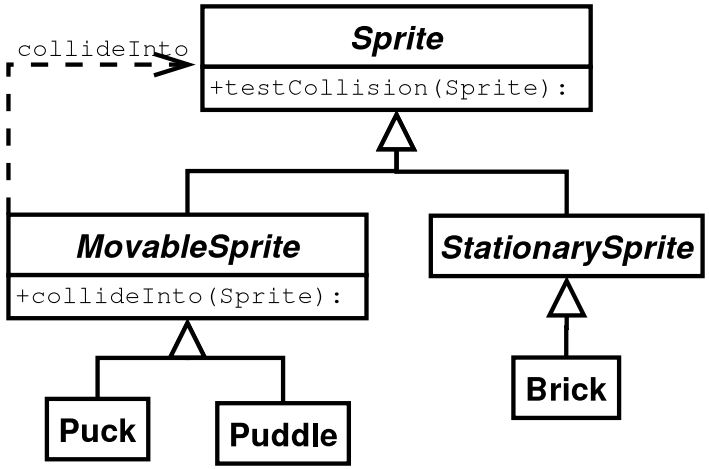

Рис. 8. Отношение зависимости

столкнуться с любым другим спрайтом. Данная зависимость представлена в виде метода `collideInto` класса `MovableSprite`, аргументом которого может быть любой спрайт, находящийся на игровом поле (рис. 8).

Диаграмма классов, изображённая на рис. 9, представляет статическую картину проекта, учитывающую связи между различными классами. Функции многих из них уже были описаны, и теперь необходимо пояснить назначение оставшихся.

`Match` — описывает сеанс игры, непосредственно связанный с игроком. Сеанс игры может быть начат `(start)`, приостановлен `(stop)` и возобновлён `(resume)`. Игрок может проиграть `(loose)` или выиграть `(win)`. С сеансом ассоциированы игровое поле, кирпичи и шайбы.

Рассмотрим более подробно назначение класса `PlayField` (игровое поле). На нём изображаются все игровые объекты, которые хранятся в специально созданном для этого классе — векторе спрайтов, отвечающем за обновление `(update)` и перерисовку `(draw)`. Под обновлением подразумевается любое изменения состояния спрайта, например, перемещение лопатки или разбивание кирпича шайбой, влекущее за собой его удаление с игрового поля.

Классы `PuckSupply` и `BrickPile` — контейнеры. Первый содержит шайбы, а второй хранит кирпичи и задаёт положение каждого из них на игровом поле. С помощью `PuckSupply` можно узнать о проигрыше игрока

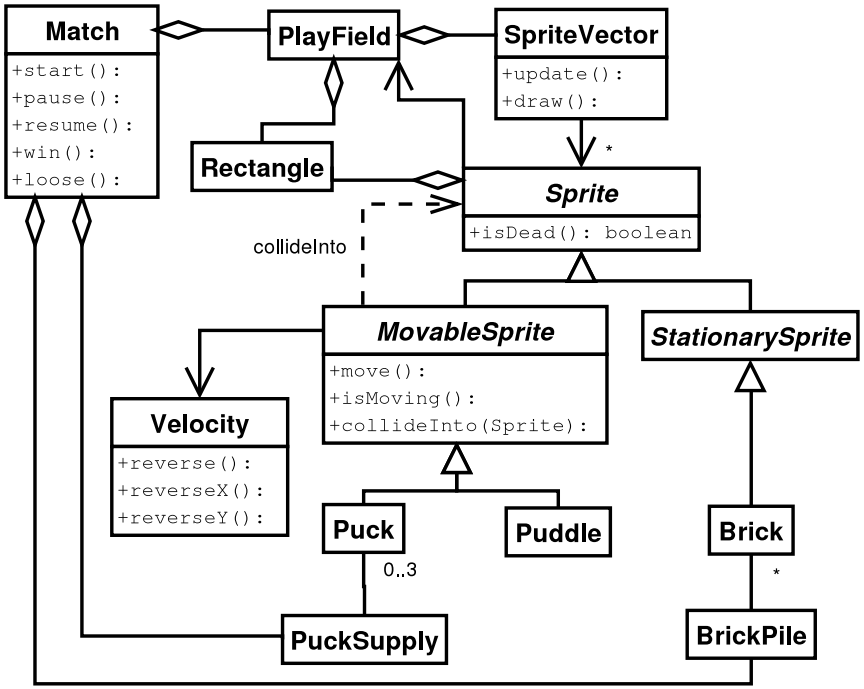

Рис. 9. Диаграмма классов

(все шайбы закончились), а `BrickPile` предоставляет информацию о выигрыше (все кирпичи разбиты).

Теперь расскажем более подробно о реализации класса `Sprite`. Положение и размеры спрайта задаются атрибутом `_pos` типа `Rectangle` (прямоугольник). Пересечение двух спрайтов `testCollision` находится как пересечение двух принадлежащих им прямоугольников-позиций[3]. Метод `isDead` отражает состояние спрайта: «жив» или «мёртв». «Мёртвый» спрайт удаляется с игрового поля.

```
abstract class Sprite {
  /*
   @_img - изображение спрайта
```

[3]Такой способ называется простой прямоугольной коллизией. Существуют и более совершенные методы, основанные на данных изображения спрайта.

выделяются все хранимые шайбы, а для шайбы выделяется экземпляр хранилища.

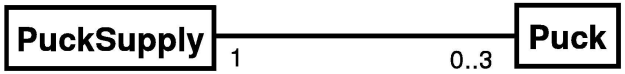

Рис. 6. Пример ассоциации

При графическом изображении ассоциации каждый из двух концов линии, её изображающей, иногда снабжают некоторой меткой, называемой *именем роли*. Ещё одним параметром, характеризующим классы, связанные ассоциацией, является *кратность*, которая указывает количество объектов, участвующих в данном отношении. В нашем проекте количество шайб в хранилище может принимать значения от нуля до трёх, что и показано на рисунке. Символ «*», использованный в качестве кратности, означает диапазон от нуля до бесконечности. Например, на игровом поле может находиться произвольное количество кирпичей.

Вот несколько других примеров ассоциаций с различными кратностями:

- у европейской жены один муж, у которого одна жена (отношение «один к одному»);
- у восточной жены один муж, у которого может быть произвольное число жён (отношение «один ко многим»);
- человек может посещать сколько угодно зданий, в каждом из которых может находиться произвольное число людей (отношение «многие ко многим»).

Наследование — это отношение, при котором один класс разделяет структуру и поведение другого класса (простое наследование) или сразу нескольких классов (множественное наследование). Новый класс называется *выведенным*, *производным* или *подклассом*. «Родительский» класс принято называть *базовым* или *суперклассом*. Совокупность классов, каждый элемент которой непосредственно или косвенно наследует свойства какого-то одного конкретного класса (*корневого*), называется *иерархией наследования*.

Построим иерархию наследования нашего проекта, корнем которой будет класс `Sprite`. Спрайт — это абстракция, представляющая любой объект, который может появиться на игровом поле (слово «спрайт» имеет исторические корни и является «стандартом» для аркадных игр). Его атрибутами являются: изображение (визуальный образ), положение на игровом поле и размеры (ширина и высота).


```
    _prevPos = _pos;
    _pos.translate(_v.getSpeedX(), _v.getSpeedY());
    /* Обработка соударения  со стенами, полом и потолком. */
    if (_pos.x <= b.x) {
      _pos.x = b.x;
      _v.reverseX();
    } else if (_pos.x + _pos.width >= b.width + b.x) {
      _pos.x = b.x + b.width - _pos.width;
      _v.reverseX();
    } else if (_pos.y <= b.y) {
      _pos.y = b.y;
      _v.reverseY();
    } else if (_pos.y + _pos.height > b.y + b.height) {
      /* Шайба упала на пол */
      _isDead = true;
       if (_ps.size() == 0) {
       _pf.getMatch().loose();
      } else {
       _pf.addSprite(_ps.get());
      }
    }
    /* Обработка соударения с другими спрайтами. */
    if (collideWith() != null) {
      _pos = _prevPos;
      collideInto(collideWith());
    }
  }
  /* Реакция на возникновение коллизии. */
  public void collideInto(Sprite s) {
    s.hitBy(this);
  }
  ...
}
```

Класс `Velocity` отвечает за движение подвижного спрайта по полю; его компонентами являются абсолютная величина скорости и направление движения. Величина скорости задаётся в некоторых условных единицах, а направление — в градусах (при этом 0 означает движение на восток, 90 — на юг и т. д., см. рис. 10).

Атрибут «скорость» разделён на две составляющих: `dx` и `dy` — проекции скорости на горизонтальную и вертикальную оси соответственно. В то время как величина скорости всегда неотрицательна, её составляющие могут принимать произвольные значения.

```
class Velocity {
```

Объект — операционная категория, которая содержит значения данных (атрибуты) и программный код, манипулирующий этими данными (методы). Объект является основой объектно-ориентированной программы, которую можно рассматривать как совокупность объектов. В процессе выполнения программы объекты создаются, взаимодействуют друг с другом, изменяются и, наконец, уничтожаются.

В игре «Кирпичики» имеется целый ряд различных объектов: лопатка, шайбы, кирпичи, игровое поле и его границы (стены, потолок, пол),— со своими наборами атрибутов и методов. Объект «шайба», например, инкапсулирует такие атрибуты, как размер, форма, положение на игровом поле и текущая скорость, а среди его методов содержатся операции перемещения шайбы и её удаления с игрового поля в случае падения шайбы на пол.

Метод — запрос на выполнение определённой операции конкретным объектом. Он состоит из имени и параметров, используемых при выполнении операции.

Класс — описание множества объектов, которые разделяют общие свойства, отношения и смысл. Многие программисты рассматривают класс как шаблон для построения объекта. Объекты являются базовыми элементами при выполнении объектно-ориентированных программ, а классы — базовыми элементами при определении таких программ.

Любой объект, используемый в объектно-ориентированной программе, должен быть экземпляром некоторого класса, который предварительно необходимо спроектировать и реализовать. Различают внешнее представление класса (интерфейс) и его реализацию. Интерфейс объявляет возможности класса, но скрывает его структуру и поведение.

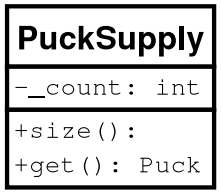

Рис. 5. Диаграмма класса PuckSupply

Интерфейс, как правило, состоит из объявлений всех операций (методов), применимых к экземплярам класса (объектам), и может быть разделён на три части:

- публичную (public), объявления которой доступны всем клиентам (в UML знак «+»);


```
    return ((int) Math.toDegrees(Math.atan2(_dy,_dx)))%360;
  }

  /* инвертировать направление движения */
  public void reverse() {
    _dx = -_dx;
    _dy = -_dy;
  }
  /* инвертировать направление по оси X*/
  public void reverseX() {
    _dx = -_dx;
  }
  /* инвертировать направление по оси Y*/
  public void reverseY() {
    _dy = -_dy;
  }
}
```

Класс `Velocity` содержит также методы `setDirection`, `reverseX`, `reverseY` и `reverse`, позволяющие повысить эффективность программы, устраняя необходимость создания новых экземпляров класса всякий раз, когда изменяется направление движения.

§ 3. Варианты заданий

В каждом из приведенных ниже вариантов курсовой работы требуется:

1) проанализировать полученное задание и найти способ модификации эталонного проекта, обеспечивающий выполнение задания;
2) изобразить все изменения, сделанные в проекте, при помощи диаграмм случаев использования и диаграммы классов;
3) реализовать новый проект, внеся необходимые изменения в интерфейсы и реализации классов.

Все задания на включение нового типа кирпичей предполагают, что из общего количества кирпичей двадцать процентов будут новыми.

Задание 1. Добавить новый вид кирпичей. Крепкий кирпич (HardBrick) исчезает с игрового поля только при двукратном попадании в него шайбой. При первом попадании он должен покрыться трещинами (подмена растрового изображения).

Задание 2. Добавить новый вид кирпичей. При попадании в такой кирпич (PowerBrick) шайба увеличивает скорость движения на два пункта.

§ 2. Проектный анализ

В этом разделе будет описано множество объектов, образующих в совокупности игру. Мы будем использовать UML (Unified Modeling Language — унифицированный язык моделирования) в качестве языка описания и Java в качестве языка программирования. Базовые понятия UML, необходимые для изложения проекта, представлены ниже. Знание основ объектно-ориентированного программирования и языка Java предполагаются.

1. **Диаграммы случаев использования.** При разработке объектно-ориентированного программного обеспечения весьма эффективным является формулирование требований к отдельным объектам с помощью *случаев использования*.

Случай использования (use case) описывает, как действующий субъект взаимодействует с системой. Действующие субъекты (актёры) исполняют различные роли, в которых пользователи выступают по отношению к системе. В нашем проекте явно присутствует только один субъект (игрок), но можно предположить наличие ещё одного действующего субъекта (администратора), который при установке игры на различные компьютеры будет изменять её параметры: размер игрового поля, количество и скорость шайб и т. д. Основные случаи использования проекта перечислены в таблице 1.

Таблица 1. Основные случаи использования

Наименование	Описание
Start (Начало)	Игрок начинает сеанс игры
Pause (Приостановка)	Игрок делает перерыв в игре
Resume (Возобновление)	Игрок возобновляет сеанс игры
Stop (Конец)	Игрок завершает сеанс игры

Случаи использования упорядочены в иерархию с помощью двух отношений: **uses** (использует) и **extends** (расширяет). Можно уточнить некоторые из случаев использования и развернуть их в наборы более специфичных случаев (рис. 4). Подобная структура позволяет упростить поиск нужного случая. Нужно отметить, что случаи использования являются не программами, а требованиями, которым должно соответствовать программное обеспечение.

В контексте конкретного случая использования можно определить один или большее число *сценариев*. Например, случай использования «Двигать

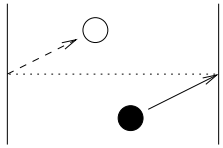

Рис. 11

Задание 13. Изменить реализацию классов-хранилищ `PuckSupply` и `BrickPile` таким образом, чтобы они использовали стандартный интерфейс `java.utils.Enumeration`.

Задание 14. Добавить возможность перемещения лопатки при помощи мыши.

Задание 15. Добавить «бонусный» кирпич (BonusBrick), при разбивании которого появляется дополнительная шайба, которая падает вертикально вниз. С этого момента в игре участвуют одновременно две шайбы.

Задание 16. Добавить «бонусный» кирпич (BonusBrick), при разбивании которого появляется «бонус» (красный круг), который затем падает вертикально вниз. Игра заканчивается проигрышем, если игрок не успеет поймать его при помощи лопатки.

Задание 17. Реализовать правила отскока шайбы от лопатки, описанные в первом разделе пособия (см. рис. 3).

Задание 18. Реализовать корректный отскок шайбы при её попадании в боковые стенки кирпича.

§ 4. Пример выполнения курсовой работы

В качестве примера рассмотрим задание первого варианта. Нам необходимо добавить новый вид кирпича, при попадании в который в первый раз кирпич изменяет свое графическое представление, а во второй раз — разбивается и удаляется с игрового поля. Затем такие кирпичи нужно добавить на игровое поле, при этом их число должно составить двадцать процентов от общего количества кирпичей.

Начнём модернизацию с добавления нового класса в иерархию спрайтов. Очевидно, что `HardBrick` должен унаследовать свои свойства от класса `Brick`. Остаётся понять, какие новые атрибуты необходимо добавить в новый класс. Их всего два: счётчик ударов шайбы (при создании равен одному) и графическое изображение повреждённого кирпича. Далее необходимо переопределить метод `hitBy`, который вызывается при соударении

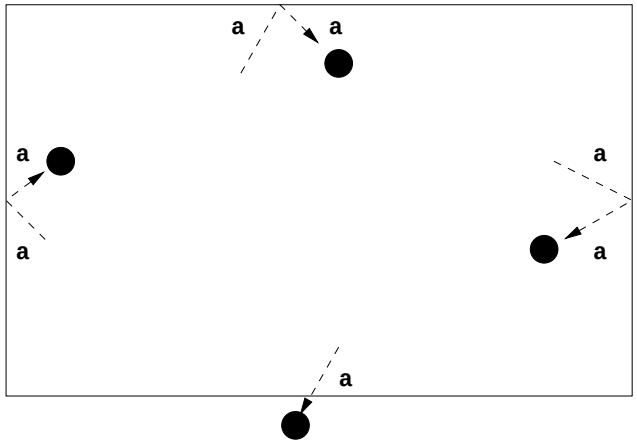

Рис. 1. Взаимодействие шайбы с границами поля

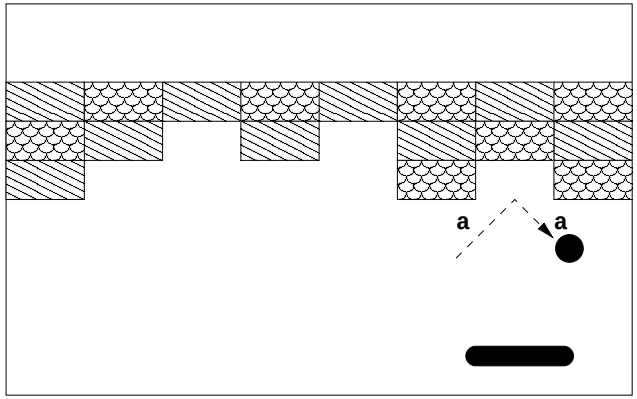

Рис. 2. Взаимодействие шайбы с кирпичами

- **Лопатка.** Игрок использует лопатку, чтобы управлять движением шайбы. Шайба отскакивает от лопатки, после чего новое направление её движения определяется двумя факторами: прежним направлением и той частью лопатки, о которую приходится удар. Разделим лопатку на три равные части и назовём *ближней третью* либо левую часть лопатки в случае, если шайба попадает в


```
                заменяет исходное изображение при первом
                ударе шайбы
   */
  private int _hitCount = 1;
  private Image _woundImg;

  public HardBrick(PlayField pf, BrickPile bp, Rectangle p,
                   Image img, Image woundImg) {
    super(pf, bp, img, p);
    _woundImg = woundImg;
  }
  /*
   * Обработка соударения с шайбой. Как только
   * значение _hitCount становится равным нулю,
   * кирпич будет удален с игрового поля
   */
  public void hitBy(Puck p) {
    if (_hitCount > 0) {
      _img = _woundImg;
      _hitCount--;
    } else {
      _isDead = true;
      if (_bp.unbrokenCount() == 0) {
        _pf.getMatch().win();
      }
    }
    p.getVelocity().reverseY();
  }
}
```

Теперь добавим новые кирпичи на игровое поле, для чего обратимся к классу `BrickPile`. Общее количество кирпичей, участвующих в игровом процессе, равно `_rows * _cols`, следовательно, необходимо добавить `(_rows * _cols)*0.2` новых. Изменим код конструктора `BrickPile`.

```
public class BrickPile {
  /*
   @_pf     - игровое поле
   @_bricks - множество кирпичей
   @_rows   - кол-во линий кирпичей
   @_cols   - кол-во кирпичей в каждой линии
  */
  private PlayField _pf;
  private Vector _bricks;
  private final int _rows = 4;
```