участке игры, то он может ввести некое игровое событие, скажем ACTION_SHOW_HELP_FOR_SCR_233 и получив его, контейнер игры приостановит оную и отобразит экран наилучшим образом для заданной платформы.

3. Игровой программист должен руководствоваться ограничениями и классами определяемыми спецификацией CLDC 1.0, для увеличения переносимости игрового процесса.
4. Запрещается в игровой модели и связанных с нею вспомогательных частях(участвующих в игровом процессе) использовать потоки. Игра является пассивным объектом и не должна содержать никаких активных частей!
5. Рекомендуется (но не обязывает) привязывать игровое время не к таймеру, а к внутриигровым тикам,так как это позволяет сделать игровой процесс менее зависимым от возможных задержек в игровом процессе по вине firmware устройства и общей производительности устройства.