Министерство транспорта Российской Федерации

Федеральное агентство железнодорожного транспорта

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Дальневосточный государственный университет путей сообщения»

Естественно-научный институт

Кафедра «Вычислительная техника и компьютерная графика»

Курсовая работа

«Применение ООП для разработки программного обеспечения змейка»

КР.09.03.03.ЯиМП.10.01-БО921ПРИ

Исполнитель

студент, БО921ПРИ гр.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.В. Наконечный

Руководитель

старший преподаватель\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_П.С. Тимош

Хабаровск 2021

Оглавление

[Введение 2](#_Toc73556879)

[1 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 3](#_Toc73556880)

[1.1 Принципы объектно-ориентированного программирования 3](#_Toc73556881)

[1.2 Процесс разработки ПО 7](#_Toc73556882)

[1.3 Жизненный цикл ПО 9](#_Toc73556883)

[2 ПРАКТИЧЕСКАЯ ЧАСТЬ 12](#_Toc73556884)

[2.1 Выделение объектов участвующих в задаче 12](#_Toc73556885)

[2.2 Описание выделенных объектов в виде иерархии классов в нотации UML 12](#_Toc73556886)

[2.3 Описание взаимодействия объектов в нотации UML 14](#_Toc73556887)

[2.4 Реализация получившихся классов 15](#_Toc73556888)

[2.4 Программа, показывающая взаимодействие описанных объектов. 20](#_Toc73556889)

[Заключение 22](#_Toc73556890)

[Список использованных источников 22](#_Toc73556891)

# Введение

Зачастую, начинающие разработчики начинают писать программу, непосредственно после получения задания, толком не разобравшись в необходимых классах и в целом в структуре программы. Чтобы избежать многократные изменения зависимостей классов, самих классов, их реализации, необходимо уметь предварительно проектировать структурную часть приложения с помощью различных диаграмм. Это не только позволит определить зависимости классов, функций, но и будет хорошей презентацией проекта, так как программист должен понимать, что заказчик зачастую не владеет навыками программирования, а поэтому разработчик должен качественно и доходчиво предоставлять информацию о приложении и его структуру, функционал и другие необходимые данные, поэтому умение проектировать диаграммы должно быть в вооружении программиста.

Объектно-ориентированное программирование (ООП) – методология программирования, основанная представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования. ООП – подход к программированию как к моделированию информационных объектов, решающий основную задачу структурного программирования – структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования, что очень важно при реализации крупных проектов. Все это предполагает минимизацию избыточности данных, их целостность и удобство в понимании как самим разработчиком, так и его последователям. Также в ООП присутствуют основные принципы структурирования, которые требуются для оптимального управления соответствующий системой: абстракция, инкапсуляция, наследования, полиморфизм [1]. Не менее полезным и интересным является ортодоксальная-каноническая форма класса, которая приводит ваш собственный тип к конкретным типам данных как int, char, double и другие.

# 1 теоретические сведения

## 1.1 Принципы объектно-ориентированного программирования

Абстракция в объектно-ориентированном программировании – это использование только тех характеристик объекта, которые с достаточной точностью представляют его в данной системе. Основная идея состоит в том, чтобы представить объект минимальным набором полей и методов и при этом с достаточной точностью для решаемой задачи.

Это важный инструмент ООП наряду с полиморфизмом, наследованием и инкапсуляцией.

Абстракция является основой объектно-ориентированного программирования и позволяет работать с объектами, не вдаваясь в особенности их реализации.

Абстракция данных – одно из наиболее старых понятий объектно-ориентированного программирования, возникшее ещё до его появления. Абстракция данных связывает лежащий в основе тип данных с набором операций над ним. Пользователь типа данных не имеет прямого доступа к его реализации, но может работать с данными через предоставленный набор операций. Преимущество абстракции данных в разделении операций над данными и внутреннего представления этих данных, что позволяет изменять реализацию, не затрагивая пользователей типа данных [2].

Инкапсуляция – в информатике размещение в одном компоненте данных и методов, которые с ними работают. Также может означать скрытие внутренней реализации от других компонентов. Например, доступ к скрытой переменной может предоставляться не напрямую, а с помощью методов для чтения и изменения её значения.

В ООП инкапсуляция тесно связана с принципом абстракции данных. В общем случае в разных языках программирования термин «инкапсуляция» относится к одной или обеим одновременно следующим нотациям:

– механизм языка, позволяющий ограничить доступ одних компонентов программы к другим;

– языковая конструкция, позволяющая связать данные с методами, предназначенными для обработки этих данных.

Слово «инкапсуляция» происходит от латинского in capsula – «размещение в оболочке». Таким образом, инкапсуляцию можно интуитивно понимать как изоляцию, закрытие чего-либо инородного с целью исключения влияния на окружающее, обеспечение доступности главного, выделение основного содержания путём помещения всего мешающего, второстепенного в некую условную капсулу (рисунок 1).



Рисунок 1 – Инкапсуляция

Наследование – концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения (рисунок 2).



Рисунок 2 - Наследование

Подкласс, производный класс, дочерний класс, класс потомок, класс наследник или класс-реализатор – класс, определённый через наследование от другого класса или нескольких таких классов [3].

Базовый класс – это класс, находящийся на вершине иерархии наследования классов и в основании дерева подклассов, то есть не являющийся подклассом и не имеющий наследований от других классов и интерфейсов. Базовым классом может быть абстрактный класс и интерфейс. Любой не базовый класс является подклассом.

Интерфейс – это структура, определяющая чистый интерфейс класса, состоящий из абстрактных методов. Интерфейсы участвуют в иерархии наследований классов и интерфейсов.

Типы наследования:

– «простое» наследование. «простое» наследование, иногда называемое одиночным наследованием, описывает родство между двумя классами: один из которых наследует второму. Из одного класса могут выводиться многие классы, но даже в этом случае подобный вид взаимосвязи остается «простым» наследованием;

– множественное наследование. При множественном наследовании, у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости.

Большинство современных объектно-ориентированных языков программирования (C#, Java, Delphi и другие) поддерживают возможность одновременно наследоваться от класса-предка и реализовать методы нескольких интерфейсов одним и тем же классом. Этот механизм позволяет во многом заменить множественное наследование – методы интерфейсов необходимо переопределять явно, что исключает ошибки при наследовании функциональности одинаковых методов различных классов-предков.

Полиморфизм в языках программирования и теории типов – способность функции обрабатывать данные разных типов.

Существует несколько разновидностей полиморфизма. Две принципиально различных из них были описаны Кристофером Стрэчи в 1967 году: это параметрический полиморфизм и ad-hoc-полиморфизм, причём первая является истинной формой, а вторая – мнимой, прочие формы являются их подвидами или сочетаниями. Параметрический полиморфизм подразумевает исполнение одного и того же кода для всех допустимых типов аргументов, тогда как ad-hoc-полиморфизм подразумевает исполнение потенциально разного кода для каждого типа или подтипа аргумента.

Ad-hoc-полиморфизм поддерживается во многих языках посредством перегрузки функций и методов, а в слабо типизированных – посредством приведения типов.

Параметрический полиморфизм позволяет определять функцию или тип данных обобщённо, так что значения обрабатываются идентично вне зависимости от их типа. Параметрическая функция использует аргументы на основе поведения, а не значения, апеллируя лишь к необходимым ей свойствам аргументов, что делает её применимой в любом контексте, где тип объекта удовлетворяет заданным требованиям поведения.

Ортодоксальная каноническая форма принадлежит к числу важнейших идиом. Если следовать этому принципу при определении классов, то переменная, созданная на базе такого класса, будет объявляться, присваиваться и передаваться в аргументах точно так же, как любая стандартная переменная.

## 1.2 Процесс разработки ПО

Процесс разработки программного обеспечения – процесс, посредством которого потребности пользователей преобразуются в программный продукт. Процесс разработки программного обеспечения является составной частью программной инженерии.

Существует несколько моделей такого процесса, каждая из которых описывает свой подход, в виде задач и/или деятельности, которые имеют место в ходе процесса: водопадная, итерационная, интерактивная, спиральная.

Водопадная модель жизненного цикла была описана Уинстоном Ройсом в статье "Managing the Development of Large Software Systems" в 1970 г. Она предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Требования, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Этапы проекта в соответствии с каскадной моделью:

1) формирование требований;

2) проектирование;

3) реализация;

4) тестирование;

5) внедрение;

6) эксплуатация и сопровождение.

Преимущества:

– полная и согласованная документация на каждом этапе;

– легко определить сроки и затраты на проект.

Недостатком водопадной модели является переход от одной фазы проекта к другой предполагает полную корректность результата предыдущей фазы.

Итерационная модель. Альтернативой последовательной модели является так называемая модель итеративной и инкрементальной, получившей также от Т. Гилба в 70-е гг. название эволюционной модели. Также эту модель называют итеративной моделью и инкрементальной моделью.

Модель интерактивной и инкрементальной разработки предполагает разбиение жизненного цикла проекта на последовательность итераций, каждая из которых напоминает «мини-проект», включая все процессы разработки в применении к созданию меньших фрагментов функциональности, по сравнению с проектом в целом. Цель каждой итерации – получение работающей версии программной системы, включающей функциональность, определённую интегрированным содержанием всех предыдущих и текущей итерации. Результат финальной итерации содержит всю требуемую функциональность продукта. Таким образом, с завершением каждой итерации продукт получает приращение к его возможностям, которые развиваются эволюционно.

Спиральная модель. Спиральная модель была разработана в середине 1980-х годов Барри Боэ. При использовании этой модели ПО создается в несколько итераций методом прототипирования.

Каждая итерация соответствует созданию фрагмента или версии ПО, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов и планируются работы следующей итерации.

На каждой итерации оцениваются:

– риск превышения сроков и стоимости проекта;

– необходимость выполнения ещё одной итерации;

– степень полноты и точности понимания требований к системе;

– целесообразность прекращения проекта.

Важно понимать, что спиральная модель является не альтернативой итерационной модели, а специально проработанным вариантом.

## 1.3 Жизненный цикл ПО

Жизненный цикл программного обеспечения – период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации.

Стандарт ГОСТ 34.601-90 предусматривает следующие стадии и этапы создания автоматизированной системы (АС):

1) формирование требований к АС;

а) обследование объекта и обоснование необходимости создания АС;

б) формирование требований пользователя к АС;

в) оформление отчета о выполнении работ и заявки на разработку АС;

2) разработка концепции АС;

а) изучение объекта;

б) проведение необходимых научно-исследовательских работ;

в) разработка вариантов концепции АС и выбор варианта концепции АС, удовлетворяющего требованиям пользователей;

г) оформление отчета о проделанной работе;

3) техническое задание;

а) разработка и утверждение технического задания на создание АС;

4) эскизный проект;

а) разработка предварительных проектных решений по системе и её частям;

б) разработка документации на АС и её части;

5) технический проект;

а) разработка проектных решений по системе и её частям;

б) разработка документации на АС и её части;

в) разработка и оформление документации на поставку комплектующих изделий;

г) разработка заданий на проектирование в смежных частях проекта

6) рабочая документация;

а) разработка рабочей документации на АС и её части;

б) разработка и адаптация программ;

7) ввод в действие;

а) подготовка объекта автоматизации;

б) подготовка персонала;

в) комплектация АС поставляемыми изделиями (программными и техническими средствами, программно-техническими комплексами, информационными изделиями);

г) строительно-монтажные работы;

д) пусконаладочные работы;

е) проведение предварительных испытаний;

ж) проведение опытной эксплуатации;

з) проведение приёмочных испытаний;

8) тестирование АС;

9) сопровождение АС;

а) выполнение работ в соответствии с гарантийными обязательствами;

б) послегарантийное обслуживание.

# 2 ПРАКтическая часть

## 2.1 Выделение объектов участвующих в задаче

Данная работа представляет собой создание многим знакомой игры – «змейка» в консоли. Так как игра будет создаваться для консоли в ней можно выделить такие объекты как поле, которое будет ограничивать пространство, змейку, которая должна увеличиваться пропорционально съеденным яблокам, а также можно отдельно вынести объект координат, так как он потребуется для вышеперечисленных, некую базу данных, которая будет хранить данные о положении змейки на поле, координаты каждого фрагмента пресмыкающегося, положение монеток, а также некоторую информацию, например о набранных очках. Также отдельно стоит вынести структуру предоставляющую связь между пользователем и системой.

## 2.2 Описание выделенных объектов в виде иерархии классов в нотации UML

Выделение основных функций, классов присущих системе:

1) класс актера, персонажа (Actor)

– добавление яблок на поле;

– появление в некоторых первоначальных координатах;

– перемещение по полю;

– зоны срабатывания для смерти или подбора монетки;

– видимость курсора;

– отчистка внутренних данных;

– перезагрузка;

– информация для пользователя о текущем сеансе.

2) класс игрового поля (Border)

– инициализация поля;

– вывод поля на экран;

3) класс координат (Coord)

– функция перемещения на заданные координаты;

– инициализация;

– оптимизация операторов;

– функции получения координат.

4) класс базы данных (Database)

– функции вывода необходимой информации;

– добавление информации в базу данных;

– сохранение полученных данных;

– функции возврата некоторой информации;

– обновление уже существующих данных.

5) класс для взаимодействия пользователя и системы (User\_system)

– вывод информации, о том, как взаимодействовать с приложением;

– функция получения пользовательского ввода;

– функция проверки на ошибки ввода;

– предоставление некоторых игровых настроек.

6) функция непрерывного запроса на ввод.

Графическая интерпретация классов представлена на рисунке 3.



Рисунок 3 – Иерархия классов в нотации UML

## 2.3 Описание взаимодействия объектов в нотации UML

Класс User\_system и Actor создают в своем теле экземпляр класса Database, чтобы получать и обновлять информацию, хранящуюся за пределами приложения. Также класс User\_system является поставщиком для класса Actor и предает введенные пользователем данные о размерах поля и скорости игры (при отсутствии ввода передает дефолтные значения). По мимо класса Database, класс Actor создает в своем теле экземпляр структуры Border, чтобы при перезапуске перерисовывать игровое поле. Как было сказано ранее для функциональности класс Coord был вынесен как отдельный тип, так как он необходим для работы сразу двух важных структур Border и Actor. Также класс Actor создает поток для постоянного запроса на ввод, необходимы для управление направлением движения змейки. Графическая схема взаимодействия объектов в нотации UML представлена на рисунке 4.



Рисунок 4 – Взаимодействие объектов в нотации UML

## 2.4 Реализация получившихся классов

Ссылку на полный листинг кода можно найти в приложении А.

Если посмотреть на функцию main() в файле Snake.cpp (рисунок 5), то можно увидеть что создается экземпляр User\_system и вызывается метод user\_input(), после чего выполняется только код классов.

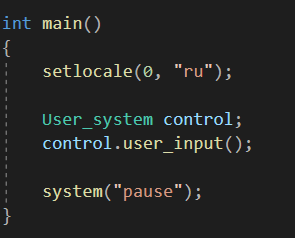


Рисунок 5 – Функция main()

Метод user\_input() выводит пронумерованное меню: играть, настройки, выход. Выбрав пункт настройки, программа передает выполнение методу settings(), который в свою очередь предлагает задать размер поля или выбрать сложность игры. Причем на каждый запрос ввода значений организована проверка на правильность введенных данных, и возможность вновь ввести значения оставаясь в том же меню. Так как пользователь должен вводить только цифровые значения проверка на символы будет везде одинаковой, поэтому была создана встраиваемая функция chec\_error():

inline void User\_system::check\_err()

{

if (cin.fail())

{

cin.clear();

cin.ignore();

throw 1;

}

}

Как только пользователь выберет пункт «Играть» User\_system создаст экземпляр класса Actor и передаст все необходимые значения ( размеры игрового поля (x, y), сложность игры (speed), имя пользователя (name), и его положение в структуре JSON (numb)), в случае если пользователь не менял их они остаются дефолтными, после чего все управление переходит классу Actor вплоть до завершения программы.

При создании экземпляра выполняется конструктор в соответствии с переданными параметрами, в данном случае вызванный конструктор инициализирует экземпляр класса Border, вызывает его метод для прорисовки полей создает поток, который через функцию \_getch() будет обрабатывать каждый введенный символ и передавать его в switch, которая в свою очередь меняет глобальную переменную rot, отключает видимость курсора (CursorVisible(false)), помещает змейку в центр поля и добавляет первоначальную монетку на поле (spawn()), выводит информацию с правого бока от игрового поля (info()), а также вызывает функцию move(rot), которая будет управлять всеми последующими действиями (рисунок 6).

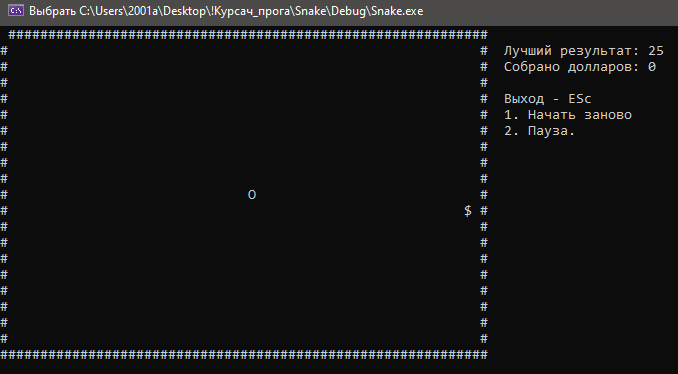


Рисунок 6 – Инициализация Actor

Видимое передвижение змейки реализовано путем удаления хвоста и перемещения головы в заданном направлении именно это и делает функция move():

void Actor::move(int& rot)

{

Coord coord;

up:

if (rot == 5)

fail = true;

Sleep(speed);

while (fail)

{

switch (rot)

{

case 1:

coord.init(x, y -= 1);

trigger(coord);

Sleep(speed+40);

break;

case 2:

coord.init(x, y += 1);

trigger(coord);

Sleep(speed+40);

break;

case 3:

coord.init(x -= 1, y);

trigger(coord);

Sleep(speed);

break;

case 4:

coord.init(x += 1, y);

trigger(coord);

Sleep(speed);

break;

case 5: // заново

system("cls");

Clear();

Reload();

Sleep(1000);

break;

case 6: // пауза

coord.GoToXY(0, \_Y + 1);

system("pause");

break;

}

}

coord.GoToXY(0, \_Y+1);

goto up;

}

Можно заметить, что при передвижении по вертикали, ко времени задержки прибавляется 40 мили секунд, это необходимо для того, чтобы сравнять скорость по горизонтали и вертикали т.к. символы по вертикали имеют больший отступ друг от друга нежели по оси x.

Функция trigger() вызываемая при каждом изменении координат изначально проверяет произошло ли столкновение с полем, частью змейки, завершилась ли игра или же была собрана монетка, в последнем случае вызывается обновление боковой информации, которое запрашивает через класс Database данные о лучшем результате из файла JSON, при превышении лучшего результата вызывается функция update() класса Database, которая обновляет данные внешнего файла. А также функция trigger() производит добавление новой монетки – add\_apple():

void Actor::add\_apple(int X, int Y)

{

srand(time(0));

int x = rand() % (X-2) + 1;

int y = rand() % (Y-2) + 1;

Coord coord(x, y);

for (int i = 0; i < v\_ABody.size(); ++i)

{

if (v\_ABody[i] == coord)

{

for (int i = 1; i < \_X - 2; ++i)

{

for (int j = 1; j < \_Y - 2; ++j)

{

coord.init(i, j);

for (int k = 0; k < v\_ABody.size(); ++k)

{

if (v\_ABody[k] == coord)

{

break;

} else

if (k == v\_ABody.size() - 1)

{

i = \_X;

j = \_Y;

}

}

}

}

}

}

Get\_apple().push\_back(coord);

coord.GoToXY(coord);

std::cout << '$';

}

Здесь происходит проверка на то, чтобы монетки не появлялись в теле змейки.

Также стоит упомянуть об оставшихся трех функциональных символов – это клавиша Esc, единица и двойка.

Клавиша Esc осуществляет выход из программы путем вызова стандартного метода exit(). Пауза программы осуществляется вызовом функции system(“pause”). При выборе первого пункта «заново» вызываются функции Clear() и Reload(), задача которых отчистить переменные и подготовить поле для нового сеанса.

## 2.4 Программа, показывающая взаимодействие описанных объектов.

При запуске программы пользователь попадает в главное меню, где на выбор предлагается три пункта: «Играть», «Настройки», «Выход», все это предоставляет класс User\_system (рисунок 7).



Рисунок 7 – Главное меню

Перейдя в настройки, пользователь может задать значение игрового поля или выбрать один из трех уровней сложности, по-прежнему данные предоставляется классом User\_system (рисунок 8).

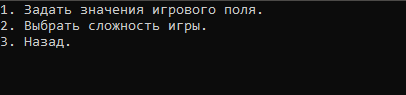


Рисунок 8 – Меню настроек

Как только выбирается пункт «Играть», все управление переходит в класс Actor вплоть до выхода из игры (рисунок 9).

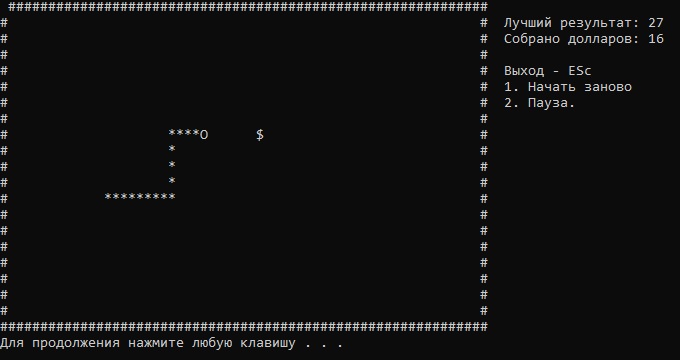


Рисунок 9 – Игровой процесс

Класс Actor вызывает структуру Border каждый раз, когда игра начинается заново. Когда игрок терпит поражение, одним из действий является проверка на лучший результат, и если текущий результат превосходит предыдущий, то вызывается метод update() класса User. Класс Coord вызывается в Actor и Border в основном для отрисовки данных соответствующих классам.

# Заключение

Соблюдая принципы объектно-ориентированного программирования можно значительно улучшить управляемость процесса моделирования, например, минимизировать избыточность данных, повысить их целостность и удобство в понимании как самим разработчиком так и тем кто будет сопровождать или изучать ваш проект.

Различные диаграммы играют не маловажную роль как при начальных этапах проектирования приложения, так и при его презентации, также же они могут облегчить понимание структуры, связей и логики больших проектов.

Оформление курсовой работы [4]

# Список ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) Объектно-ориентированное программирование: сайт [Электронный ресурс]. URL: <https://ru.wikipedia.org/wiki/Объектно-ориентированное> программирование (дата обращения: 30.05.2021).

2) Абстракция\_данных: сайт [Электронный ресурс]. URL:

[https://ru.wikipedia.org/wiki/Абстракция\_данных](https://ru.wikipedia.org/wiki/Абстракция_данных 3)

[3](https://ru.wikipedia.org/wiki/Абстракция_данных 3)) Наследование: сайт [Электронный ресурс]. URL: <https://ru.wikipedia.org/wiki/Наследование_(программирование)>

4) Гопкало В.Н. Выпускная квалификационная работа. Общие требования и правила оформления: мтод. пособие. / В.Н. Гопкало, О.А. Графский. – Хабаровск: Изд-во ДВГУПС, 2014. – 44 с.