1. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 1.1. Постановка задачи

Необходимо разработать программное обеспечение для реализации автоматизированной системы учета услуг автотранспортного предприятия.

Требования к приложению:

1) Выбор пользовательских функций – Закладки;

2) Вариант диаграммы – Столбиковая;

3) Каталоги с файлами БД не определяются в программе;

* 1. Жизненный цикл программного обеспечения

Понятие жизненного цикла ПО (ЖЦ ПО) является одним из базовых понятий программной инженерии.

ЖЦ ПО — это период времени, который начинается с момента решения о необходимости создания ПО и заканчивается полным изъятием его из эксплуатации.

Основным нормативным документом, регламентирующим состав процессов ЖЦ ПО, является международный стандарт ISO/IEC 12207: 1995 «Information Technology - Software Life Cycle Processes». Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО (его российский аналог ГОСТ РИСО/МЭК 12207-99 введен в действие в июле 2000 г.). В данном стандарте процесс определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными от других процессов, и результатами. Каждый процесс разделен на набор действий, каждое действие — на набор задач.

Каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем не существует заранее определенных последовательностей выполнения (естественно, при сохранении связей по входным данным).

Модель жизненного цикла программного обеспечения — структура, содержащая процессы действия и задачи, которые осуществляются в ходе разработки, использования и сопровождения программного продукта. При выборе схемы модели жизненного цикла для конкретной предметной области, решаются вопросы включения важных для создаваемого продукта видов работ или не включения несущественных работ. К широко используемым типам моделей ЖЦ относятся следующие: каскадная, спиральная, инкрементная, эволюционная, стандартизованная и др .

В данной работе я выбрал каскадную модель. Согласно данной модели ЖЦ работы и задачи процесса разработки обычно выполняются последовательно. Однако вспомогательные и организационные процессы обычно выполняются параллельно с процессом разработки. В данной модели возвращение к начальному процессу предусматривается после сопровождения и исправления ошибок.

Особенность такой модели состоит в фиксации последовательных процессов разработки программного продукта. В ее основу положена модель фабрики, где продукт проходит стадии от замысла до производства, затем передается заказчику как готовое изделие, изменение которого не предусмотрено, хотя возможна замена на другое подобное изделие в случае рекламации или некоторых ее деталей, вышедших из строя.

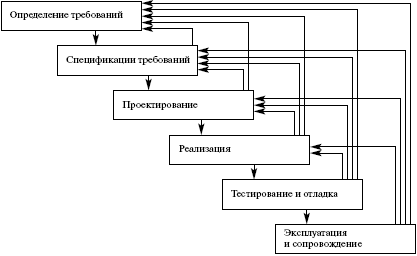


Рис 1. Каскадная модель ЖЦ

Недостатки этой модели:

• процесс создания ПС не всегда укладывается в такую жесткую форму и последовательность действий;

• не учитываются изменившиеся потребности пользователей, изменения во внешней среде, которые вызовут изменения требований к системе в ходе ее разработки;

• большой разрыв между временем внесения ошибки (например, на этапе проектирования) и временем ее обнаружения (при сопровождении), что приводит к большой переделке ПС. При применении каскадной модели имеют место следующие факторы риска:

• требования к ПС недостаточно четко сформулированы, либо не учитывают перспективы развития ОС, сред и т.п.;

• большая система, не допускающая компонентной декомпозиции, может вызвать проблемы с размещением ее в памяти или на платформах, не предусмотренных в требованиях;

• внесение быстрых изменений в технологию и в требования может ухудшить процесс разработки отдельных частей системы или системы в целом;

• ограничения на ресурсы (человеческие, программные, технические и др.) в ходе разработки могут сузить отдельные возможности реализации системы; Полученный продукт может оказаться плохим для применения по причине недопонимания разработчиками требований или функций системы или недостаточно проведенного тестирования. Преимущества реализации системы с помощью каскадной модели, следующие:

• все задачи подсистем и системы реализуются одновременно (ни одна задача не забыта), что способствует установлению стабильных связей и отношений между ними;

• полностью разработанную систему с документацией на нее легче сопровождать, тестировать, фиксировать ошибки и вносить изменения не беспорядочно, а целенаправленно, начиная с требований (например, добавить или заменять некоторые функции) и повторить процесс. 11 Разработанное ПО основано на каскадной модели. Это обосновано тем, что каждая работа выполняется полностью, и после ее завершения и перехода к следующему этапу возвращение к предыдущему не требуется. Промежуточный результат проверяется известными методами верификации и фиксируется в качестве готового эталона для следующего процесса

* 1. Разработка базы данных Методика определения и документирования требований к базе данных заключается в составлении словаря данных. Словарь данных перечисляет и определяет отдельные элементы данных, которые должны храниться в базе (табл. 1).

Таблица 1

|  |  |
| --- | --- |
| Элементы данных | Описание |
| Код клиента | Описание клиента |
| ФИО |
| Адрес |
| Номер |
| Цвет | Описание транспортного средства |
| Гос. номер |
| Марка |
| Модель |
| Дата заказа | Информация о услуге |
| Дата прибытия |
| Расстояние |
| Сумма |
| Код водителя | Описание водителя |
| ФИО |
| Адрес |
| Номер |

Далее модель развивается путем определения атрибутов для каждого объекта. Для этого из составленного ранее словаря данных выделяем необходимые элементы (табл.2).

Таблица 2. Объекты и атрибуты

|  |  |  |  |
| --- | --- | --- | --- |
| Клиент | Водитель | Услуга | Транспортное средство |
| ФИО | ФИО | Дата заказа | Цвет |
| Адрес | Адрес | Дата прибытия | Гос. номер |
| Номер | Номер | Расстояние | Марка |
|  |  | Сумма | Модель |

Для того чтобы данные схемы стала реляционной моделью необходимо использование ключей (первичных и внешних) и отношений

Программы для создания UML таблиц и дизайна

Программа Draw.io

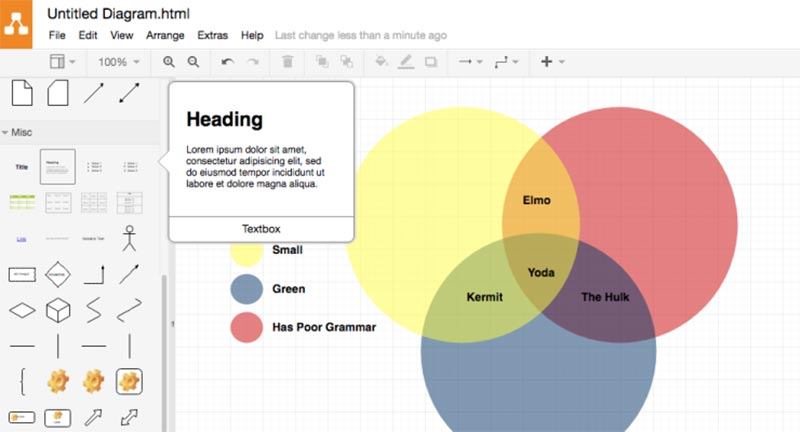
diagrams.net **-** диаграмма.net является кроссплатформенным 

Рис. 2. Интерфейс Draw.io

программным обеспечением для рисования графиков, разработанным в HTML5 и JavaScript. Его интерфейс можно использовать для создания диаграмм, таких как блок-схемы, каркасы, диаграммы UML, организационные диаграммы и сетевые диаграммы.



Рис. 3. Эмблема Draw.io

Программа Figma

Figma — онлайн-сервис для разработки интерфейсов и прототипирования с возможностью организации совместной 

Рис. 4. Интерфейс Figma

работы в режиме реального времени. Сервис доступен по подписке, предусмотрен бесплатный тарифный план для одного пользователя. Имеются офлайн-версии для Windows, macOS.

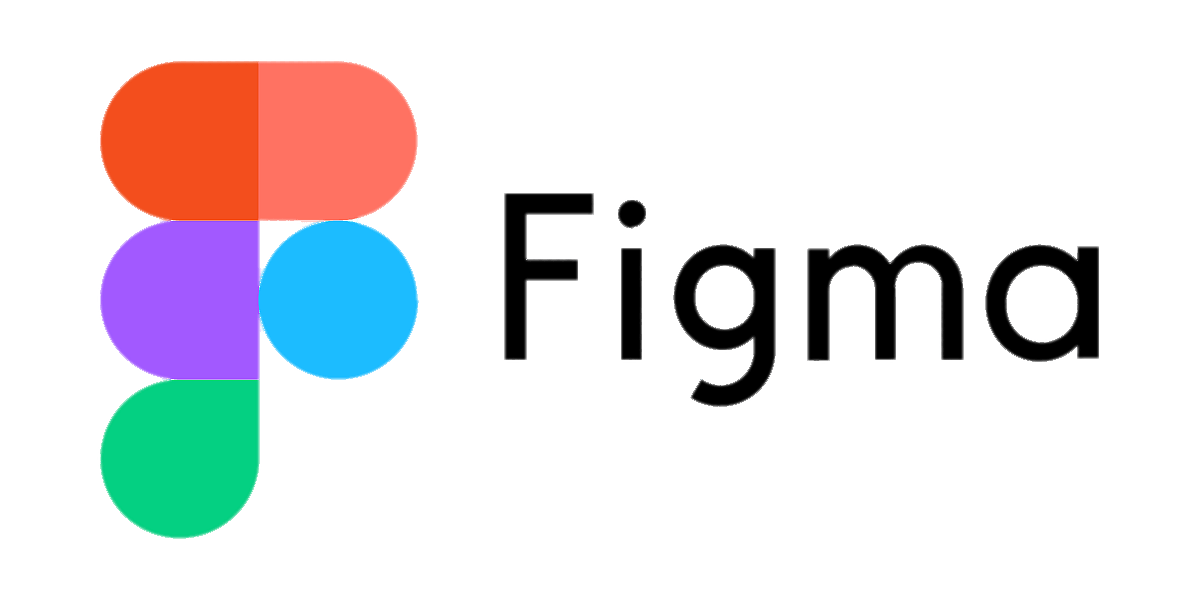


Рис. 5. Эмблема Figma

1. UML диаграммы

2.1 Как появились UML диаграммы? Официально создание UML началось в октябре 1994 года, когда Рамбо перешел в компанию Rational Software, где работал Буч. Первоначальной целью было объединение методов Буча и ОМТ. Первая пробная версия 0.8 Унифицированного Метода (Unified Method), как его тогда называли, появилась в октябре 1995 года. Приблизительно в это же время в компанию Rational перешел Джекобсон, и проект UML был расширен с целью включить в него язык OOSE. В результате совместных усилий в июне 1996 года вышла версия 0.9 языка UML. На протяжении всего года создатели занимались сбором отзывов от основных компаний, работающих в области конструирования программного обеспечения. За это время стало ясно, что большинство таких компаний сочло UML языком, имеющим стратегическое значение для их бизнеса. В результате был основан консорциум UML, в который вошли организации, изъявившие желание предоставить ресурсы для работы, направленной на создание полного определения UML.

Версия 1.0 языка появилась в результате совместных усилий компаний Digital Equipment Corporation, Hewlett Packard, I-Logix, Intellicprp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments и Unisys. UML 1.0 оказался хорошо определенным, выразительным, мощным языком, применимым для решения большого количества разнообразных задач. В январе 1997 года он был представлен Группе по управлению объектами (Object Management Group, OMG) на конкурс по созданию стандартного языка моделирования.

2.2 Что такое UML диаграммы? UML (с английского аббревиатура расшифровывается как Unified Modeling Language — унифицированный язык моделирования) — это способ наглядно описать архитектуру, проектирование и реализацию комплексных программных систем. Код типичного приложения включает в себя тысячи строк, за связями и иерархиями которых очень непросто уследить. С помощью диаграмм UML структуру программы можно разделить на компоненты и подкомпоненты.

2.3Для чего нужны диаграммы UML? UML — стандартизированный язык моделирования. Он совместим с разными языками программирования и процессами разработки, а потому большинству программистов не составит труда понять и применить его на практике.

Хотя многие программисты боятся диаграмм как огня, схематизация всё же очень полезна в гибкой разработке и позволяет [сосредоточиться на проекте и получить больше отдачи](https://www.lucidchart.com/blog/why-developers-should-view-diagrams-as-core-documentation). Вместо того чтобы рассуждать, как здорово было бы воспользоваться диаграммами UML, просто включите их в документацию в качестве основного компонента. Диаграммы UML помогут разработчикам:

* оперативно ввести в курс дела новых сотрудников или коллег из других отделов;
* легко сориентироваться в исходном коде;
* основательно спланировать новые функции, прежде чем взяться за программирование;
* доступным языком объяснить материал аудиториям с разными уровнями технической подготовки.

Безусловно, от схем, которые не развиваются вместе с проектом, мало толку, поэтому документацию необходимо постоянно обновлять. Поскольку Lucidchart работает в «облаке», это уже большой плюс. Lucidchart позволяет генерировать [диаграммы последовательностей UML](https://www.lucidchart.com/pages/uml/uml-sequence-markup) непосредственно из текстовой разметки, открывая вам широкие возможности для автоматизации и гибкой работы.

2.4 Виды UML диаграммы Тем, кто мало знаком с диаграммами UML, может показаться, что их разновидностям нет числа, но это не так. Стандарты UML признают 13 видов диаграмм, которые делятся на две группы, как указано ниже.

2.5 Диаграмма компонентов [Диаграмма компонентов](https://www.lucidchart.com/pages/uml-component-diagram) — по сути, более подробная версия диаграммы классов: и в той, и в другой действуют одни и те же правила. Диаграмма компонентов позволяет разбить комплексную систему на более мелкие составляющие и наглядно продемонстрировать установленные между ними связи.

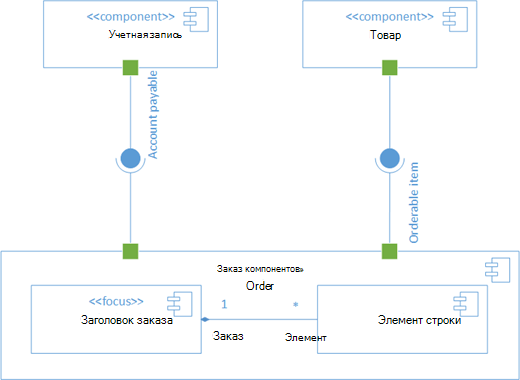


Рис. 6. Диаграмма компонентов

#### 2.6 Диаграмма последовательностей

[Диаграммы последовательностей](https://www.lucidchart.com/pages/uml-sequence-diagram), которые иногда также называют диаграммами (или сценариями) событий, показывают, в каком порядке происходит взаимодействие между объектами. С помощью таких диаграмм довольно легко составлять незамысловатые сценарии рабочих циклов.

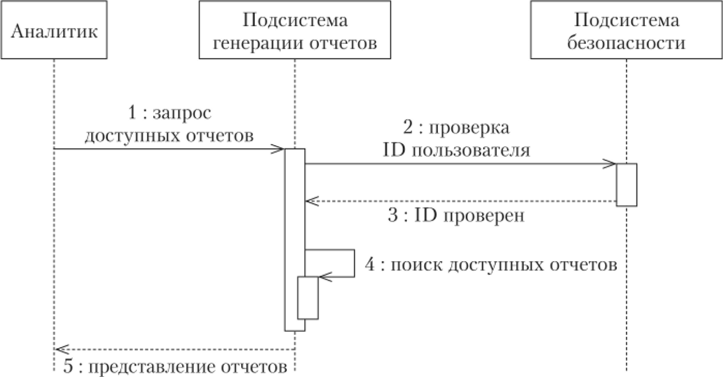


Рис. 7. Диаграмма последовательностей

2.7 Диаграмма объектов [Диаграммы объектов](https://www.lucidchart.com/pages/uml-object-diagram) дают нам примеры того, как структуры данных выглядят в конкретный момент времени. Если статичную структуру удобно представить в виде диаграммы классов, то диаграммы объектов, как контрольные примеры, позволяют судить, все ли ее компоненты на месте. Из диаграммы объектов также можно почерпнуть полезную информацию об элементах модели и их ссылках.

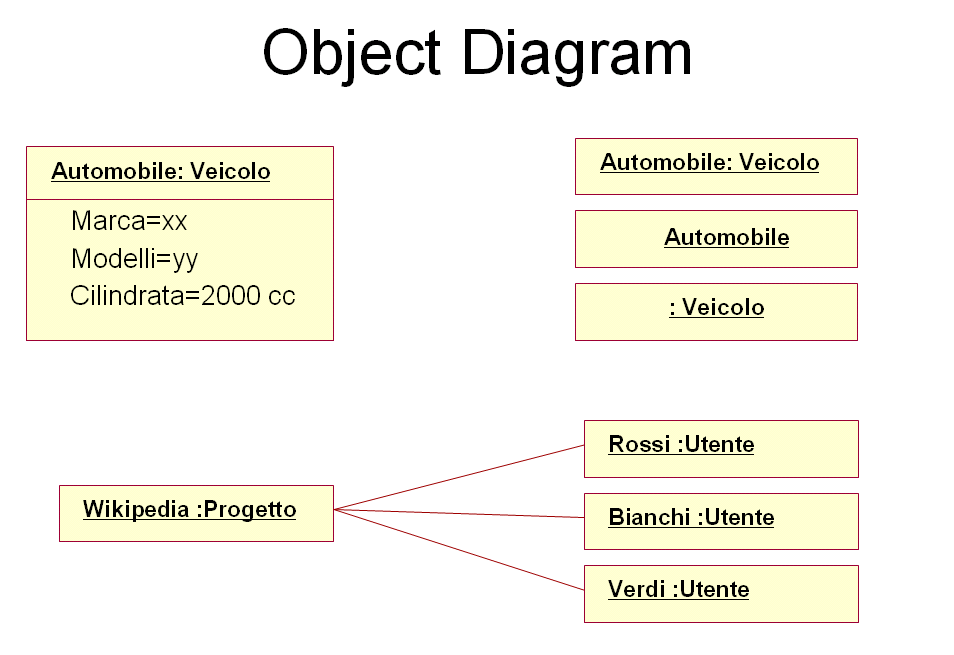


Рис. 8. Диаграмма объектов

2.8 Как создать диаграмму UML Диаграммы UML подчиняются конкретному набору правил и требуют применения стандартных фигур, поэтому освоение каждой их разновидности отнимает массу времени. К счастью, мы подготовили для вас серию [доступных уроков](https://www.lucidchart.com/pages/uml-class-diagram?a=1), где всё, начиная с диаграмм классов, разложено по полочкам и разбито на простые шаги.

Функции в UML

Сущности – это абстракции, которые являются основными элементами модели, связи соединяют их между собой, а диаграммы группируют представляющие интерес наборы

Поведенческие сущности – динамические части моделей UML. Это «глаголы» моделей, представляющие поведение модели во времени и пространстве. Основной из них является **взаимодействие** – поведение, которое заключается в обмене сообщениями между наборами объектов или ролей в определенном контексте для достижения некоторой цели. Сообщение изображается в виде линии со стрелкой, почти всегда сопровождаемой именем операции.  
   
Поведенческие сущности

Рис. 9. Поведенческие сущности  
Аннотирующие сущности – это поясняющие части UML-моделей, иными словами, комментарии, которые можно применить для описания, выделения и пояснения любого элемента модели. Главная из аннотирующих сущностей – примечание. Это символ, служащий для описания ограничений и комментариев, относящихся к элементу либо набору элементов. Графически представлен прямоугольником с загнутым углом; внутри помещается текстовый или графический комментарий.  
   


Агрегация– особая разновидность ассоциации, представляющая структурную связь целого с его частями. Как тип ассоциации, агрегация может быть именованной.

Агрегация

Рис. 10. Агрегация

Композиция — более строгий вариант агрегации. Известна также как агрегация по значению.  
Композиция – это форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого.

Композиция

Рис. 11. Композиция

Класс – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой.  
  
Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями:

* имя класса
* атрибуты (свойства) класса
* операции (методы) класса.

Для атрибутов и операций может быть указан один из трех типов видимости:

* **—** — private (частный)
* **#** — protected (защищенный)
* **+** — public (общий)

Первая из них – зависимость – семантически представляет собой связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого). Графически представлена пунктирной линией, иногда со стрелкой, направленной к той сущности, от которой зависит еще одна; может быть снабжена меткой.  
   
Зависимость

Рис. 12. Зависимость