Министерство образования и науки Российской Федерации

**Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Тамбовский государственный технический университет»**

**Ю.Т. ЗЫРЯНОВ, И.Г. РЯЗАНОВ**

# ЛАБОРАТОРНЫЙ ПРАКТИКУМ

**по дисциплине**

# ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ЭЛЕКТРОННЫХ СРЕДСТВ

**Для студентов, обучающихся по направлению:**

211000.68 – «Конструирование и технология электронных средств»

Тамбов
Издательство ФГБОУ ВПО «ТГТУ»
2013

УДК 004.432
ББК 32.973.26-018
З-

Р е ц е н з е н т ы:

.

Профессор кафедры «Радиотехника» ФГБОУ ВПО «ТГТУ»
доктор технических наук Иванов А.В.
Ведущий специалист Воронежского филиала ОАО «Воентелеком»
кандидат технических наук, доцент Букин М.В

**Зырянов, Ю.Т.**

З- Лабораторный практикум по дисциплине информационные технологии проектирования электронных средств/ Ю.Т. Зырянов, И.Г. Рязанов – Тамбов : Изд-во ФГБОУ ВПО «ТГТУ», 2013. –39 с.

В лабораторном практикуме представлены задания, предназначенные для изучения базовых возможностей языка программирования Java .

Настоящий лабораторный практикум предназначен для студентов, обучающихся по направлению 211000.68 – «Конструирование и технология электронных средств», а также может быть использовано студентами смежных специальностей и разных форм обучения.

УДК 004.432
ББК 32.973.26-018

# ВВЕДЕНИЕ

Значение информационных технологий велико в наше время. Для успешного развития современной промышленности и экономики используются различные виды программного обеспечения в виде приложений и операционных систем. Весь программный продукт на данный момент базируется на языках программирования высокого уровня C++, Delphi, Java и др. Из перечисленных языков широкое распространение получил язык Java.

Язык Java является очень мощным языком программирования, созданным для построения защищенных, кросс-платформенных и интернациональных программ. Java постоянно расширяется, предоставляя все новые и новые возможности для решения постоянно возникающих проблем. Он построен на концепции ООП (Объектно-Ориентированного Программирования).

Содержание лабораторного практикума соответствует рабочей программе учебной дисциплины «Информационные технологии проектирования электронных средств». Практикум предназначен для студентов, обучающихся по направлению 211000.68 – «Конструирование и технология электронных средств».

В лабораторном практикуме содержатся методические указания по разработки основных программных модулей на языке программирования Java, приведены примеры, позволяющие в полной мере освоить базовые возможности языка программирования Java.

*Лабораторная работа 1*

## УСТАНОВКА И ЗАПУСК ECLIPSE

**Цель работы:** научиться инсталлировать и запускать Eclipse.

### Методические указания

Eclipse - программное обеспечение с открытым кодом, используемое в основном, как Интегрированная Среда Разработки (IDE) и как основа для программных продуктов.

***Шаг 1: Загрузить Eclipse***

Программное обеспечение может быть выгружено с сайта www.eclipse.org/downloads/index.php. С этой страницы выберите соответствующую географическую область (Северная Америка, например) и

выберите пакет, который вы хотите выгрузить. В большинстве случаев вы захотите выгрузить самый последний пакет. Щелкните на последнем пакете, а затем выберите платформу, на которой вы хотите инсталлировать продукт.

Вам будет задан вопрос, что делать с файлом? Выберите сохранение

файла на локальном диске, а затем задайте место для сохранения файла.

Теперь вы выгрузили Eclipse на вашу локальную машину.

***Шаг 2: Инсталляция Eclipse***

Когда вы выгрузили Eclipse на ваш локальный диск, вы можете приступить к инсталляции. Файл, который вы выгрузили - это ZIP-файл,

который содержит полное программное обеспечение Eclipse. Чтобы инсталлировать Eclipse, вы просто распакуйте выгруженный файл на локальный диск, например, на диск c:\. Распаковка ZIP-файла на диск c:\ приведет к созданию каталога c:\eclipse, который содержит исполняемый файл Eclipse (eclipse.exe).

***Шаг 3: Запуск Eclipse***

Щелкните дважды на файле eclipse.exe, чтобы запустить Eclipse. Поскольку Eclipse построен на Java, программное обеспечение требует для выполнения Java runtime environment (JRE) или Java Development Kit (JDK) с javaw.exe. Если у вас не установлен маршрут для javaw.exe, появится диалоговое окно с сообщением об отсутствии JRE/JDK. В таком случае, установите переменную окружения JAVA_HOME, чтобы она указывала на каталог вашей инсталляции JDK. Если на вашем локальном диске нет установки JDK или JRE, вы можете выгрузить JRE с сайта Eclipse (http://eclipse.org/downloads/index.php).

Когда вы установили на локальном диске JRE, вы можете установить

переменную окружения JAVA_HOME, если это еще не сделано.

После того, как вы будете иметь JAVA_HOME на месте и дважды щелкните на файле eclipse.exe, инсталляция будет завершена приглашением

создать каталог рабочего пространства (workspace) - каталог рабочего пространства по умолчанию является подкаталогом в каталоге инсталляции, например, *c:\eclipse\workspace* - и появится Рабочее Место (Workbench).

**Ход работы**

***1. Выгрузка Eclipse***

Для скачивания Eclipse заходим на сайт http://www.eclipse.org/downloads/


Рис. 1. Список доступных для скачивания пакетов на сайте http://www.eclipse.org/downloads/.

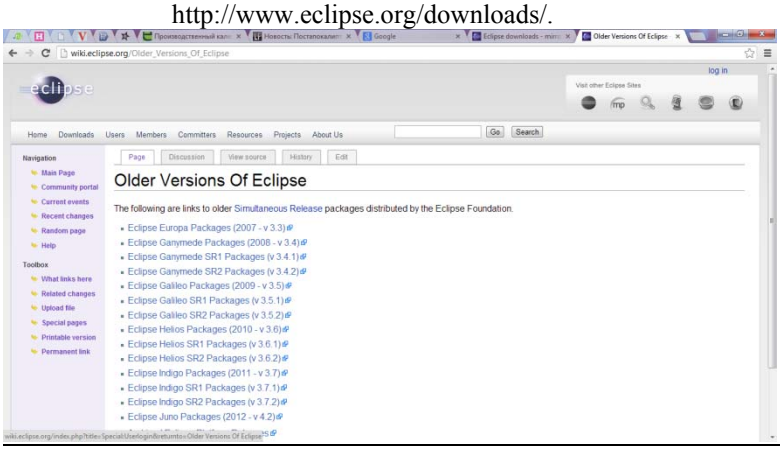



Рис. 2. Список различных версий Eclipse (http://wiki.eclipse.org/Older_Versions_Of_Eclipse).

Далее выбираем из списка Eclipse Helios Sr2.

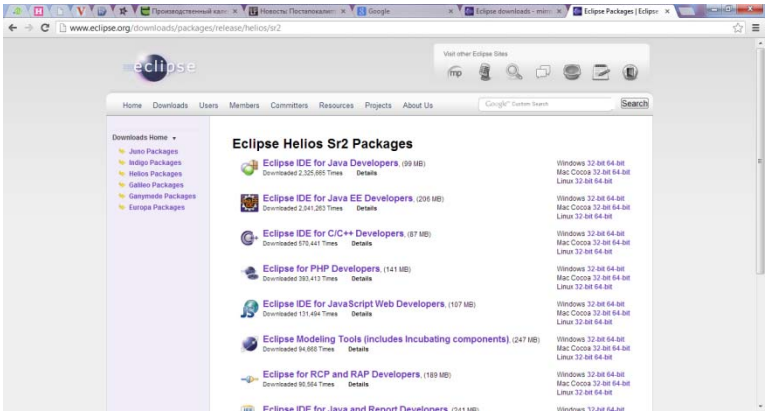



Рис. 3. Пакет для скачивания Eclipse Helios Sr2 (http://www.eclipse.org/downloads/packages/release/helios/sr2).

Выбираем для загрузки 32х-битную версию для Windows и нажимаем кнопку «Download»:

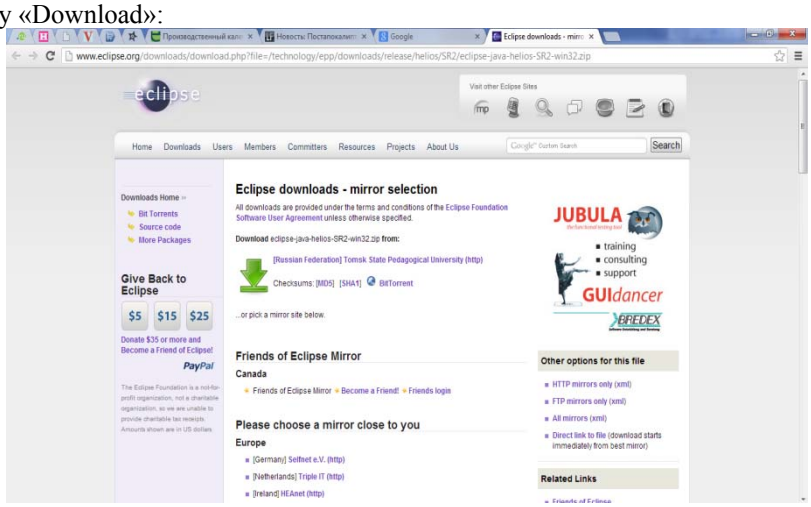



Рис. 4. Окно для выбора варианта скачивания Eclipse Helios Sr2.

Сохраняем файл на локальном диске, а затем задаем место для сохранения файла.

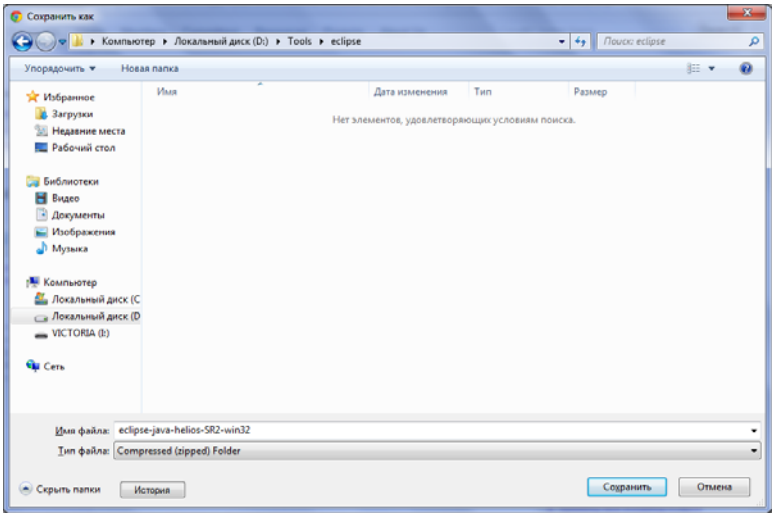



Рис. 5. Окно сохранения Eclipse Helios Sr2 на локальном диске.

2. ***Инсталляция Eclipse***

Распаковываем выгруженный файл на локальный диск, например, на диск d:\.

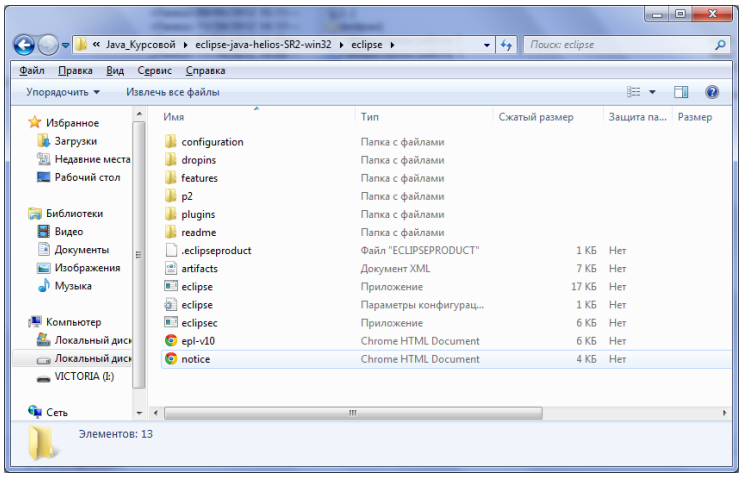



Рис. 6. Файлы архива eclipse-java-helios-SR2-win32.zip.

***3. Запуск Eclipse***

Выгрузим Java runtime environment (JRE) с сайта Eclipse (http://eclipse.org/downloads/index.php).

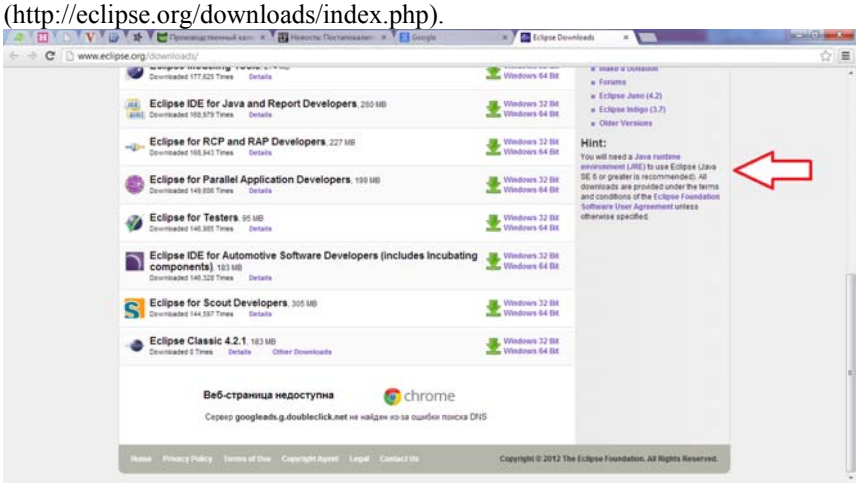



Рис. 7. Ссылка для скачивания JRE

Далее установим переменную окружения JAVA_HOME.

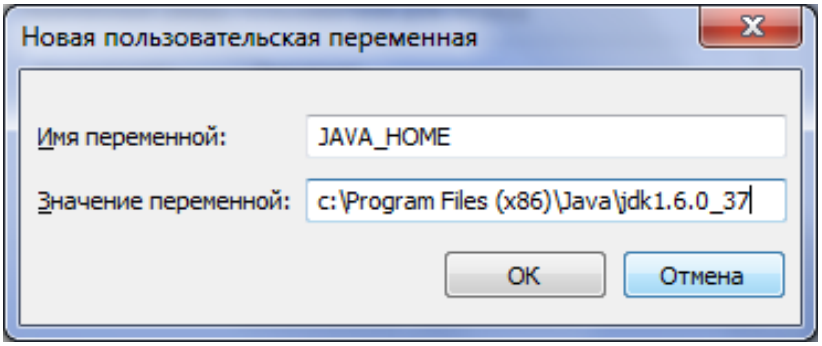



Рис. 8. Добавление новой переменной среды.

После этого дважды щелкаем на файле eclipse.exe, инсталляция будет завершена приглашением создать каталог рабочего пространства workspace).

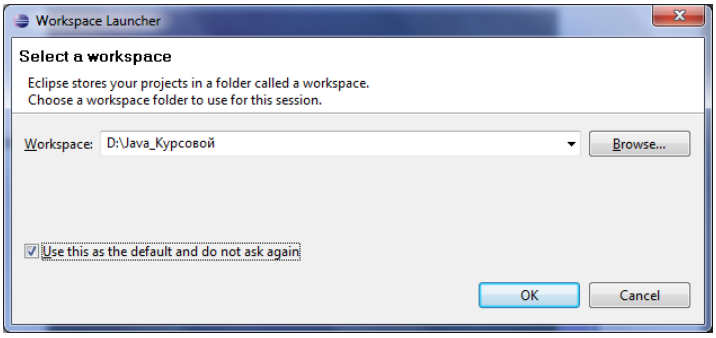



Рис. 8. Окно выбора места хранения каталога рабочего пространства. Появляется Рабочее Место (Workbench).


Рис. 9. Рабочее пространство Eclipse

**Вывод**: в данной работе мы установили и запустили Eclipse.

*Лабораторная работа №2*

## ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ КОНЦЕПЦИЯ. ИГРА ЖИЗНИ

**Цель работы:** изучить предметную область, в которой предстоит работать всю остальную часть лабораторного практикума. Приобретение навыков моделирования Игры жизни.

**Методические указания.**

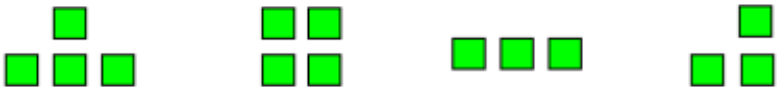
Игра жизни - одна из наиболее известных 2-мерных моделей клеточной структуры. Это не настоящая игра, так как в ней нет игроков, нет победителей и побежденных. Это игра, в которой начальная позиция и правила определяют все, что случится после этого. Больше информации об игре и ее обоснование см. по ссылкам *http://www.math.com/students/wonders/life/life.html* и *http://mathworld.wolfram.com/CellularAutomaton.html*.

Правила Игры жизни: Игра жизни происходит на сетке из квадратных клеток - как шахматная доска, - расширяясь произвольным образом по всем направлениям. Клетка может быть живой или мертвой.

Отметка на клетке показывает живую клетку. Пустая клетка является мертвой. Каждая клетка в сетке имеет восемь смежных клеток по всем направлениям, включая диагональные. Чтобы выполнить один шаг игры, подсчитайте число живых соседей для каждой клетки. Что случится дальше - зависит от этого числа:

1. Правило рождения: Мертвая клетка, имеющая ровно три живых соседа становится живой клеткой

2. Правило выживания: Живая клетка с двумя или тремя живыми соседями остается жить.



3. Правило смерти: Во всех других случаях клетка умирает или остается мертвой.

Число живых соседей всегда вычисляется до применения правила.

**Ход работы.**

***1. Определение объектов в игре***

Определим объекты, вовлеченные в игру (объекты обычно определяются поиском в требованиях имен существительных).

Какие объекты вы можете определить в игре?

Вот несколько вопросов, которые могут помочь определить объекты:

- Как представляется сама игра? - Итерационное выполнение процессов
- В чем состоит игра? – Выживание или отброс лишних клеток
- Что влияет на развитие игры во времени? - Выполнение условий выживания или смерти.

Поскольку на вопросы, приведенные выше, может быть несколько подходящих ответов, возможно, правильным будет сказать, что наиболее очевидными объектами являются игра, доска и правило, и эти объекты составляют игру.

***2. Определение основных классов***

На основании результатов предыдущего шага, какие классы составляют систему игры? Каковы имена классов, их данные и поведение?

Основными классами игры могут быть Game, Board и Rule. Игра имеет доску и ряд правил, а также индекс "поколение" для определения того, сколько поколений досок отслеживается. Доска имеет клетки, и она инициализируется при создании некоторой конфигурацией живых и мертвых клеток. Правило может иметь имя и тип и вычисляется для каждой клетки на доске. Типом правила может быть: рождение, выживание или смерть. Когда оно вычисляется, проверяется тип правила, и на базе типа выполняются различные вычисления.

**Вывод:** в этой работе мы проанализировали предметную область игры, которую будем реализовывать в остальных работах.

*Лабораторная работа №3*

## ОБЗОР JAVA

**Цель работы**: приобретение навыков использования Scrapbook для вычисления некоторого Java-кода, продолжение знакомства со средой Eclipse, а также с основным синтаксисом и правилами языка Java.

**Методические указания**

Scrapbook используется в Eclipse для инспектирования и вычисления кода. Сначала вам нужно создать проект, а в проекте - новый Scrapbook.

**1. Создание страницы Scrapbook**

В среде Eclipse выберем *File -> New -> Project: -> Java -> Java Project*, задав имя проекта *ExperimentLab*, щелкнем на *Finish*.

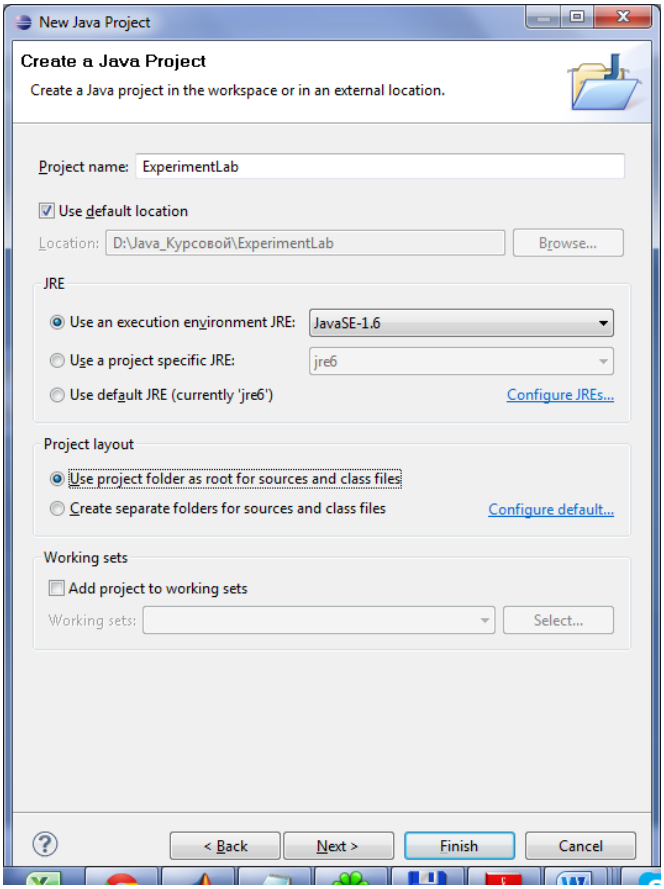



Рис. 1. Создание нового проекта

Создаем новый проект Java. Из выпадающего меню выберем File ->New -> Other: -> Java -> Scrapbook Page, затем задим имя страницы (ExperimentgPage) с выбранным проектом и щелкнем на Finish.

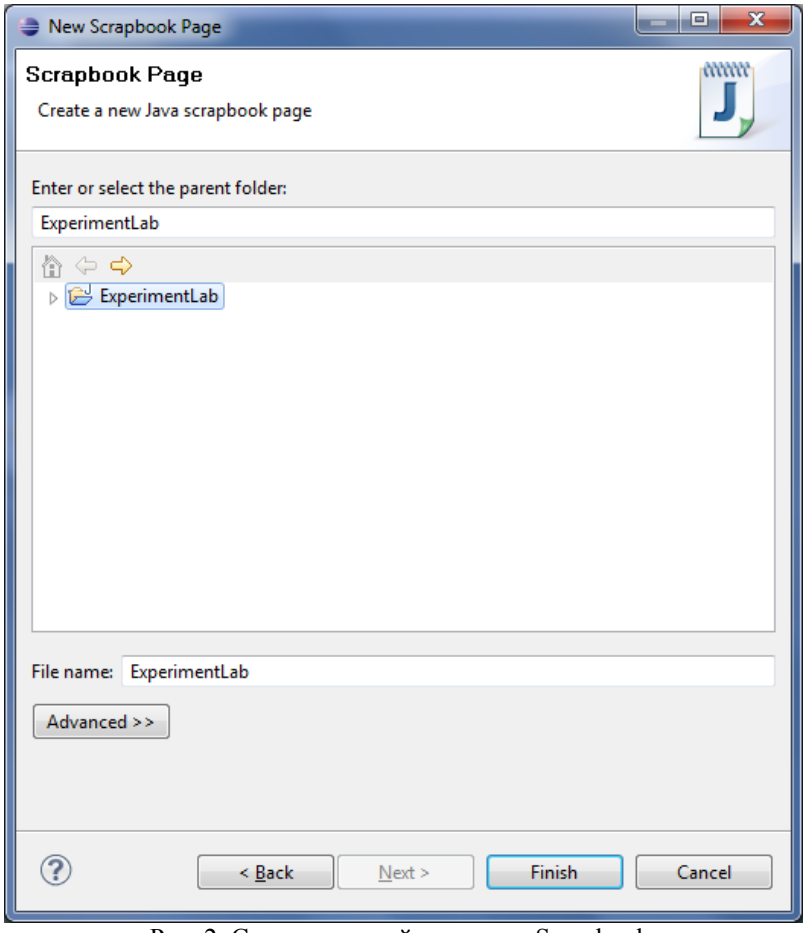



Рис. 2. Создание новой страницы Scrapbook.

Редактор страницы откроет страницу.

**2. Вычисление и отображение выражений**

Напишите и отобразите результат вычисления следующего выражения:

*5 + 2 * 10 /4 - 7;*

Какой получился результат? Тот ли он, которого вы ожидали?

**Ответ:** В вычисление приведенного выше выражения сначала вычисляется 2*10 с результатом 20 , а затем вычисляется 20/4 с результатом 5. После вычисления 5+5-7 получается финальный результат 3.

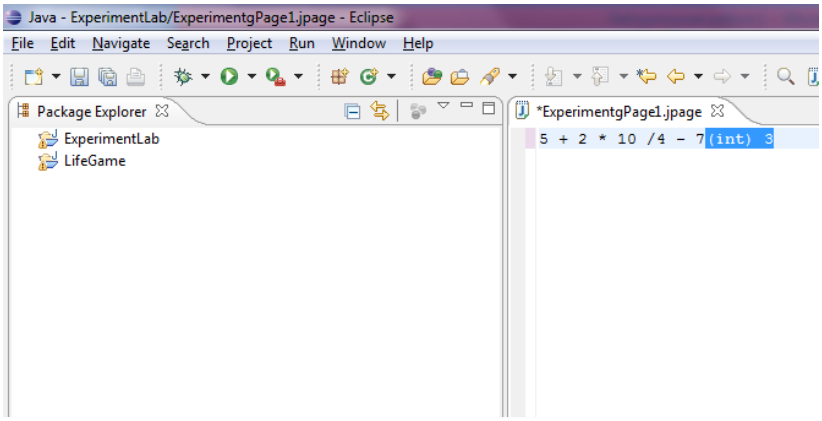



Рис. 3. Результат вычислений согласно пункту 2.

***3. Эксперименты с примитивными типами и переменными***

Напишите и отобразите результат вычисления следующих выражений:

*int a = 6;*

*double b = a;*

*b;*

Что произошло? Преобразовалось ли int в double и почему?

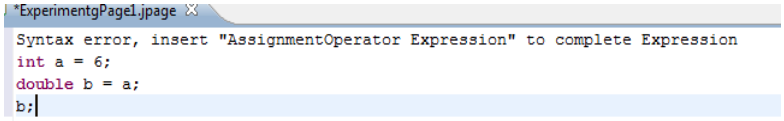



Рис. 4. Результат вычислений согласно пункту 3.

**Ответ:** Примитивный тип double - более широкий тип, чем int, и JVM в состоянии преобразовать int в double автоматически, неявным преобразованием типов.

Теперь напишите и отобразите результат вычисления следующих выражений:

*double a = 6;*

*int b = a;*

*b;*

Что произошло теперь? Почему вы получили сообщение "Type mismatch: cannot convert from double to int"? Что, если вы вычислите (int)b в конце? **Ответ:** Тип double не может быть преобразован в тип int, поскольку он

более широкий и неявное преобразование типов не выполняется. При помощи явного преобразования (int)b вы сообщаете JVM, что можно присвоить более широкий тип узкому. Напишите и отобразите результат вычисления следующих выражений:

*long a = 45L;*
*double b = a;*
*b;*
Каков теперь результат вычисления выражений?
Напишите и отобразите результат вычисления следующих выражений:
*double a = 45.00;*
*long b = a;*
*b;*
Каков теперь результат вычисления выражений? Получили ли вы тот же результат, что и при предыдущем вычислении? Почему? **Ответ:** Оба типа имеют одинаковую ширину. При вычислении первого выражения тип long присваивается double, и это обрабатывается JVM, поскольку double имеет большую точность. Однако, во втором выражении тип большей точности присваивается типу меньшей точности, и вы получите ошибку. Даже если будет применено явное преобразование, JVM не сможет обработать его.

***4. Сообщения и объекты***

Проинспектируйте следующее выражение из Scrapbook:
*String myString = new String("My String");*
*myString*
Инспектор покажет следующее.

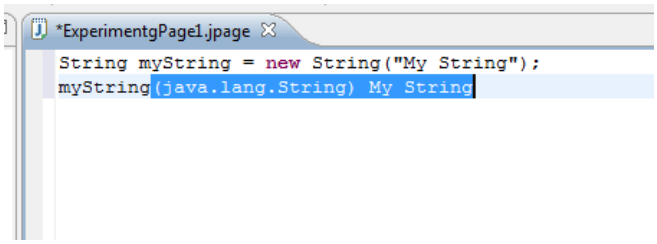



Рис. 5. Результат инспекции.

Просмотрите все символы, которые составляют проверяемую строку.
Теперь вычислите следующее выражение:
*String myString = new String("My*
*String");*
*System.out.println(myString);*
Какое сообщение посылаете вы в объект вывода? Где будет строка напечатана?

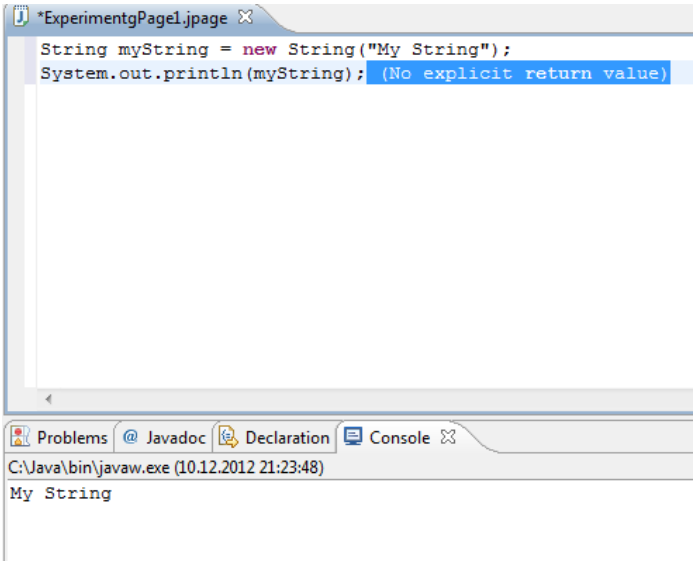



Рис. 6. Результат вычислений

**Ответ**: println является посылкой сообщения в объект out. Строка печатается на стандартной консоли Java, которую представляет в Eclipse представление Console.

***5. Литералы***

Проинспектируйте литерал "This is a string" в Scrapbook. Каким видом объекта он является? В чем различие между литералом и вычислением выражения new String("This is a string")?

Ответ: "This is a string" является литеральным строковым объектом. Он - то же, что и строковый объект, созданный при помощи выражения new String("This is a string").

***6. Массивы***

Создайте и проинспектируйте массив из 5 целых чисел, проинициализированных нулями. Создайте и проинспектируйте многомерный массив из 5 массивов, созданных на предыдущем шаге.

**Вывод**: в этой работе мы научились использовать Eclipse Scrapbook для вычисления и выполнения простого кода Java.

*Лабораторная работа №4*

## ПОСТРОЕНИЕ КЛАССОВ JAVA

**Цель работы:** знакомство с Java Development Tools в Eclipse. приобретение навыков создания классов при помощи Eclipse, а также с использованием инструментов рефакторинга Java в Eclipse.

**Ход работы:**

1. *Создание проекта*

Создадим новый проект Java с именем LifeGame. Для создания проекта выберем из выпадающего меню: File -> New -> Project -> Java -> Java Project и щелкнем на Next. Задаём имя проекта и щелкаем на Finish.

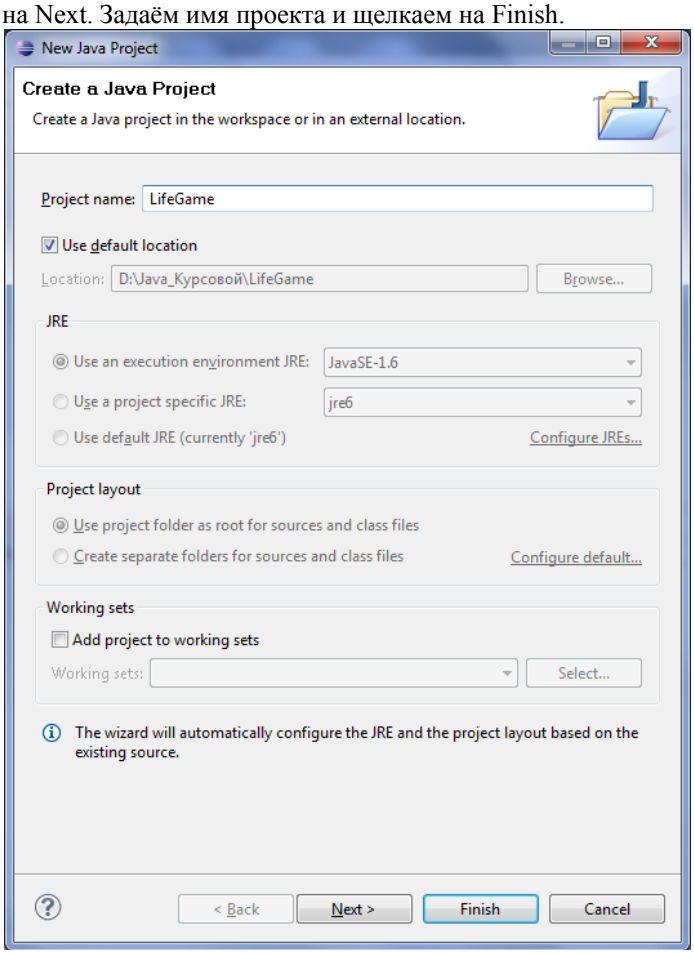



Рис.1 Создание нового проекта

2. ***Создание пакета***

Создадим пакет, который будет содержать прикладные классы для игры. Пакеты группируют классы по функциональности и по их пространствам имен. Обычно классы, которые представляют модель,

находятся в одном пакете, классы, которые используются для тестирования, находятся в другом пакете и т.д. Для создания нового пакета выберем проект и выберем New -> Package из контекстного меню. В качестве имени пакета - org.eclipse.lifegame.domain и щелкаем на Finish.

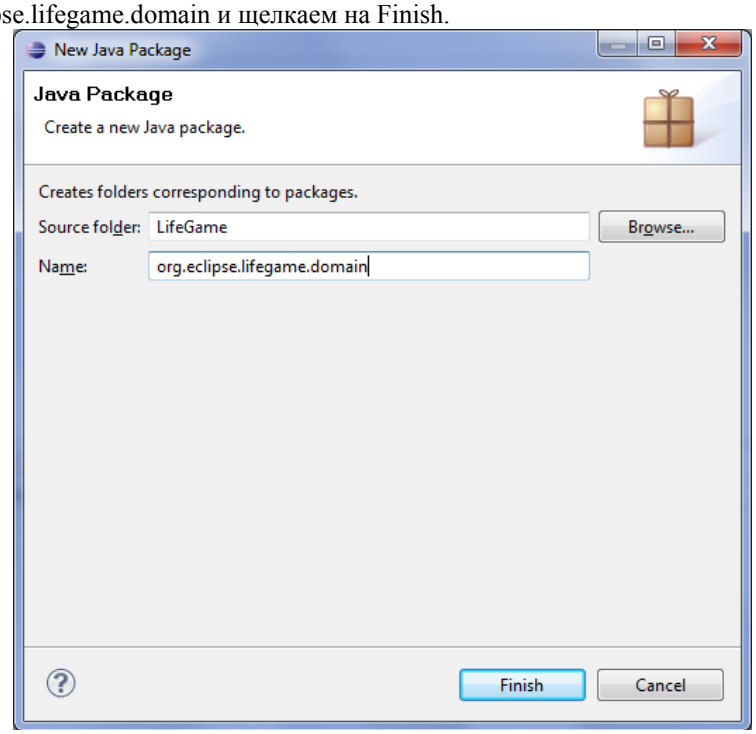



Рис. 2. Создание нового пакета.

***3. Создание классов Game и Board***

Создадим класс Board, который будет иметь закрытое поле cells, являющееся 2-мерным массивом целых чисел. Поле cells должно быть 10x10 и содержать после запуска игры нули или единицы.

Сгенерим методы-акцессоры для поля. Добавим конструктор класса Board, который будет принимать в качестве параметра 2-мерный массив целых чисел. Он будет представлять шаблон (начальное состояние клеток) для игры, когда она запустится. Также заместим конструктор по умолчанию, чтобы просто инициализировать клетки в 2-мерном (10x10) массиве при создании.

Определим в классе два метода разворачивания: evolve(Game) и evolve(Game,int). Эти методы должны иметь возвращаемый тип void и должны проходить через клетки и на основании правил развивать следующую стадию. Метод развития, который принимает целый параметр,

развивает доску через много стадий (в зависимости от параметра). Оставим эти методы пока пустыми.

Заместим метод toString() класса Board, чтобы он просто возвращал строку "Board for game of life" . В следующих работах вы добавите больше поведения в этот метод.

Создадим класс Game, который имеет поле board типа Board и поле generation типа int (показывающее, сколько поколений доска развивалась).

Сгенерим методы-акцессоры для полей. Добавим конструктор класса Game, который будет принимать в качестве параметра доску и инициализировать поле board значением параметра, а поколение - числом 1.

***4. Создание экземпляров Game и Board***

Используем Scrapbook из предыдущей работы для создания экземпляра класса Board. Классы не будут видны в Scrapbook. Сначала добавим проект LifeGame, как зависимость, к проекту, содержащему Scrapbook.

Выберем проект ExperimentLab и Properties из контекстного меню. Щелкнем на Java Build Path и выберем закладку Projects. Выберем проект LifeGame и нажмем на OK.

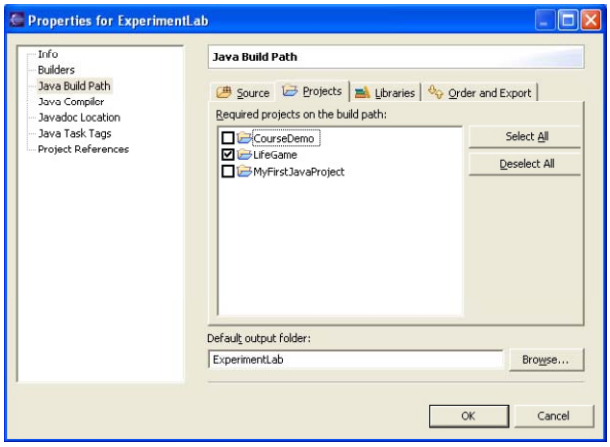



Рис. 3. Добавление проекта в Java Build Path.

В Scrapbook выберем Set Imports из контекстного меню и выберем кнопку Add Packages. Выберем пакет, который содержит классы, и щелкнем на OK дважды.

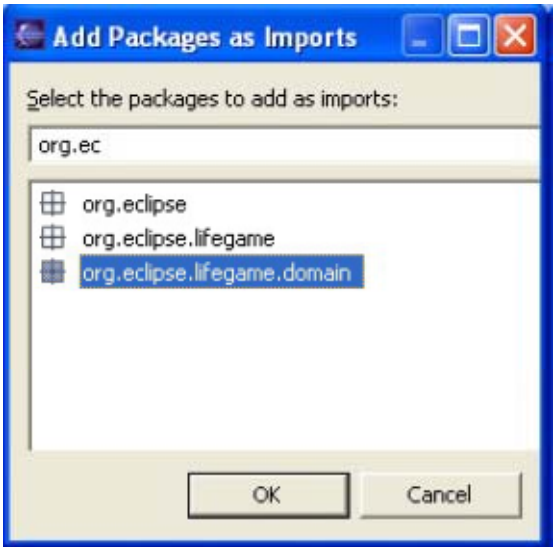


Рис. 5. Импорт пакета.

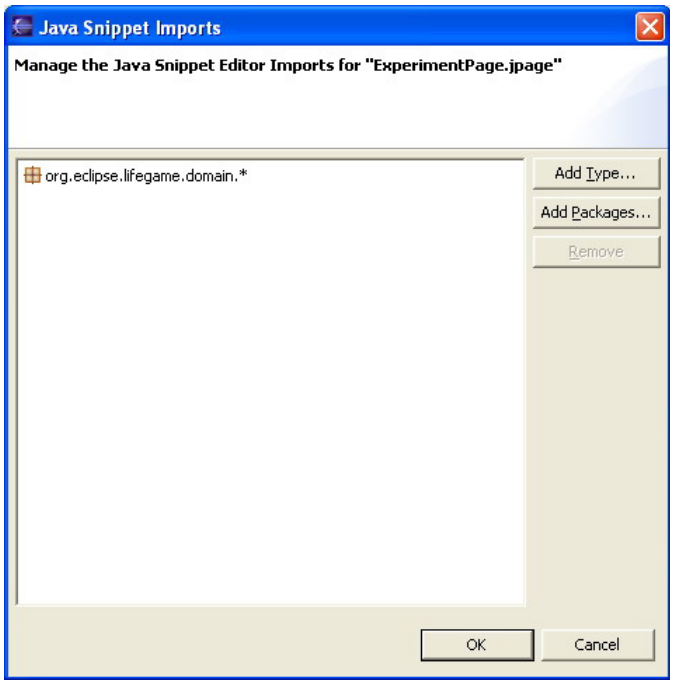


Рис. 6. Результат импорта пакета в Scrapbook.

Создадим экземпляр класса Game, передавая ему созданный объект доски. Проинспектируем объект игры.

В Scrapbook получим код:

```
int [][] boardCells = {
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1}};
Board board = new Board(boardCells);
Game game = new Game(board);
game
```

**Выводы**: в этой работе создали базовые классы для Игры жизни, и познакомились с написанием конструкторов, методов и определением полей.

*Лабораторная работа №5*

## ИСПОЛЬЗОВАНИЕ ОТЛАДЧИКА

**Цель работы:** познакомиться с Java Development Tools в Eclipse; пошагово пройти через выполнения конструктора Board и определить и исправить неправильное поведение.

**Выполнение работы**:

**Шаг 1: Создание класса Rule**

В проекте LifeGame создадим новый класс Rule (в том же пакете, что и остальные классы игры), который имеет закрытое поле name типа String.

Сгенерим методы-акцессоры для поля. Добавим конструктор класса, который принимает параметр типа String и присваивает его полю name. Конструктор выглядит так:

```
public Rule(String name){
name = name;
}
```

**Шаг 2: Создание экземпляра Rule**

Создадим в Scrapbook новый экземпляр правила, используя созданный конструктор, и проинспектируйте его.

Поле name не установлено в конструкторе по нескольким причинам. Понадобится использовать Debugger, чтобы увидеть, что происходит, когда имя передается в конструктор. В проекте LifeGame создадим новый пакет

org.eclipse.lifegame.test и добавим в пакет класс LifeTester. Выберем создание метода main(). В методе main() напишем код из Scrapbook.

Код выглядит так:

```
public static void main(String[] args) {
Rule rule = new Rule("Survival");
System.out.println(rule.getName());
}
```

Класс Rule находится в другом пакете, нужно добавить оператор импорта. Запустим класс, выбрав Run -> Run As -> JavaApplication из выпадающего меню. В Console выведется null.

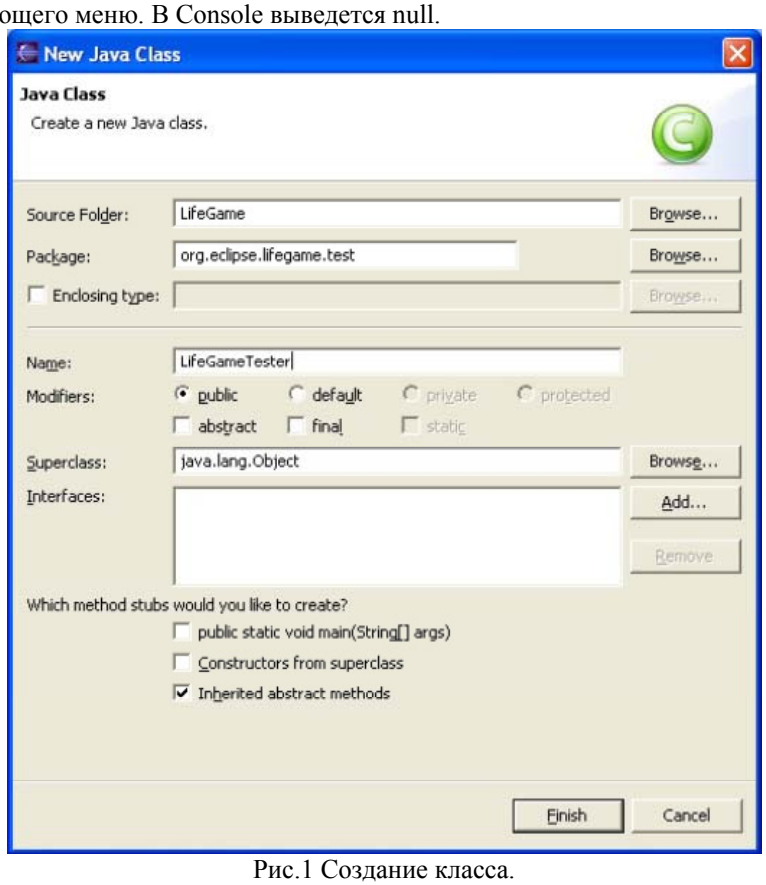



Рис.1 Создание класса.

**Шаг 3: Отладка конструктора**

Установим точку останова в методе main и снова запустим LifeGameTester. Чтобы вызвать Debugger, выберем опцию Debug вместо опции Run для класса LifeGameTester.

Debugger появляется и останавливает выполнение в методе main, где была установлена точка останова.

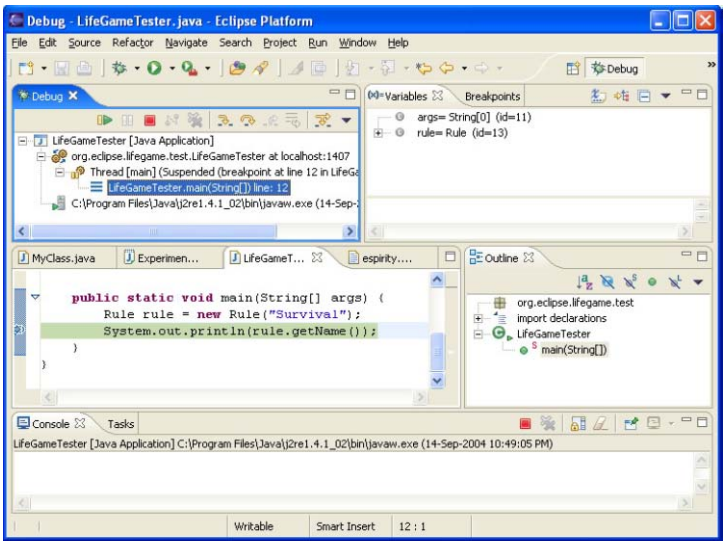



Рис.2 Перспектива «Debug»

Пройдем конструктор по шагам и вычислим переданный параметр, а также и поле name.

Конструктор использует выражение name = name; где name – это параметр, переданный в конструктор, и в то же время - поле класса. Когда код вычисляется, переданное значение имени назначается переданной переменной name. Это не дает никакого эффекта и Eclipse распознает это и отображает предупреждение следом за этим выражением. Правильное выражение - this.name = name; где this.name означает поле, а name означает переданный параметр. Перед выполнением кода с точкой останова в Scrapbook, откроется Debugger.

**Вывод:** В этой работе мы познакомились с использованием Eclipse Debugger.

*Лабораторная работа №6*

## ОПЕРАТОРЫ УПРАВЛЕНИЯ

**Цель работы:** научиться использовать операторы управления для реализации некоторого поведения классов игры. В частности, реализовывать шаблон Singleton для класса Game и метод toString() класса Board.

**Выполнение работы**

**Шаг 1: Реализация шаблона Singleton для класса Game**

Необходим только один экземпляр класса Game в одном приложении. Этот экземпляр будет представлять саму игру. Если в одной JVM может быть создано не более одного экземпляра класса, это называется шаблоном Singleton (одиночка). На этом шаге мы реализуем этот шаблон для класса Game.

Определим закрытое статическое поле с именем theInstance в классе Game. Это поле будет хранить экземпляр класса, так что тип поля должен быть Game. Добавим закрытый статический метод getInstance() в класс Game, который будет возвращать экземпляр класса, сохраненный в поле theInstance. В методе проверим, прежде всего, не имеет ли поле значение null, если да, то создадим новый экземпляр класса Game и присвоим его этому полю. Такая реализация называется ленивой инициализацией.

Изменим конструктор класса Game на закрытый, чтобы только сам класс мог его использовать. Это не даст возможности кому-либо еще создавать экземпляры класса.

Используем созданный сейчас шаблон в Scrapbook для инспектирования кода. Проинспектируем выражение Game.getInstance();

**Шаг 2: метод toString() класса Board**

В предыдущих работах мы определили метод toString() класса Board, как возвращающий простую String. Метод toString() является текстовым представлением объекта. Когда объект печатается на стандартную консоль Java, используется этот метод. Доска содержит клетки, которые являются нулями или единицами. Когда доска печатается на консоли, для клеток, содержащих нули, должны печататься пробелы, а для клеток, содержащих единицы должны печататься символы X.

Реализуем метод toString(), чтобы он возвращал строку на основании этой спецификации. Это позволит нам увидеть на Console, как выглядит наша доска. Чтобы реализовать правильное поведение, нам понадобится перебрать массив массивов и проверить каждую ячейку. Используем класс StringBuffer для добавления символов в конец строки.

Используем Scrapbook, чтобы проверить метод toString() путем создания и распечатки экземпляра класса на Console. Код Scrapbook будет выглядеть следующим образом:

```
int [][] boardCells = {
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0 , 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
```

{0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
{0, 0, 0, 0, 0, 0, 0, 0, 0, 1}};
Board board = new
Board(boardCells);
System.out.println(board);
Этот код будет выведен
на Console так:

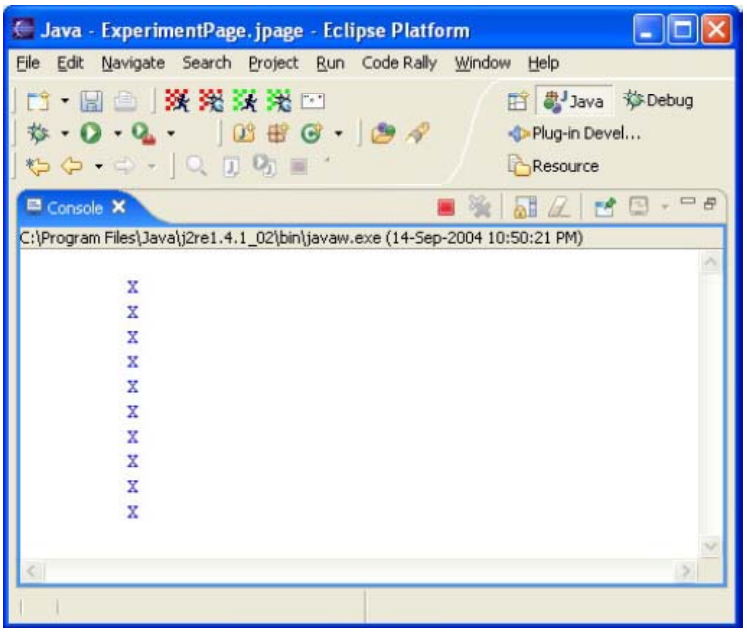



Рис. 1. Вывод в консоль метода toString() класса Board.

**Выводы**: В этой работе мы научились использовать операторы управления для реализации шаблона Singleton для класса Game и метода toString() класса Board.

*Лабораторная работа №7*

## НАСЛЕДОВАНИЕ

**Цель работы:** научиться использовать наследование для улучшения проекта, повышения гибкости и возможностей сопровождения системы.

**Выполнение работы:**

Наследование помогает при рефакторинге кода, а также при правильном представлении объектов (т.е. оно делает объект более близким к реальной

жизни). Довольно часто некоторые общие свойства похожих классов абстрагируются в абстрактном суперклассе и повторно используются в подклассах. Абстрактный класс - это класс, который не может иметь экземпляров. Интерфейсы используются в языке Java для большего абстрагирования, поскольку они определяют только объектный протокол.

Интерфейсы также используются для кросс-иерархического полиморфизма (перекрестного наследования). В этой работе мы будем использовать и абстрактные классы, и интерфейсы. Мы сосредоточимся на 2-мерном моделировании клеточной структуры в Игре жизни. Конкретно в этой работе мы будем делать систему гибкой для обработки различных версий игр моделирования клеток.

**Шаг 1: Перепроектирование класса Board**

Класс Board, который мы определили в предыдущих работах, содержит 2-мерную клеточную структуру, это означает, что он в реальности является 2-мерной доской. Всякий раз, когда мы захотите ввести новую игру (задачу), нам понадобиться менять класс. Различие между досками состоит в их клетках, поскольку 1-мерная доска является 1-мерным массивом, 2-мерная доска является 2-мерным массивом и т.д. Также различается реализация протокола evolve(), поскольку на разных досках игра развивается по-разному.

Изменим имя класс Board на TwoDimensionalBoard, поскольку это то, что на самом деле реализуется в классе Board. Чтобы изменить имя класса, выберем класс и выберем опцию Refactor ->Rename из контекстного меню. Eclipse откроет диалог для изменения имени.

Создадим абстрактный класс Board с абстрактным протоколом развития - методами evolve(Game, int) и evolve(Game). Это заставит все подклассы класса Board реализовать у себя эти методы. Изменим суперкласс класса TwoDimensionalBoard на Board (:extends Board:).

Когда мы изменяем имя класса, изменяются все ссылки, это означает, что поле board в классе Game теперь будет типа TwoDimensionalBoard. Чтобы сохранить это поле родовым, изменим его тип на Board.

Изменим его методы-акцессоры, чтобы они возвращали и принимали также тип Board.

**Шаг 2: Перепроектирование класса Rule - шаблона Strategy**

Класс Rule, который мы определили до этого, имеет поле name. Оно может быть индикатором типа правила. Правило должно иметь метод doesApply(), который возвращает значение boolean, зависящее от того, применимо ли правило для текущей клетки или нет. Протокол должен сначала проверять имя правила, а потом на его основании производить различные вычисления. При таком подходе возможны некоторые потенциальные проблемы в сопровождении при введении нового правила, поскольку метод должен будет измениться. Лучшим подходом является наличие специфического класса Rule для каждого правила в игре. Каждый

класс должен будет реализовывать собственный протокол doesApply(). Если мы сделаем так, то при введении нового правила в игру, будет создаваться новый специфический класс со своей реализацией метода doesApply(). Мы можем абстрагировать поведение правил в абстрактном классе Rule.

Изменим определение класса Rule на abstract. Создадим абстрактные классы DeathRule, SurvivalRule и BirthRule как подклассы класса Rule, создадим TwoDimensionalDeathRule, TwoDimensionalSurvivalRule и TwoDimensionalBirthRule - подклассы только что созданных абстрактных классов.

Метод doesApply() весьма специфичен для разных размерностей игры. Другими словами, метод 2-мерного правила принимает в качестве параметра 2-мерный массив клеток. Как мы можем заставить все 2-мерные правила иметь одинаковый протокол и в то же время наследовать разным суперклассам? Ответом является перекрестное наследование и интерфейсы.

Создадим новый интерфейс TwoDimensionalRule и определим методы:

**public int** getCellValue();

**public boolean** doesApply(**int**[][] cells, **int** i, **int** j);

Реализуем метод getCellValue() во всех 2-мерных классах. Метод должен просто возвращать значение, которое должно быть 1 для правил рождения и выживания и 0 для правила смерти. Изменим все классы TwoDimensionalXYZRule, чтобы реализовать созданный интерфейс. Добавим поле rules в класс Game. Поле будет массивом правил, и в нем должно быть точно 3 правила, записанные в массив. Шаблон, который мы сейчас реализовали и который показан на картинке выше, известен как шаблон Strategy.

**Шаг 3: Импорт классов Rule**

Поскольку в классах правил много кодирования, просто импортируем все классы правил (3. Эти классы реализуют методы на основе простых правил, заданных в требованиях для этой игры.

**Шаг 4: Перепроектирование класса Game**

Сделаем ранее определенный класс Game абстрактным. Eclipse выдает ошибку, поскольку не могут создаваться экземпляры этого класса, а мы создали экземпляр в реализации "одиночки". Создадим класс TwoDimensionalGame, наследующий классу Game. Конструктор по умолчанию в классе Game инициализирует правила, и он выглядит следующим образом:

```
public Game(){
this.rules = new Rule[3];
}
```

Инициализируем массив rules в конструкторе класса Game, чтобы создать экземпляры соответствующих классов 2-мерных правил,

определенных на предыдущем шаге. Конструктор выглядит следующим образом:

```
private TwoDimensionalGame(){
setBoard(new TwoDimensionalBoard());
getRules()[0] = new TwoDimensionalBirthRule();
getRules()[1] = new TwoDimensionalSurvivalRule();
getRules()[2] = new TwoDimensionalDeathRule();
}
```

**Шаг 5: Реализация протокола Board**

Реализуем методы развития в классе TwoDimensionalBoard. Метод evolve(Game, int) просто выполняет цикл до переданного ему индекса (int), в каждой итерации вызывает метод evolve(Game) и печатает себя на стандартной консоли. Код выглядит таким образом:

```
public void evolve(Game aGame, int index){
if (index == 0) return;
for (int i = 1; i < = index; i++){
evolve(aGame);
System.out.println(this);
}
}
```

Метод evolve(Game) перебирает клетки доски и для каждой клетки проверяет, применимо ли к ней какое-либо правило игры. Если правило применяется, то текущее значение клетки устанавливается в значение правила. Если никакое правило к клетке не применимо, то просто выводится на консоль сообщение о том, что к текущей клетке не применимо никакое правило к клетке Код выглядит таким образом:

```
public void evolve(Game aGame){
boolean doesAnyApply = false;
TwoDimensionalRule rule = null;
for (int i = 0; i < cells.length; i++){
for (int j = 0; j < cells[0].length; j++){
doesAnyApply = false;
for (int k = 0; k < aGame.getRules().length; k++){
rule = (TwoDimensionalRule)aGame.getRules()[k];
if (rule.doesApply(getCells(), i, j)) {
getCells()[i][j] = rule.getCellValue();
doesAnyApply = true;
break;
}
}
if (!(doesAnyApply))
System.out.println("No rules apply for cell[" + i + "][" + j + "]");
```

*}*
*}*
*}*

**Шаг 6: Реализация метода run в классе Game**

Добавим метод run() с возвращаемым значением типа void в класс Game. Метод просто вызывает метод развития игровой доски и передает ему поколение игры и саму игру в качестве параметров.

Метод выглядит так:

*public void run(){*
*getBoard().evolve(this, getGeneration());*
*}*

***Шаг 7: Тестирование игры***

Изменим класс LifeGameTester, чтобы создавать новый экземпляр TwoDimensionalBoard, присваивать некоторые инициализированные boardCells доске, а затем получать экземпляр TwoDimensionalGame, назначать ему ранее созданный board, назначать экземпляру 10 как generation и, наконец, выполнять run для экземпляра.

Метод main выглядит так:

```
public static void main(String[] args) {
int[][] boardCells = {
{0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
{0, 0, 1, 0, 1, 0, 0 , 0, 0, 0},
{0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
{0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
{0, 0, 1, 0, 1, 0, 0 , 0, 0, 0},
{0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
{0, 0, 0, 1, 0, 0, 0 , 0, 0, 0},
{0, 0, 1, 0, 1, 0, 0 , 0, 0, 0},
{0, 0, 1, 0, 1, 0, 0 , 0, 0, 0},
{0, 0, 0, 1, 0, 0, 0 , 0, 0, 0}};
TwoDimensionalBoard board = new
TwoDimensionalBoard(boardCells);
TwoDimensionalGame game = TwoDimensionalGame.getInstance();
game.setBoard(board);
game.setGeneration(10);
game.run();
}
```

Запустим класс тестера. Пронаблюдаем поведение при многократном запуске с разными значениями клеток. После рефакторинга и перепроектирования классов мы можем иметь диаграмму классов, как на картинке ниже (Рис. 1).

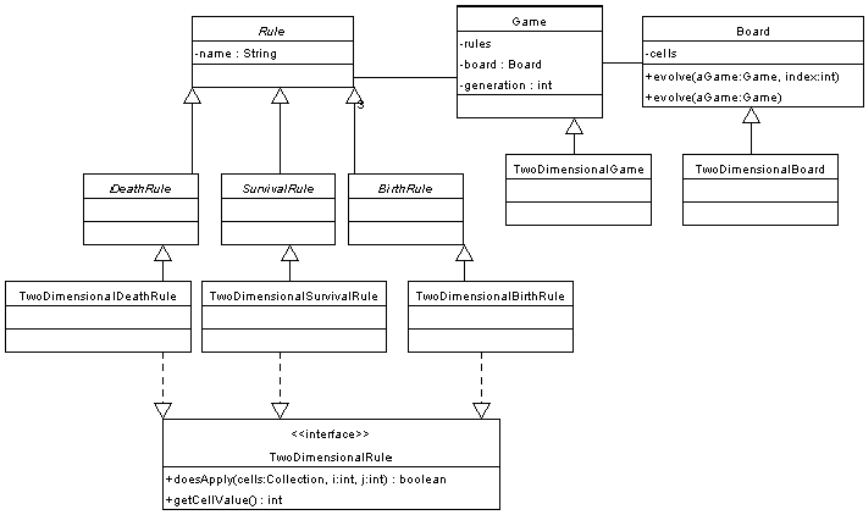



Рис. 1 Диаграмма классов

**Вывод:** мы научились использовать наследование, абстрактные классы, интерфейсы и инструменты рефакторинга Eclipse.

*Лабораторная работа №8*

## КОЛЛЕКЦИИ

**Цель работы:** научиться использовать коллекции для реализации истории развития доски в ходе игры.

**Выполнение работы:**

Когда доска развивается, ячейки меняют свои значения, и доска переходит в совсем другое состояние. В конце игры доска имеет состояние нового поколения, а все предыдущие состояния потеряны. Мы реализуем историю развития доски в класса Game.

**Шаг 1: Добавление поля boardHistory в класс Game**

Определите новое поле boardHistory типа ArrayList в классе Game. Сгенерируйте для поля методы-акцессоры. В конструторе Game инициализируйте поле новым экземпляром ArrayList.

**Шаг 2: Добавление протокола манипулирования историей**

Определим новый метод addToHistory(Board) в классе Game. Метод должен принимать в качестве параметра экземпляр класса Board и просто добавлять его к коллекции boardHistory игры. Определим другой метод в классе Game, clearHistory(), который будет удалять все элементы из коллекции boardHistory.

**Шаг 3: Использование протокола манипулирования историей**

Изменим метод run() в классе Game, чтобы он очищал историю доски перед запросом к доске на развитие. После того, как доска развилась в свое новое состояние, она добавляется к истории доски прежде, чем она разовьется опять.

Модифицируем метод evolve(Game, int) класса Board, чтобы добавлять доску к истории игры после каждого развития. Мы можем использовать выражение aGame.addToHistory(this); в цикле for.

**Шаг 4: Реализация распечатки истории**

В предыдущей работе мы добавили распечатку доски на стандартной консоли всякий раз, когда доска развивается, в метод evolve(Game,int) . Это было сделано в целях тестирования, чтобы было видно, как клетки доски изменяют значения. В действительности ответственность за распечатку доски будет у несколько иного объекта. Чтобы поддержать это, мы должны удалить оператор печати из метода evolve(Game,int) . После этого добавим в класс Game метод printHistory(), который будет перебирать коллекцию истории и в каждой итерации печатать доску на консоли. Каждый объект в коллекции приведем к его типу. BoardHistory будет содержать объекты типа Board, так что в каждой итерации по истории нам понадобится приводить объекты к этому типу.

Метод выглядит так:

```
public void printHistory(){
Iterator iterator = getBoardHistory().iterator();
Board board = null;
while (iterator.hasNext()){
board = (Board)iterator.next();
System.out.println(board);
}
}
```

**Шаг 5: Добавления к "одиночке"**

Когда мы используем шаблон "одиночки", мы используем единственный экземпляр. В динамической среде разработки, когда мы регулярно изменяем состояние и поведение объекта, это может привести к нежелательному поведению. Нам нужно очищать "одиночку" всякий раз, когда мы делаем изменения, чтобы быть уверенными, что он веден себя правильно.

Добавим открытый статический метод в класс TwoDimensionalGame, который будет просто сбрасывать экземпляр в null:

```
public static void clearInstance(){
theInstance = null;
}
```

**Шаг 6: Тестирование кода**

Изменим класс тестера для использования реализованного протокола печати истории после выполнения игры. Запустим тестер снова.

**Вывод**: мы научились использовать коллекции Java для отслеживания и сохранения истории поведения доски в игре.

*Лабораторная работа №9*

## КОЛЛЕКЦИИ

**Цель работы:** научиться использовать коллекции для реализации истории развития доски в ходе игры.

**Выполнение работы:**

Когда доска развивается, ячейки меняют свои значения, и доска переходит в совсем другое состояние. В конце игры доска имеет состояние нового поколения, а все предыдущие состояния потеряны. Мы реализуем историю развития доски в класса Game.

**Шаг 1: Добавление поля boardHistory в класс Game**

Определите новое поле boardHistory типа ArrayList в классе Game.

Сгенерируйте для поля методы-акцессоры. В конструторе Game инициализируйте поле новым экземпляром ArrayList.

**Шаг 2: Добавление протокола манипулирования историей**

Определим новый метод addToHistory(Board) в классе Game. Метод должен принимать в качестве параметра экземпляр класса Board и просто добавлять его к коллекции boardHistory игры. Определим другой метод в классе Game, clearHistory(), который будет удалять все элементы из коллекции boardHistory.

**Шаг 3: Использование протокола манипулирования историей**

Изменим метод run() в классе Game, чтобы он очищал историю доски перед запросом к доске на развитие. После того, как доска развилась в свое новое состояние, она добавляется к истории доски прежде, чем она разовьется опять.

Модифицируем метод evolve(Game, int) класса Board, чтобы добавлять доску к истории игры после каждого развития. Мы можем использовать выражение aGame.addToHistory(this); в цикле for.

**Шаг 4: Реализация распечатки истории**

В предыдущей работе мы добавили распечатку доски на стандартной консоли всякий раз, когда доска развивается, в метод evolve(Game,int) . Это было сделано в целях тестирования, чтобы было видно, как клетки доски изменяют значения. В действительности ответственность за распечатку доски будет у несколько иного объекта. Чтобы поддержать это, мы должны удалить оператор печати из метода evolve(Game,int) . После этого добавим в класс Game метод printHistory(), который будет перебирать коллекцию истории и в каждой итерации печатать доску на консоли. Каждый объект в коллекции приведем к его типу. BoardHistory будет содержать объекты типа Board, так что в каждой итерации по истории нам понадобится приводить объекты к этому типу.

Метод выглядит так:

```
public void printHistory(){
Iterator iterator = getBoardHistory().iterator();
Board board = null;
while (iterator.hasNext()){
board = (Board)iterator.next();
System.out.println(board);
}
}
```

**Шаг 5: Добавления к "одиночке"**

Когда мы используем шаблон "одиночки", мы используем единственный экземпляр. В динамической среде разработки, когда мы регулярно изменяем состояние и поведение объекта, это может привести к нежелательному поведению. Нам нужно очищать "одиночку" всякий раз, когда мы делаем изменения, чтобы быть уверенными, что он веден себя правильно.

Добавим открытый статический метод в класс TwoDimensionalGame, который будет просто сбрасывать экземпляр в null:

```
public static void clearInstance(){
theInstance = null;
}
```

**Шаг 6: Тестирование кода**

Изменим класс тестера для использования реализованного протокола печати истории после выполнения игры. Запустим тестер снова.

**Вывод**: мы научились использовать коллекции Java для отслеживания и сохранения истории поведения доски в игре.

*Лабораторная работа №10*

## ПОТОКИ

**Цель работы:** работа с потоками, приобретение навыков в написании данных в файл и считывании из файла, используя потоки.

**Выполнение работы:**

**Шаг 1: Создание методов Game**

Добавим два новых метода в класс Game с именами:

```
public void saveCurrentBoardToFile(String filename);
public void loadBoardFromFile(String filename);
```

Эти методы должны записывать игровую доску в заданный файл, а также читать доску из заданного файла. Другими словами, метод записи должен сериализовать игровую доску в файл, а метод чтения десериализовать объект доски из файла и устанавливать в него игровую доску.

***Шаг 2: Создание абстрактных методов Board***

Добавим два абстрактных метода в класс Board с именами:

*public abstract void saveCurrentBoardToFile(String filename);*

*public abstract void loadBoardFromFile(String filename);*

***Шаг 3: Проверка ваших методов***

Проверим реализацию путем вычисления следующего кода в Scrapbook:

```
int[][] boardCells = {
{0, 0, 0, 1, 0, 0, 1 , 0, 1, 0},
{0, 0, 1, 0, 1, 0, 1 , 0, 0, 0},
{1, 0, 0, 1, 0, 0, 1 , 0, 0, 0},
{0, 0, 0, 1, 0, 0, 1 , 0, 0, 0},
{0, 0, 1, 0, 1, 0, 1 , 1 ,0, 0},
{0, 0, 0, 1, 0, 0, 1 , 1, 1, 0},
{0, 0, 0, 1, 0, 0, 1 , 1, 0, 1},
{1, 0, 1, 0, 1, 0, 1 , 0, 1, 0},
{0, 0, 1, 0, 1, 0, 1 , 0, 1, 1},
{0, 0, 0, 1, 0, 0, 1 , 0, 0, 1}};
TwoDimensionalBoard board = new TwoDimensionalBoard(boardCells);
TwoDimensionalGame.clearInstance();
TwoDimensionalGame game = TwoDimensionalGame.getInstance();
game.setBoard(board);
game.saveCurrentBoardToFile("c:/game.txt");
int[][] newBoardCells = {
{1, 1},
{1, 1}};
board = new TwoDimensionalBoard(newBoardCells);
game.setBoard(board);
System.out.println(game.getBoard());
game.loadBoardFromFile("c:/game.txt");
System.out.println(game.getBoard());
```

В этом коде мы сначала создаем доску, а затем сохраняем ее в файле. Затем мы создаем новую, меньшую доску, присваиваем ее игре, чем, следовательно, удаляем старую игровую доску. После этого мы распечатываем новую доску, загружаем старую доску, присваиваем ее игре и распечатываете ее, так что вы визуально проверите, работают ли загрузка и запись. Всегда можем проверить, что было записано в файл, просмотрев его "вручную" при помощи текстового процессора.

При сериализации объекта в файл записываются двоичные данные, но если мы записываем другие данные, они должны быть читабельными.

**Выводы:** в данной работе мы научились использовать потоки Java для сохранения игровой доски в файле и загрузки ее из файла.

## ПОСТРОЕНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ ПРИ ПОМОЩИ SWT

**Цель работы:** знакомство с Standard Widget Toolket (SWT) в Eclipse; получение навыков в использовании одного из наиболее популярных менеджеров раскладки: FormLayout, многих элементов (widgets) SWT и построении графических пользовательских интерфейсы, используя элементы SWT.

**Выполнение работы:**

**Шаг 1: Запуск приложения SWT**

Для запуска независимого приложения SWT установим аргументы VM в закладке Run Wizzard/argument в следующие (обратим внимание на прямые слеши - /):

*-Djava.library.path="C:/Program Files/eclipse/plugins/org.eclipse.swt.win32_3.0.0/os/win32/x86"*

*где C:/Program Files/eclipse заменяется вашим каталогом инсталляции Eclipse.*

**Шаг 2: Разработка списка To Do**

Разработаем представление с именем SecondToDoList. Добавим внешний файл swt.jar в маршрутвашего проекта. Этот файл находится в каталоге подключения SWT каталога инсталляции Eclipse.

Будем использовать ToDoListView - находится в Lab13, - который содержит код из слайда, как стартовую точку для нового представления.

**Шаг 3: Модель светофора**

Разработаем модель светофора, которая является визуальным представлением светофора. Модель светофора обеспечивается, как класс TraficLight- находится вLab 13:

```
public class TrafficLight {
int currentState;
// Конструктор, который создает красный светофор
public TrafficLight() {
currentState = 1;
}
// Перевод светофора в следующее состояние
public int advance() {
setState(getState() % 3 + 1);
return currentState;
}
// Возврат состояния светофора (как числа от 1 до 3)
public int getState() {
return currentState;
```

```
}
// Установка состояния светофора (как числа от 1 до 3)
// Если число выходит за пределы, не делается ничего
public void setState(int newState) {
if ((newState > 0) && (newState <4))
currentState = newState;
}
// Возврат строкового представления светофора
public String toString() {
switch (getState()) {
case 1: return "Red Traffic Light";
case 2: return "Yellow Traffic Light";
case 3: return "Green Traffic Light";
};
return "Broken Traffic Light";
}
public boolean isRed() {
return getState() == 1;
}
public boolean isYellow() {
return getState() == 2;
}
public boolean isGreen() {
return getState() == 3;
}
}
```

Представление будет содержать следующие элементы:

1. Текст (Text)
2. Флажок (CheckButton)
3. Выпадающий список (Combo)
4. Три радиокнопки (Radio Button)
5. Кнопку (Push Button) Advance
6. Список (List)

Можно разместить элементы любым образом, только чтобы они не перекрывали друг друга. Общее размещение будет выглядеть так, как показано на картинке ниже (Рис.1).

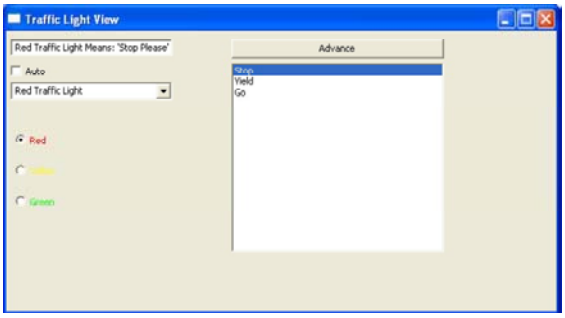


Рис.1 Общее размещение

Каждый элемент может быть использован для установки модели, за исключением элемента Text (показан в верхнем левом углу). Например, нажатие на кнопку Advance изменяет состояние семафора. Мы можем выбрать цвет в List или использовать Radio Buttons, или использовать Combo. Установка флажка Auto заставляет цвет циклически меняться каждую секунду. Важной частью решения является то, что независимо от того, какой элемент устанавливает состояние семафора, все другие элементы должны быть изменены для отражения нового состояния.

Когда мы создаем разные типы кнопок, мы задаем разные параметры для конструктора кнопки:

*pushButton = new Button(shell, SWT.PUSH); //создает обычную кнопку*

*radioButton = new Button(shell, SWT.RADIO); //создает радиокнопку*

*checkBox = new Button(shell, SWT.CHECK); //создает флажок*

Для создания Timer, который посылает сообщение каждые n миллисекунд, мы используем следующее: new javax.swing.Timer (n, new TimerEventHandler()); Сообщение actionPerformed посылается в TimerEventHandler. Это класс, который мы создаем и используем для выполнения действий через определенные периоды:

```
private class TimerEventHandler implements ActionListener {
public void actionPerformed(ActionEvent e) {
// код обработчика
}
}
```

Большинство элементов имеет свойства цвета. Можно посмотреть их в конкретных классах, но обычно делают следующее:

```
aButton.setForeground (new Color(display, 255,0,0));
```

**Вывод:** получили навыки в использовании одного из наиболее популярных менеджеров раскладки: FormLayout, многих элементов (widgets) SWT и построении графических пользовательских интерфейсы, используя элементы SWT.

**СОДЕРЖАНИЕ**

## СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ


1. Головицына, М.В. Информационные технологии проектирования радиоэлектронных средств [Электронный ресурс]: учеб. / М.В. Головицына. М.: БИНОМ. Лаборатория знаний, 2008. – Режим доступа: http://window.edu.ru/window/catalog?p_rid=64587

2. Информационные технологии проектирования радиоэлектронных средств: Учебное пособие с грифом УМО / Ю.Л. Муромцев [и др.]. – М.: Издательский центр «Академия», 2010. – 384 с.

3. Балыбин, В.М. Информационные технологии проектирования радиоэлектронных средств: курсовое проектирование: учебно-метод. пособие / В.М. Балыбин, Ю.Л. Муромцев, Л.П. Орлова. – Тамбов: ТГТУ, 2004. – 84 с.

4. Муромцев, Ю.Л. Информационные технологии проектирования РЭС [Электронный ресурс]: Методические указания / Ю.Л. Муромцев, А.Н. Грибков. – Электрон. дан. (612 Мб). – Тамбов. Издательство ТГТУ, 2008. – 1 электрон. опт. диск (CD-ROM); 12 см. – Загл. с этикетки диска.

5. Сухов, С.А. Основы программирования на JAVA: Учебное пособие [Электронный ресурс]: учеб. / С.А. Сухов. - Ульяновск: УлГТУ, 2006. - 88 с. – Режим доступа: http://window.edu.ru/resource/305/26305

6. Матросов, А.В. Программирование на Java. [Электронный ресурс]: учеб. / А.В. Матросов - СПб.: СПбГИТМО(ТУ), каф. КТ, 2003.– Режим доступа: http://window.edu.ru/resource/775/23775

7.Бородин, С. М. Обеспечение надежности при проектировании РЭС [Электронный ресурс]: учебное пособие / С. М. Бородин. – Ульяновск: УлГТУ, 2010. – Режим доступа: http://window.edu.ru/window/catalog?p_rid=74509

8.Балыбин, В.М. Информационные технологии проектирования радиоэлектронных средств: курсовое проектирование: учебно-метод. пособие / В.М. Балыбин, Ю.Л. Муромцев, Л.П. Орлова. – Тамбов: ТГТУ, 2004. – 84 с.

9.Кольтюков, Н.А. Основы компьютерного проектирования и моделирования [Электронный ресурс]: Лабораторные работы / Н.А. Кольтюков. – Электрон. дан. (520 Мб). – Тамбов. Издательство ТГТУ, 2010. – 1 электрон. опт. диск (CD–ROM); 12 см. – Загл. с этикетки диска.

10.Чернышова, Т.И. Моделирование электронных схем [Электронный ресурс]: Учебное пособие / Т.И. Чернышова, Н.Г. Чернышов. – Электрон. дан. (410 Мб). – Тамбов. Издательство ТГТУ, 2010. – 1 электрон. опт. диск (CD–ROM); 12 см. – Загл. с этикетки диска.

11.Королев, А.П. Автоматизация технологического проектирования РЭС: Учебное пособие / А.П. Королев, С.Н. Баршутин. – Тамбов. Издательство ТГТУ, 2006. – 76 с.