Содержание

Весь код описанный в данном руководстве будет расположен по git репозитории по адресу:
https://github.com/DmitriyODS/laboratory-work-on-VP
Шаблон кода – библиотека, была построена по образцу 18 – ого варианта Л/Р

Разбор принципов и методов выполнения заданий из серии лабораторных работ по высокоуровневому программированию за второй семестр.

Автор – Осиповский Д. С.

Общие положения

В рамках лабораторных работ по высокоуровневому программированию за второй семестр вам предлагается реализовать свой собственный проект, который является «Автоматизированной системой» по работе с какими-то данными из той или иной отрасли. Данная работа призвана выработать навыки по решению практических задач из той предметной области, на которую делается больший упор в изучении. В данной случае, выполняя лабораторную работу, вы будете максимально приближены к реальным проектам, с которыми вам придётся столкнуться на своей будущей работе по специальности. Также работа поможет выработать у вас навыки решения нестандартных задач, разработки алгоритмов и архитектуры промышленных продуктов, обоснованного выбора пути разработки.

В данном руководстве по решению задач будет рассмотрено решение одного из вариантов лабораторных работ. Стоит отметить, что все варианты являются однотипными и принципы архитектуры у них схожи. На основании этого мы рассмотрим только один вариант целиком, а также добавим информацию по конкретным дополнительным случаям, которые являются особенными для других вариантов. Данное руководство будет полезно тем, кто столкнулся с проблемами в решении заданий, а также тем, кто хочет сравнить свои результаты. ***Данное руководство не является пособием, код в нём не является эталонном для реализации. Из этого документа вы должны взять только принципы размышления и построения архитектуры программы, сам код и реализация итогового продукта может в корне отличаться от приведённой здесь.*** Также стоит отметить, что руководство не стремится сделать привязку к языку C++, несмотря на то что код итогового продукта будет писаться на нём, это не мешает вам реализовать приведённые алгоритмы на других языках и перенести в них архитектуру, разработанную здесь. Точно также ***руководство не является справочником по языку C++***: здесь будут описаны и пояснены только те операторы и функции языка, которые будут непосредственно задействованы, за остальной справочной информацией стоит обратиться на следующие ресурсы и схожие с ними: *METANIT.COM; RAVESLI.COM.*

Весь код описанный в данном руководстве будет расположен по git репозитории по адресу: https://github.com/DmitriyODS/laboratory-work-on-VP
Шаблон кода – библиотека, была построена по образцу 18 – ого варианта Л/Р

Техническое задание

В рамках лабораторных работ по ВП вам предложено написать программный продукт из серии «Автоматизированная система». Эта задача была разбита на шесть частей (л/р), в каждой части вам предлагается доработать свой продукт, добавив новый функционал или изменив существующий. Помимо этого, каждая Л/Р освещает конкретную тему из области изучения ООП на языке С++. Таким образом, на каждой Л/Р вы знакомитесь с новыми приёмами использования средств языка для решения поставленной задачи.

Каждая Л/Р разбивает задание на подзадания, которые являются отправными точками реализации вашей задачи. Подзадания Л/Р не несут алгоритмический смысл и не являются руководством к действию. Иными словами, в подзаданиях не указаны методы и способы выполнения задания, а даны лишь общие рекомендации к выполнению. Это позволяет запустить мыслительный процесс и получить опыт разработки архитектуры большого продукта.

В серии Л/Р предоставляется 20 вариантов программ, архитектура которых схожа, но прикладное применение различно. Варианты:

1. Автоматизированная система института
2. Автоматизированная система школы
3. Автоматизированная система парикмахерской
4. Автоматизированная система авто – салона
5. Автоматизированная система банка
6. Автоматизированная система мастерской
7. Автоматизированная система кинотеатра
8. Автоматизированная система диалога (чат - бот)
9. Автоматизированная система поликлиники
10. Автоматизированная система почты
11. Автоматизированная система гостиницы
12. Автоматизированная система завода
13. Автоматизированная система организации музыкальной коллекции
14. Автоматизированная система электронной почты
15. Автоматизированная система организации коллекции фотографий
16. Автоматизированная система вокзала
17. Автоматизированная система магазина
18. Автоматизированная система космодрома
19. Автоматизированная система библиотеки
20. Автоматизированная система аэропорта

Как видно из списка, все продукты – это автоматизированная система, как было сказано ранее. Для всех этих автоматизированных систем можно выделить общие функции, которыми они будут снабжены, а также общие принципы работы. Давайте рассмотрим их.

Принципы и функции, которые должны быть реализованы в каждом варианте Л/Р:

- Сущности (модели данных). Сущность (модель данных) – это логическая бизнес – единица информации, которая будет передаваться между частями вашей программы. К сущностям можно отнести: пользователя, администратора, учителя, преподавателя, студента, мастера, продукт, товар и т д. Реализация сущностей на

языке C++ в рамках объектно – ориентированного подхода к программированию должна представлять из себя класс, который будет иметь поля, необходимые для хранения информации о данной сущности (поля – это переменные, которые находятся внутри класса – сущности). Также каждый класс должен иметь методы: это те функции, которые будут работать с данными данной сущности (полями – переменными, содержащимися в этом классе). Для примера рассмотрим класс – сущность: «Пользователь»:

```
#include <string>
#include <iostream>


using std::string;
using std::cin;
using std::cout;



// класс - сущность "пользователь"
// данный класс имеет поля (переменные)
// и методы (функции), по работе с ними
class User {
public:
	// модификатором "public" - мы открываем доступ к методам (функциям)
	// которые мы сможем вызывать у объекта класса, через точку
	// в методы отправятся следующие функции: получение/изменение
	// значений полей класса
	// конструктор (инициализатор) нового объекта класса, деструктор -
	// - метод, который вызывается при удалении объекта класса
	// а также, дополнительные функции (методы), по работе с объектом:
	// печать информации о пользователи, увеличение его возраста на ед.

	// констурктор класса
	User(string name, string surname, size_t age)
		: m_name(std::move(name)), m_surname(std::move(surname)), m_age(age) {

		// пустое тело метода - конструкора
		// вся инициализация вынесена в блок инициализации (после двоеточия)
		// подробнее про списк инициализации прочитайте на ravesli.com
		// также, в списке инициализации задействовано такое понятие,
		// как перемещение объекта: std::move, про него тоже прочитайте
		// на ravesli.com

	}

	// метод изменения поля - имя
	void setName(const string name) {
		m_name = name;
	}

	// метод изменения поля - фамилия
```

```
        void setSurname(const string surname) {
                m_surname = surname;
        }

        // метод изменения поля - возраст
        void setAge(size_t age) {
                m_age = age;
        }

        // метод получения значения поля - имя
        string getName() {
                return m_name;
        }

        // метод получения значения поля - фамилия
        string getSurname() {
                return m_surname;
        }

        // метод получения значения поля - возраст
        size_t getAge() {
                return m_age;
        }

        // метод печати на консоле информации о пользователе
        void print() {
                cout << m_name
                        << '\n'
                        << m_surname
                        << '\n'
                        << m_age
                        << std::endl;
        }

        // метод увеличения возраста пользователя на один год
        void addOneYaer() {
                ++m_age;
        }

private:
        // модификатором доступа "private" - мы закрываем доступ к переменным
        // класса из вне, для защиты этих данных
        // доступ к ним, будет осуществляться через методы получения и установки
        // значения, известные также, как "set", and "get"

        string m_name{}; // переменная (поле), которая хранит имя пользователя
        string m_surname{}; // переменная (поле) для фамилии
        size_t m_age{}; // переменная (поле) - возраст
};
```

Весь код описанный в данном руководстве будет расположен по git репозитории по адресу:
https://github.com/DmitriyODS/laboratory-work-on-VP
Шаблон кода – библиотека, была построена по образцу 18 – ого варианта Л/Р

Как видно, этот класс представляет из себя набор данных и методов, которые относятся к определённой сущности – «пользователь». Теперь в рамках нашей программы мы сможем с лёгкостью работать с этими данными.

Таким образом, в каждом варианте Л/Р будет находиться несколько сущностей, которыми и будет управлять наша программа. Сущности по-другому называются моделями. Это предпочтительное название для директории, в которой будут лежать все модели программы. Стоит отметить, что каждая модель – сущность – класс должен состоять из двух файлов на языке C++: .h и .cpp. Эти файлы должны находиться в папке, которая называется так же, как эти файлы, а эти файлы должны называться так же, как сущность, которая в них описана. Например, для сущности User файлы будут называться: user.h и user.cpp, а папка, в которой они будут находиться, «user». В свою очередь эта папка должна находиться в папке под названием «models» – модели. Таким образом мы строим бизнес логику нашего продукта.

- Теперь поговорим о функциях, которые должны присутствовать в каждой программе. Раз у нас есть сущности, то должны быть и функции, которые с этими сущностями работают. Опишем все эти функции: каждая функция применима и должна быть реализована во всех вариантах Л/Р:
    - Функции создания объектов сущностей. Данные функции будут создавать новые объекты классов (сущностей – моделей) и возвращать созданный объект. В рамках языка C++ они должны будут возвращать указатель на динамически созданный объект. Какой функционал они должны иметь? Каждая функция должна получать данные и формировать на их основе новый объект. Откуда она будет получать эти данные? Нужно предусмотреть возможность получения информации от пользователя (с консоли), а также из параметров функций, в которые прокидываются нужные значения. В качестве примера рассмотрим функцию создания объекта пользователя:

```
// структура для хранения данных пользователя
// которые будут помещены в класс пользователя
struct UserData {
    string name{};
    string surname{};
    size_t age{};
};

string getStringEnter() {

    // тело функции
    // функция получения пользовательского ввода
    // проверке его на ошибки
    // и возврату корректного значения

    return string();
}

size_t getIntEnter() {

    // тело функции
    // функция получения пользовательского ввода
    // проверке его на ошибки
    // и возврату корректного значения

    return 0;
}

// функция создания нового пользователя
// принимает структуру данных пользователя, как указатель
// если указатель равен нулю, то мы запрашиваем данные
// у пользователя с консоли
User* createUser(const UserData* userData = nullptr) {
    if (userData) {
            return new User(userData->name, userData->surname, userData->age);
    }

    cout << "Введите имя нового пользователя:" << std::endl;

    // вызов функции проверки пользовательского ввода
    // которая возвращает корректную строку
    // функцию можно реализовать любым способом
    string name = getStringEnter();
    string surname = getStringEnter();
    size_t age = getIntEnter();

    return new User(name, surname, age);
}
```

В качестве вспомогательного функционала для функции по созданию пользователей была реализована структура и две дополнительные функции. Структура хранит набор данных для полей сущности «пользователь», а функции нужны для вынесения универсальной функциональности проверки пользовательского ввода в отдельное место. Благодаря этому мы сможем использовать один и тот же код для других функций, которые должны будут запрашивать данные у пользователя из консоли.

Таким образом, мы должны реализовать функции по созданию объектов для всех моделей, которые есть в нашей программе. Функционал у них будет однотипный.

- Функции редактирования объекта модели. Наравне с функциями создания объекта модели вам необходимо реализовать функции редактирования. Это функции, которые будут схожи по функциональности с созданием нового объекта, за тем исключением, что будут принимать дополнительный параметр – указатель на редактируемый объект. Вторым параметром так и останется структура данных для заполнения объекта. И точно так же, если данные будут нулевыми, мы будем запрашивать их у пользователя с консоли. Возвращать ли ссылку на изменённый объект из функции – решать вам. Ведь редактирование мы будем делать в параметрах функции. Но с точки зрения бизнес – логики будет верно предусмотреть возврат изменённого объекта. Ниже будет приведён примерный код такой функции:

```
// функция для редактирования объекта
// модели "пользователь"
// принимет указатель на редактируемый объект
// и указатель на данные заполнитель (если таки есть)
// возвращает указатель на тот объект, который мы редактировали
// может ничего не возвращать
User* editUser(User* user, UserData* userData = nullptr) {

    // переменная - триггер, которая нужна для проверки
    // удалять ли нам объект, т к мы можем удалить его
    // только в том случае, если создали сами
    bool deleteObj = false;


    if (!userData) {
            deleteObj = true;

            // для сокращение дублирования кода
            // если в указателе не будет данных
            // то мы сами создадим эти данные
            userData = new UserData{};

            cout << "Введите новые данные для пользователя:" << std::endl;

            // вызов функции проверки пользовательского ввода
            // которая возвращает корректную строку
            // функцию можно реализовать любым способом
            userData->name = getStringEnter();
```

```
            userData->surname = getStringEnter();
            userData->age = getIntEnter();
        }

        user->setName(userData->name);
        user->setSurname(userData->surname);
        user->setAge(userData->age);

        if (deleteObj) {
            delete userData;
        }

        return user;
    }
```

- Функция удаления объекта. Вместе с первыми двумя функциями вам стоит реализовать функцию удаления объекта. Как правило, данная функция не будет содержать какой – то специальной логики. Лишь будет принимать указатель на объект, который нужно удалить, и возвращать логическое значение, которое будет сигналом об успешном/неуспешном проведении удаления. Иногда в такую функцию добавляется определённая логика журналирования удаления, проверки возможности удаления объекта и, самое банальное, логирование удаления. Но в нашем ***общем*** случае функция будет просто удалять сущность. Пример:

```
// функция удаления объекта
// модели - "пользователь"
bool removeUser(User* user) {
        delete user;

        return true;
}
```

Следующим шагом будет создание ряда функций по управлению «наборами» данных (объектов моделей). Наборами здесь и далее мы будем именовать массивы (динамические массивы), векторы, списки и т.д., которые хранят объекты той или иной сущности. Какие функции нужны здесь?

- Функция сортировки по ключу. Данная функция сортирует массив объектов по полю, которое передано в параметрах. На вход, кроме указания поля сортировки, передаётся: сам набор данных, которые мы должны сортировать. Функция должна возвращать логическое значение, которое будет сигналом об успешной или неуспешной операции.
- Функция фильтрации по значению. Данная функция должна принимать набор объектов, функцию фильтрации и поле, к которому эта функция будет применена. Возвращать она должна новый набор объектов, в который будут добавлены только те данные, которые прошли фильтрацию.

- Функция поиска элемента по ключу. Это очень простая функция, которая принимает поле, по которому будем искать, значение, которое будем искать и набор данных, в котором будем искать. На выходе функция возвращает указатель на найденный элемент или нулевой указатель, если объекты не найдены.
- Функция добавления нового элемента. Эта функция будет принимать набор данных и вызывать функцию создания нового элемента, который будет вставлен в этот набор данных. Возвращать она будет указатель на добавленный элемент.
- Функция удаления элемента по ключу. Данная функция будет принимать набор объектов и ключ, по которому нужно будет найти удаляемый элемент и удалить его. Возвращать функция будет логическое значение, которое будет говорить об успешности/не успешности операции.

Давайте заметим, что большинство функций здесь будет иметь общий функционал по работе с разными наборами объектов. Вы, конечно, можете пойти несколькими путями. Первый вариант – это написать отдельные функции для каждой сущности и в названии их определить, к какой сущности эта функция относится. Этот подход очень прост в реализации, но его минус – огромное количество дублированного кода и функции, которые делают одно и то же. Второй вариант – это использовать перегрузки функций. Этот вариант предполагает создание функций, которые будут различаться по передаваемым в них аргументам. Таким образом, мы создадим общие интерфейсы обработки наборов данных, но при этом не избавимся от количества функций, а также от дублирования логики. Третий вариант – это использования парадигмы обобщённого программирования. Согласно ей, мы будем писать максимально абстрактную логику, которая будет применяться почти ко всем (к большинству) наборам объектов. В этой реализации будет задействован механизм языка C++ – шаблоны. Про шаблоны вы можете прочитать на сайте ravesli.com. Четвёртый вариант предполагает использование одного из принципов ООП: полиморфизм. То есть обобщённый тип для переносимости более узконаправленного. Этот подход будет самым эффективным по части сокращения дублирования кода, но будет требовать разработки сложных алгоритмов идентификации типов данных, с которыми мы будем работать. В данном руководстве будет использоваться гибридный подход, который будет совмещать в себе все изложенные варианты. Конкретные реализации функций мы рассмотрим при разборе варианта.

Давайте рассмотрим, что ещё есть общего у всех вариантов. Все автоматизированные системы (комплексы) предполагают непосредственное взаимодействие с пользователем. Как вы построите это взаимодействие зависит от вас, но стоит отметить, что логика самой программы и логика взаимодействия с пользователем должны быть реализованы отдельно. И тут нам необходимо рассмотреть модульный подход к реализации поставленной задачи.

Модульный подход

Что же предполагает модульный подход? Он предполагает разделение логики продукта на независимые части – модули (на языке C++ они именуются как библиотеки). Каждая часть не должна зависеть от своей коллеги, но при этом они должны уметь взаимодействовать между собой. Какие преимущества даёт модульный подход? Самое главное преимущество — это повторное использование кода. Написав один модуль для решения конкретной задачи, вы сможете повторно использовать его много раз в своих программах. Другой плюс модульного подхода – это переносимость программы. Мы с

лёгкостью можем вытащить любой модуль из нашей программы и перенести его в другую, которая знает, как его использовать. А использование модуля завязано на знании интерфейса взаимодействия с ним. Таким образом, мы получаем замечательный способ внедрения частей нашей программы в другие программы. И ещё один немаловажный плюс заключается в том, что мы сможем легко модернизировать свою программу. И здесь мы выигрываем сразу в двух случаях:

- Если мы хотим добавить функциональность или переписать часть какого – то модуля, это не повлияет на существующие части, которые используют этот модуль. Это происходит потому, что для них важен только интерфейс взаимодействия.
- Если мы хотим расширить функционал нашей программы, нам придётся только написать новый модуль и подключить его внутри. Нам не будет нужно редактировать функционал существующих модулей.

Давайте рассмотрим пример. Скажем, есть у нас программа, которая использует модуль отрисовки в консоли. В один момент мы захотим сохранять всю информацию не в консоли, а в файле, либо выводить в браузере. Если бы мы писали программу монолитно (без модулей, жёстко привязываясь к консоли), то нам бы пришлось переписывать существенную часть функционала. Но, используя модули, нам нужно будет лишь заменить модуль консоли на модуль файла или любой другой среды вывода и отрисовки, сохранив при этом общий интерфейс. Для основной программы ничего не поменяется, но данные мы уже будем получать в том месте, где задумали.

Таким образом, система модулей очень сильно упрощает нам разработку и дальнейшую поддержку программного продукта.

Рассматривая задания вариантов всех Л/Р, вы увидите, что все они будут разделены на несколько модулей, причём часть модулей может быть спокойно перенесена между вариантами. Давайте рассмотрим эти модули.

Начнём с того, что все программы должны взаимодействовать с пользователем, как и было сказано ранее. Из этого следует, что необходимо придумать этот «экран» взаимодействия. Где будет помещаться информация и откуда она будет вводиться? Для простоты мы будем рассматривать консоль. Вы же в своих Л/Р можете использовать иные средства вывода/ввода информации (Qt, SFML, WinApi и пр.) Как было сказано ранее, чтобы не привязываться к функциям вывода и ввода, мы спрячем их в отдельный модуль. В рамках данного руководства мы будем рассматривать модуль отрисовки в консоли. Давайте назовём его **ConsoleManager**. Этот модуль будет отвечать за считывания и отправку данных в консоль.

Следующим модулем станет модуль **Menu**. Это универсальный модуль, который будет позволять пользователю делать выбор из каких – то представленных возможностей (например, главное меню или список книг в библиотеке).

Также по заданию необходимо предусмотреть многопользовательский режим работы программы. Из этого следует, что нам нужен модуль, который будет отвечать за авторизацию пользователей. Назовём его **Auth**. И последний модуль, который мы должны будем реализовать – это модуль самой программы, её основной логики. Назовём его **App**.

Ну и не стоит забывать про общие функции обработки наборов данных, а также про стандартные системные функции: считывание данных из консоли и валидация

пользовательского ввода. Это всё можно вынести в отдельный модуль и назвать **Tools или Utils**.

Теперь подведём итоги. Модули, которые должны получиться у вас в каждой Л/Р будут следующими:

- ConsoleManager – отрисовка и получение данных в консоли.
- Menu – модуль отображения каких – то пунктов с возможностью выбора конкретного.
- Auth – модуль, позволяющий реализовать многопользовательский режим.
- App – модуль, отвечающий за основную смысловую часть программы.
- Utils (Tools) – модуль, отвечающий за общие алгоритмы обработки данных.

Как видно из списка модулей, отличаться программы будут только по одному из них: по модулю App. Причём только в этом модуле будет находиться функционал, привязанный к конкретному варианту задания. Все остальные модули могут быть спокойно перенесены из варианта в вариант. Таким образом, реализовывая эти модули для варианта, разбираемого в данном руководстве, вы реализуете их и для своего варианта. Теперь, разобрав все необходимые нюансы реализации, мы приступим к непосредственному решению варианта.

Разбор решения 18 варианта

Разбор 18 – ого варианта начнём с чтения задания. Оно звучит так:

Вам будет предложено написать программу – «Автоматизированная система космодрома». Которая будет включать следующий функционал:

- Ведение базы сотрудников и астронавтов
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы космических судов и запланированных запусков (реализовать возможность запуска – виртуально!)
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

Ниже этого текста идут конкретные задачи, которые вам необходимо выполнить. Но давайте вспомним, что в задачах не дан конкретный алгоритм решения, там представлены лишь путевые точки, чтобы вы смогли начать реализацию своего продукта, двигаясь в верном направлении. В этом руководстве мы не будем обращать внимание на эти задания, а спроектируем всё сами, принимая во внимание лишь главную задачу.

Итак, нам нужно написать автоматизированную систему космодрома. Из задания видно, какими функциями она должна быть обеспечена, а именно:

- Авторизация – возможность входить под разными учётными записями. Это будет отдельный экран. Сразу обратим внимание, что входить может два типа пользователей: сотрудники и астронавты. Возьмём за администраторов системы сотрудников.
- Ведение базы космических судов и запланированных запусков. Это означает, что у нас должны быть отдельные экраны, отвечающие за просмотр и управление

запусками и доступными космическими кораблями. Также не стоит забывать, что ранее мы обозначили возможность авторизации пользователей. Этими учётными записями также нужно управлять на отдельном экране. Какие функции управления нужно реализовать? В задании сказано: добавление, редактирование, удаление, сортировка и фильтрация. Итог: нам нужно реализовать три экрана, на которых будут отображаться следующие наборы данных: космические корабли, пользователи и запланированные запуски. Для каждого экрана будут определены функции: добавить/редактировать/удалить/сортировать/фильтровать. Для последних, в качестве упрощения, так как в задании не сказано, по каким полям будем фильтровать и сортировать, выберем по одному главному полю.

- Возможность запуска. Это означает, что у нас будет отдельный экран для отображения возможности запуска. Что это будет? Так как в задании об этом ничего не сказано, сделаем на своё усмотрение. Предположим, что, выбирая нужный запланированный запуск, нас будет отправлять на экран, на котором будут кнопки «Старт» и «Отмена». При нажатии на последнюю мы будем возвращаться к экрану запланированных запусков, а при нажатии на кнопку «Старт» – будем запускать обратный отсчёт. После чего на экране отобразится сообщение об успешном запуске.
- В последнем пункте нас просят формировать некие отчёты и сохранять состояние нашего приложения. Нам опять не сказали, что должны представлять из себя отчёты, поэтому условимся, что отчётами будут файлы, в которых будут храниться запланированные запуски. Что касается сохранения состояния приложений, то здесь условимся, что будем сохранять списки кораблей, пользователей и запусков в разные файлы типа .txt. Затем будем загружать из них информацию. Эти же файлы станут для нас отчётами.

Итак, после того как мы разобрали основные моменты задания, начнём строить архитектуру нашего будущего продукта. Как она будет выглядеть? Чтобы понять это, нужно представить, как будет выглядеть взаимодействие пользователя с нашей программой. Давайте попробуем:

При первом запуске пользователь видит экран авторизации, так как мы должны понять, кто он: администратор или астронавт. За администраторов мы условились брать сотрудников космодрома. Если это первый запуск программы, то очевидно, что никаких данных в программе нет. Что же делать? Решим эту проблему учётной записью администратора по умолчанию, которая будет уже храниться в программе и иметь фиктивные данные по типу: login – admin, password – admin. И при первом входе пользователь может войти по этим данным.

Дальше ситуация расходится. Рассмотрим, что будет, если пользователь зашёл под учётной записью администратора:

Он видит перед собой меню, в котором может сделать следующие вещи:

- Перейти к списку кораблей
- Перейти к списку запусков
- Перейти к списку пользователей
- Выход из учётной записи

Последний пункт выходит из учётной записи и возвращает нас на экран авторизации.

Теперь рассмотрим, чем будет отличаться панель меню для астронавта. Скорее всего, ничем. На ваше усмотрение вы можете убрать доступ к пункту перехода к просмотру учётных записей, давайте это и сделаем. Оставшиеся три пункта будут общими для двух типов пользователей. Теперь рассмотрим каждый пункт:

Перейти к списку кораблей.

В этом пункте нам будет отображаться список всех доступных кораблей, а также возможность ввода, чтобы выбрать номер нужного корабля. После ввода номера нам будет предложено сделать несколько действий, таких как: редактировать корабль или удалить его. Сразу здесь скажем, что опции «добавить новый» или «выбрать существующий для редактирования» астронавту будут недоступны. Также нужно подумать, как вернуться на предыдущий экран. Это можно сделать, добавив соответствующий пункт меню внизу или вверху списка. Давайте для простоты сделаем первые два пункта меню одинаковыми для всех экранов, а именно: «добавить новый» и «назад». Первая опция для астронавта будет недоступна.

Что будет отображаться для пункта «добавить»? Нам откроется новый экран, где нужно будет ввести название корабля, и после сохранения он отобразится в общем списке.

Если нажать на редактирование, то нам откроется такое же меню для ввода нового имени корабля.

Другие экраны

Для экранов «список запусков» и «список пользователей» функциональность будет аналогичная. Отдельно для запусков мы добавим дополнительный пункт, с помощью которого можно перейти на экран запуска, а после успешного запуска, мы вычеркнем его из списка.

Общие итоги.

Итак, что касается общих итогов? В голом остатке имеем следующий функционал:

- Авторизация
  - Администраторы
  - Астронавты
- Экран запусков
  - Добавить
  - Редактировать
  - Удалить
  - Сортировать
  - Фильтровать
  - Экран самого запуска
- Экран пользователей
  - Добавить
  - Редактировать
  - Удалить
  - Фильтровать
  - Сортировать
- Экран кораблей
  - Добавить
  - Редактировать

- Удалить
- Сортировать

Теперь, когда немного ясен план и куда нам двигаться, давайте начнём реализацию.

Весь код описанный в данном руководстве будет расположен по git репозитории по адресу:
https://github.com/DmitriyODS/laboratory-work-on-VP
Шаблон кода – библиотека, была построена по образцу 18 – ого варианта Л/Р

Базовая архитектура программы

Правильно составленная архитектура программы – это залог успешной разработки в дальнейшим. Она позволит без труда добавлять новые функции в готовый продукт, изменять старые, или вовсе удалять какие – то части. И самое главное, писать программу с продуманной архитектурой гораздо легче. Дальнейшие расширение продукта – это то, к чему необходимо стремиться.

В рамках лабораторных работ вам необходимо написать полноценный бизнес продукт, который будет являться логически – законченной программой. Для того, чтобы это сделать, нужно разработать универсальную архитектуру, которая позволит писать главные функции программы наиболее оптимально. Для данных работ будет предложено несколько решений, которые станут универсальными и подойдут к любому из вариантов. Эти подходы будут связаны – многоуровневым программированием. Но, для начала, следует разобраться в нескольких паттернах проектирования.

Паттерн

Паттерн проектирования — это часто встречающееся решение определённой проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.

Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.

Если привести аналогии, то алгоритм — это кулинарный рецепт с чёткими шагами, а паттерн — инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации.

Класс – синглтон

В проектировании программ существует такое понятие, как паттерн – синглтон (Singleton pattern). Синглтон – это класс, объект которого можно создать лишь один раз, и он будет единственным в течение всего выполнения программы. Для чего это нужно? Такой подход позволяет получить централизованный доступ к объекту из любой части программы, также он гарантирует, что состояние данного объекта (его данные), будет всегда актуальным при любом вызове. Такой паттерн очень упрощает разработку, делает программу гибкой и может использовать, например, для синхронизации частей программы, или даже целого программного комплекса.

Для реализации паттерна – синглтон необходимо написать класс, который будет закрывать свои методы: конструкторов и копирования. Единственное создание объекта данного класса, будет происходит в одном из его методов – статическом методе, а объект такого класса будет сохраняться в статической переменной, что позволит существовать ему

на протяжении всей работы программы. Статические поля и методы – это такие переменные, которые не привязаны к классу, но при этом входят в его пространство имён. Классы, которые состоят из статических данных, называются – статическими классами. Класс синглтон – как правило статический класс. Для получения его объекта мы вызываем статический метод, который возвращает нам уже созданный, или созданный на ходу – объект.

Состояние приложения - State

Любая программа имеет интерфейс взаимодействия с пользователем и данные, с которыми она работает. Часто, некоторые данные влияют на её функциональность с точки зрения поведения, или интерфейса. Например, в зависимости от пользователя, который сделал вход в вашу программу, вы можете предоставлять, или не предоставлять определённые функции в меню. Или, в зависимости от выбранного главного цвета в настройках, ваше приложение будет, либо тёмным, либо светлым. Данные, которые влияют на поведение программы называются – состоянием. Состояние программы – это одно из основополагающих понятий разработки. При правильной архитектуре состояние приложения отделяется от самой логики и реализации. Благодаря чему, мы можем менять поведение приложения подставляя нужное состояние в виде данных. Кроме того, состояние (State), также может включать в себя данные, с которыми работает программы, это позволит отвязать от логики не только данные состояния, но ещё и данные обработки. Подобная отвязка обеспечит гибкий подход к хранению информации вашего приложения и получения доступа к ней из любой точки программы. При этом мы не зацикливаемся на типе получаемых данных, мы обрабатываем их потоково согласно предоставленным алгоритмам. Этот подход используется в многоуровневом программировании.

Многоуровневое программирование

Многоуровневое программирование – это подход к разработке архитектуры программы (паттерн), который предполагает деление программы на логические, обособленные уровни абстракции. Такой подход обеспечивает гибкую разработку, позволяющую изменять отдельные части программы, не влияя при этом на другие. Для данного подхода очень уместно использование ООП, которое благодаря полиморфизму помогает реализовать интерфейсы взаимодействия уровней.

Каждый уровень – это обособленная логическая единица, которая через определённые рычаги взаимодействует с другой такой же обособленной единицей. Давайте рассмотрим пример: пусть у нас есть программа – планировщик задач. Давайте разобьём её на уровни абстракции. Мы пойдём снизу – вверх, и первым уровнем будет - уровень данных. Этот уровень обеспечивает работу с данными: добавление, хранение, обработку и получение. Он реализует общий интерфейс, через два метода: добавить данные и получить данные. Теперь, вызывая эти методы в любой части программы, вы будете уверены, что получите, или отправите свои данные. Вас не будет беспокоить, как уровень реализует работу с вашей информацией, оно вам и не будет нужно.

Вторым уровнем пойдёт уровень вашей логики программы. На этом уровне вы пишите основные функции программы: создание, редактирование, удаление задачи, сортировка, фильтрация, или поиск нужной и т д. Этот уровень, также является обособленным, потому что работает с некоторым объектом данных, который возвращает нам уровень – данных. Объект данные – реализует определённый интерфейс – контракт, который гарантирует наличие методов по работе с данными. Таким образом, уровню логики, совершенно не важно какой именно тип будет иметь объект данных, с которым он работает, ему важно лишь то, чтобы он реализовывал методы, которые придётся вызывать.

Третий уровень – уровень пользовательского интерфейса. Здесь ещё всё проще. Компоненты – которые отрисовываются на экране, предоставляют функции взаимодействия с программой пользователю – вообще не знают о том с какими данными, а уже тем более с какой логикой они работают. Этот подход называется – подходом с глупыми компонентами. Почему глупыми? – потому что компоненту не важно знать то, что он делает и над чем он это делает. Он не привязан к данным. Каждый компонент может выполнять лишь одну функцию, которая является его главным поведением. Например – кнопка. Её главное поведение – это запуск какого – то действия. Этой кнопке не нужно знать, какое действие она запускает. Ей просто нужна функция со знакомым интерфейсом, который она сможет использовать для запуска. Кнопка не должна передавать интерфейсу никаких данных, кроме тех, которые являются необходимыми для её существования (например – текст кнопки). При этом стоит обратить внимание, что такие данные, как – текст кнопки, её активность/не активность, функция обратного вызова и т д, являются паттерном State – состояние кнопки. А все состояния хранятся в одном месте и при изменении их, мы меняем поведение кнопки. Здесь чётко прослеживается гибкость централизованного управления данными.

Отдельно стоит остановиться на паттерне – callback, или функция обратного вызова. По сути, это такая функция, которая вызывается не сразу, и не из того места, в котором упомянул её программист. Такие функции передаются в качестве параметров другим функциям и те в свою очередь, в зависимости от определённых событий вызывают их. Такой подход обеспечивает гибкость взаимодействия множества функций и используется

повсеместно в проектировании и разработке логики пользовательских интерфейсов. Например: кнопка, список, какое – то события и т д.

Возвращаясь к уровням абстракции, у нас получилось три уровня:

- Уровень пользовательского интерфейса
- Уровень логики
- Уровень данных

Все эти уровни не связаны между собой и являются логически законченными и самостоятельными единицами программы, но предоставляя общий интерфейс, они могут передавать друг – другу данные. Т. к. каждый уровень не зависим, то может быть модифицирован, или изменён, не нарушая при этом логики интерфейса.

Паттерн - Immutable Object

Давайте рассмотрим ещё один паттерн, который называется – иммутабельность, или неизменяемость. Речь идёт об объекте, состояние которого не может быть изменено. Такой объект можно сравнить с константой, но он таковой не является. Термин – константа, чаще всего применятся к самой переменной, которая хранит объект. Именно значение переменной нельзя переписать. А immutable объект, это сам объект, который нельзя изменить. Переменная может хранить любые объекты, мы можем спокойно присваивать ей другие сущности и т д, но эти сущности нельзя будет никак модифицировать. Чем полезны такие объекты? – плюсов у таких объектов много, но самый главный из них в том, что такие объекты можно использовать для хранения данных. Данные – это ценный ресурс, которые нужно защищать от случайного изменения, или повреждения. Именно неизменяемые объекты спасают ситуацию.

Как же изменить данные, если объект не изменяемый? – ответ просто – никак. Для таких целей создаётся новый объект, который включает старые данные и модифицированные. Таким объектом является, например, объект State, который мы рассмотрели ранее. Его нельзя изменить напрямую, но можно через специальные методы его заменить на другой такой же объект с другими данными. Как это сделать – узнаете ниже.

Общее хранилище состояний приложения – Store – Sate

Теперь настало время поговорить о таком сложно на первый взгляд, от того, что он очень сильно расширяем, паттерне – Store. Как было сказано ранее, любая программа для дальнейшей, удобной расширяемости по функционалу делится на уровни абстракции. Эти уровни предоставляют общий интерфейс взаимодействия для других уровней. Мы сейчас подробнее остановимся на уровне – данных. Мы сказали, что данные нашей программы делаться на – данные состояния приложения и на обрабатываемые данные. Причём обрабатываемые данные могут приходит к нам от куда угодно и когда угодно, а вот данные состояния программы – существуют всегда и лишь изменяются в самой программе. Два этих вида данных должны быть соединены в каком – то общим хранилище, через которое можно будет ими управлять и получать непосредственно доступ. При этом, само хранилище должно быть абстрагировано от типа этих данных и предоставлять схожие действия для

любого из них. Для реализации тако логики на помощь приходит такой паттерн проектирования, как – Store, или хранилище. Этот паттерн состоит из главных трёх частей:

- Store – непосредственно сам объект, который предоставлять доступ к API (основные методы по работе с объектом) по работе с данными.
- State – данные нашей программы.
- Reducer – функция модификации наших данных.

На самом деле паттерн Store включает ещё несколько функция, которые были позаимствованы их паттерна – Observer (наблюдатель), но мы их не будем рассматривать в качестве упрощения и не надобности их в нашей разработке.

Теперь, давайте подробно остановимся на каждой из частей Store.

Итак, начнём с того, что мы уже знаем – State, это объект, который просто содержит данные всей программы. Это данные могут быть особым образом структурированы и т д, но смысл от этого не меняется – это просто объект с данными, он ничего не знает о тех данных, которые у него есть – он не умеет их обрабатывать и т д, он просто их хранит. вследствие чего, в некоторых языках программирования он может быть реализован через структуру.

Reducer – это функция, которая выполняет модификации State. Именно она изменяет данные в нашем хранилище, как она узнаёт, что менять? На самом деле всё просто. На вход данной функции приходит объект – Action, этот объект имеет всего два поля: тип события и данные. Тип события, как раз и описывает, что нужно сделать с данными функции Reducer, а данные – это новые данные для State. Как было сказано выше, State – является immutable (неизменяемым) объектом, его нельзя модифицировать напрямую. Поэтому Reducer работает следующим образом:

На вход поступает Action с типом события и новыми данными для State. Редусер определяет по типу события, что нужно сделать (какие данные нужно модифицировать), берёт текущий State, копирует его в новый объект при этом модифицируя нужное поле данными, которые пришли к нему в Action и кладёт его на место. Таким образом состояние нашего приложения меняется. Логика редусеры прозрачна и проста, поэтому более останавливаться мы здесь не будем. Главное понять, что все взаимодействия с данными идут через Reducer, никакой другой объект, функция и т д, не может модифицировать эти данные.

Store – это связующее звено, которое соединяет State и Reducer вместе. Стор инициализирует начальными данными State (через Reducer), управляет их сохранением в базе и выгрузкой. Store – является классом сингтоном, который доступен из любой части программы, что, собственно, логично. Чтобы получить данные у него есть метод: getState, он возвращает объект хранилища (как правило – константу, либо копию этого хранилища) в любом случае, модификация того, что он вернул никак не отразиться на реальных данных. Также, у него есть метод: dispatch, которые вызывает нужный Reducer для модификации данных. На вход диспатч принимает объект события (Action), который потом пробрасывается в reducer.

На этом, собственно, и всё. Если хорошо разобраться, логика паттерна Store – достаточно проста и логична, такой подход обеспечивает удобны, а главное централизированный доступ к данным, гарантируя при этом их актуальность при любом вызове.

Поведение программы – единица поведения

Теперь, пришло время рассмотреть уровень логики программы. Как правильно его спроектировать, чтобы он мог быть расширен и не зависел от вида программы? Для этого мы будем использовать тако подход, как – единица поведения. Что это такое? Любая сложная программа может быть разбита на множество подпрограмм, вы это уже знаете. Но, вам предлагается не разбить вашу программу на функции по логике действий, а разбить её на смысловые блоки. Давайте рассмотрим это на примере: допустим у нас есть всё таже программа – список задач. Какие функции она должна предоставлять пользователю?

- Авторизация
- Управление задачами
- Управление пользователями

Мы видим, что у нас прослеживаются три независимых блока функций. Сразу, мы можем представить их в виде окон – мини программ, которые обособлены. Например, окно авторизации ничего не должно знать об окне списка задач, а список задач ничего не должен знать об окне список пользователей. Каждое такое окно имеет своё уникальное поведение, такие окна мы будем называть – единицами поведения программы. Таких единиц может быть сколько угодно, что позволит нам не концентрироваться на конкретной программе, а просто добавлять нужные окна по мере необходимости. Такой подход обеспечивает расширяемость вашего программного продукта. Давайте в дальнейшим к таким единицам применять термин – экран, это больше подходит в контексте консольного приложения, т к окно у нас одно – консоль, а вот отрисовывать мы в нём будем разные экраны для разного поведения. Важно понять, что все экраны должны быть не зависимы друг от друга, с первого взгляда кажется, что экран авторизации должен взаимодействовать с экраном список пользователей, ведь и тот и тот работает с данными о пользователях, но это не так. Да, они оба получают доступ к пользователям, но выполняют логически разные действия. Поэтому необходимо разнести их в разные единицы поведения.

Что долен включать в себя экран? – ну во – первых, это может быть класс – синглтон, т к чаще всего, экраны программы задаются на этапе проектирования и больше не модифицируется во время выполнения. Во – вторых, он должен включать в себя два главных метода: start и render. Первый будет являться главной точкой запуска экрана, которой – будет являться главной точкой отрисовки этого экрана на экране. Разделение логики и отрисовки, добавит гибкости нашей программе, и мы не должны будем заботиться о том, как наши данные будет видеть пользователь.

Т К все наши экраны должны будут содержать два определённых метода, то они все будут логически походи между собой, все они будут обеспечивать единый интерфейс взаимодействия. Что это значит? Это значит, что самое время применить полифонизм и создать интерфейс, который будут реализовывать эти экраны. Через этот интерфейс мы сможет одинаково управлять экранами, что обеспечит централизованный, а самое главное абстрактный подход к запуску единиц поведения.

Интерфейс – класс

Что такое класс – интерфейс? Это паттерн, который создаёт определённый контракт, а точнее описывает, какие методы должен содержать тот класс, который хочет реализовать этот интерфейс. Благодаря такому подходу, мы можем обрабатывать данные не зная их типа, опираясь лишь на то, что они реализуют конкретный, нужный нам интерфейс, а это значит, что у них точно есть реализация всех тех методов, которые прописаны в интерфейсе и мы спокойно можем их вызывать. По сути, это всё. Интерфейс помогает нам построить объекты таким образом, чтобы они относились к определённой категории объектов, реализовывали определённые методы и позволяли получить к ним доступ.

Управление экранами – App

Итак, мы решили использовать при разработке подход, который основывается на единицах поведения – экранах. Теперь самое время подумать о том, как эти экраны будут работать, а точнее, кто этими экранами будет управлять? – зачем ими управлять, спросите вы. Ответ будет очень простым и логичным – т к экраны обособленные единицы, по сути это подпрограммы в вашей основной программе, они не могут напрямую вызывать друг – друга, это нарушило бы принцип абстракции. С другой стороны, раз эти объекты обособлены, то они могут быть, а могут и не быть в нашей программе. Получается, если один экран будет жестко связан с другим, то программа будет аварийно завершена в случае, если экран захочет вызвать связанный с ним экран, а того экрана не окажется. И такая ситуация имеет место быть в модульном программировании. Например, если мы экран авторизации свяжем с главным экраном программы. Этот экран, после каждой успешной авторизации будет открывать главный экран, с которым связан, но если мы захотим поменять главный экран, или перенести модуль в другую программу? В таком случае вам придётся переписывать логику открытия, что не очень хорошо.

Другая проблема, с которой можно столкнуться это рекурсивные вызовы. Например, у нас есть главный экран, который запускает другой экран, но в запущенном экране, нам снова нужно открыть главный экран при этом не закрывая текущий, что делать? Мы можем снова вызывать главный экран, но тогда в стеке программы образуется ненужная стопка функций, которая будет лишь расти по мере работы программы. Это лишь малая часть самых важных проблем, которые у нас появятся при жёстком связывании экранов между собой. Поэтому, следует выбрать другой подход в проектировании. Таким подходом будет – создание объекта приложения – App.

Что такое App и за что он будет отвечать? App – это по сути контейнер, который будет включать в себя все части нашей программы, а также будет ими управлять. Давайте рассмотрим это на примере уже со знакомой нам программой: организатор задач.

Как мы ранее говорили у нас есть уровни абстракции, такие как:

- Пользовательский интерфейс
- Логика программы
- Данные

Все эти части и будет соединять контейнер App. Структура нашей программы может быть такой:

- Store – хранилище данных
- Screens – экраны нашей программы
- App – главный контейнер

Как это будет работать? В объекте апп, который будет являться экземпляром класса – синглтон, ведь этот объект будет един для нашей программы и будет существовать в памяти до её завершения, будет два системных поля. Первое системное поле будет хранить указатель на текущий экран с которым мы работаем. По сути, у нас будет переменная, которая будет хранить указатель на экран. Эту переменную мы будем использовать для запуска экрана через метод start. Запущенный экран, при этом, получает всё управление программой. После отработки основной логики экран может сделать следующие вещи: просто закрыться. Тогда, управляющий контейнер app, удалить указатель на экран из переменной, и программа закроется. Ещё, экран может сказать апп, что хочет после себя открыть другой экран. Эту информацию он положит в state, после чего закроется. Контейнер app, после получения управления проанализирует state на предмет указателей на другие экраны и если такой найдёт, то поместит его в переменную, которая хранит текущий экран и запустит её. Так, мы уже реализовали возможность запуска другого экрана. Но, что если мы хотим, чтобы после запуска нового экрана мы вернулись к старому экрану? – можно сделать так, чтобы вновь запущенный экран записывал в хранилище указатель на открываемый экран, как это было уже сделано, но здесь встаёт другая проблема, мы не знаем к какому экрану нужно обращаться. Иными словами, мы не знаем, какой экран скомандовал нам запуститься, поэтому вернутся к тому экрану таким способом мы не сможем. Что делать? Добавить дополнительное поле к нашему app, которое будет являться стеком запущенных экранов. А управлять добавлением и удалением из этого стека мы будем через state. Таким образом, если экран хочет запустить другой экран, но при этом, после закрытия нового экрана снова отрисоваться, ему нужно сделать две вещи: указать запускаемый экран в state, а также выставить флаг о том, что его нужно добавить в стек. После закрытия app проанализирует state и, если увидит, что текущий экран хочет записаться в стек, он добавить указатель на него в стек, а в переменную текущего экрана добавит указатель на запускаемый экран. После закрытия запускаемого экрана app просмотрит стек на предмет сохранённых экранов и если такие будут вытащит из него указатель и поместит его в текущий экран.

Идея стека открывает нам доступ к абсолютной гибкости запускаемых экранов. Мы можем добавлять в стек множество экранов и создавать большую логическую вложенность элементов. После закрытия последнего программа сама будет закрыта. Здесь появляется наводящий вопрос: мы ранее озвучили, что экраны не должны жёстко быть связаны между собой, но как же тогда экраны будут говорить, какой экран они хотят запустить, это разве не одно и тоже? – нет, это совершенно разные вещи. В первом случае экран именно вызывает метод старта у другого экрана, и если такового не будет, программа упадёт. В нашем случае происходит не запуск экрана, а некоторый Intent (намерение). Экран записывает в state, что он хочет открыть такой – то экран после себя и передаёт его Id (даже не указатель на реальный объект), а уже app просматривая State принимает решение об открытии того, или иного экрана и если такой есть, он отроет его, а если нет, просто не откроет – ошибки не будет и вылета программы тоже.

Теперь остался последний, но не менее важный вопрос. Что если один экран хочет передать данные другому экрану? – поступим с этим точно также – используем state. Просто запишем

в state данные, которые хотим послать другому экрану, а другой экран прочитает их и всё. Как видите с нашей архитектурой решение прикладных задач превращается в простые реализации.

Мы рассмотрели все необходимые паттерны для начала реализации задачи из лабораторных работ. Теперь пришло время применить полученные знания на практике, а именно приступить к решению конкретной задачи.

Решение 18 – ого варианта ВП

Давайте вспомним, что мы сейчас рассмотрели паттерны для универсальной разработки программы, таким образом нам осталось только реализовать их на конкретном языке программирования. Связав их вместе, мы получим общую архитектуру программы для всех вариантов, которую мы просто будем заполнять логикой в зависимости от наших потребностей. По сути, мы разработаем свою библиотеку – фреймворк, которая будет выполнять роль некого макета, который мы будем заполнять нужными данными. Давайте начнём.

Начнём мы с постановки технического задания:

Формулировка технического задания

Вам будет предложено написать программу – «Автоматизированная система космодрома». Которая будет включать следующий функционал:

- Ведение базы сотрудников и астронавтов
  - Создание / удаление / редактирование записей
  - Сортировка / фильтрация
- Ведение базы космических судов и запланированных запусков (реализовать возможность запуска – виртуально!)
- Возможность авторизации
- Создание файлов – отчётов и сохранения состояния

Данное техническое задания является обобщённым, это даёт нам некоторые плюсы в реализации программы. Плюсы заключаются в том, что мы сможем отдельные её функции реализовать, как удобно нам. Давайте обозначим сущности и их поля, которыми будет управлять наша программа:

Сущности

- Пользователь
  - Id
  - Имя
  - Фамилия
  - Логин
  - Пароль
  - Уровень доступа
- Корабль
  - Id
  - Название
- Задача

    - Id
    - Id пользователя
    - Id корабля
    - Название
    - Дата начала
    - Is completed // поле отвечает за проверку – завершена ли задача

Теперь следует заметить, все эти сущности могут быть реализованы через структуры, а не классы. В них не содержится какой – то сложной логики, а также валидаторов полей. Также, мы можем спокойно опустить конструкторы и деструкторы этих сущностей, т к их поля примитивны. В других вариантах лабораторной работы вам, возможно, придётся столкнуться со сложными структурами данных, которые будут включать наследование, проверку валидности полей и сложные конструкторы, поэтому скорее всего вам придётся использовать классы, но смысла это не меняет. Также, особо обратите своё внимание на использование наследования в сущностях, нужно ли оно вам? Возможно, его можно и нужно заменить композицией, но это уже ваше право.

Теперь, касательно id. Id – это уникальный идентификатор объекта, который должен быть УНИКАЛЬНЫМ для каждого объекта (объектов текущей сущности (одного типа (структуры, или класса))). Как это возможно реализовать? Самый оптимальный путь – использовать статические поля. Про статические поля класс вы можете прочитать на ravesli.com. Итак, у нас в каждой сущности будет статическое поле класса, которое будет хранить новый, свободный id для нового объекта. Создавая новый объект той, или иной сущности, его полю id мы даём значение из статического поля текущего id, а поле текущего id увеличиваем на единицу (значение этого поля). Таким образом у нас будут реализованы универсальные генераторы id в каждой сущности, что удобно.

Роли

Теперь, касательно поля – уровня доступа. Это поле будет хранить тип учётной записи пользователя. В нашей программе существует два типа учётных данных:

- Космонавт
- Сотрудник – он же администратор

Эти типы учётных данных мы будем называть – ролями. Обозначим действия, которые будут доступны для каждой роли.

Роль – сотрудник – администратор

Роль сотрудника, по сути, является ролью администраторы в нашей программе. Ей будут доступны все действия, которые возможны:

- Доступ к пользователям
    - Добавление
    - Редактирование
    - Удаление
    - Сортировка по имени
    - Фильтрация по ролям
- Доступ к задачам запуска
    - Добавление
    - Удаление

    - Редактирование
    - Сортировка по времени старта
    - Фильтрация по кораблям
    - Фильтрация по космонавтам
- Доступ к кораблям
    - Добавление
    - Редактирование
    - Удаление
    - Сортировка по названию

Роль – космонавт

Для роли космонавт мы обозначим следующие доступы:

- Просмотр списка кораблей
    - Сортировка по названию
- Просмотр списка задач
    - Сортировка по дате запуска
    - Фильтрация по кораблям
- Возможность запускать задачу, которая была создана для данного космонавта

Задачи, решаемые программой

Теперь, давайте обозначим главные задачи, которые должна решать программа, выделим, так сказать, её «логические экраны»:

- Экран авторизации
- Главный экран программы
- Экран списка пользователей
- Экран списка кораблей
- Экран списка задач запуска
- Экран самого запуска

Отлично, нам удалось выделить логические экраны поведения программы. Теперь, давайте обозначим задачи, которые будет решать каждый экран.

Экран авторизации

Данный экран будет отвечать за авторизацию пользователя в системе. Что конкретно будет происходить? На экране мы будем спрашивать у пользователя логин и пароль, затем обращаться к объекту хранилища Store из которого будем получать состояние нашего приложения State. В этом состоянии будет храниться набор объектов пользователей. Мы пробегаемся по этому набору и находим совпадение по логину и паролю, если оно найдено, то берём указатель на нужный объект пользователя и сохраняем этот указатель, также в State в поле текущего пользователя. Если пользователь не найден, возвращаем нулевой указатель и запрашиваем данные для входа ещё раз и так в цикле. После успешной записи текущего пользователя мы можем спокойно закрывать экран, перед этим указав экран, который мы хотим использовать после закрытия текущего. Кстати, здесь вы можете пойти двумя путями: первый путь, это сохранить текущий экран в стеке, чтобы после закрытия

главного экрана вернуться к нему. Второй путь – он более подходящий, полностью закрыть экран авторизации и открывать его только, если пользователь на главном экране выбрал пункт – выйти из учётных данных.

Главный экран

На данном экране будет отображено главное меню нашей программы. В этом меню будут отображены главные функции, которые можно выполнить. Эти функции будут открывать другие экраны. Самый последний пункт будет являться пунктом выхода из учётных данных. Это будет происходить следующим образом: мы очищаем указатель на текущего пользователя в хранилище, а затем закрываем текущий экран передавая управление экрану авторизации, ну а уже он будет выполнять свои функции. Заметим, что при этом сценарии развития событий, на не следует помещать главный экран в стек. Во всех остальных случаях, когда мы будем открывать другие экраны программы – мы обязательно должны сохранить его в стеке, чтобы после закрытия вернуться к нему. Также, стоит обратить внимание на само отображение пунктов меню. У нас есть роли доступа, каждой роле доступны свои пункты меню, например, космонавту не доступен доступ к пользователям, а также закрыта возможность редактирования задач и кораблей. Как нам это реализовать? Тут нужно вспомнить про модуль меню. Реализация модуля меню не будет представлять из себя нечто сложное. Это всего лишь класс – контейнер, который в себе содержит список других классов – пунктов меню. Каждый пункт меню содержит два поля: название пункта и запускаемую функцию. Меню запрашивает у пользователя индекс нужного пункта из списка и обращаясь к нему по индексу запускает функцию, которая содержится в этом поле. Также, в класс меню следует добавить возможность добавления нового пункта меню (опционально – удаление существующего).

Теперь, когда мы в экране будем формировать новое меню, мы будем производить проверку по роле пользователя и в зависимости от этого будем заполнять меню нужными пунктами с функциями. Как видите, всё достаточно просто.

По сути, больше никакой логикой этот экран обладать не будет. Двигаемся дальше.

Экран списка пользователей

Теперь рассмотрим, какая логика будет у нас в экране, который будет управлять пользователями. Всё аналогично главному экрану. Создаём объект меню, заполняем его нужными пунктами в зависимости от роли пользователя. Выполняем отрисовку и ждём действий от пользователя. В объекте данного экрана будут реализованы методы по управлению данными пользователей: метод добавление, удаление, редактирования и т д Эти методы будут получать данные из Store, что логично и удобно. Каких – то отличительных черт, на которых нужно заострить особое внимание на этом экране больше нет, поэтому двигаемся дальше. С экраном списка кораблей будет таже история, он не будет ничем отличаться от экрана списка пользователей.

Экран списка задач на запуск

По сути, экран списка задач, опять же, не отличается от прошлых экранов по функциональности за исключением способа перехода к экрану запуска задачи. Давай рассмотрим эту логику более подробно.

В меню данного экрана будет пункт, который отвечает за переход к запуску нужной задачи. При выборе этого пункта будет запускаться метод, который спрашивает id нужной задачи для запуска, дальше записывает в хранилище экран перехода (на который мы будем переходить, в данном случае это будет экран запуска задачи). Теперь, экрану запуска задачи нужен id той задачи, которую он будет запускать (логично), тут мы как раз приходим к теме передачи данных между экранами. Как и обговаривали ранее, здесь нет ничего сложного. Для этого мы просто записываем определённые данные в State, а затем не забываем выставить флаг помещения текущего экрана в стек, чтобы после закрытия экрана запуска задачи, мы могли вернуться к текущему экрану. Ну, а теперь закрываем экран.

Экран запуска задачи

Этот экран снова рассмотрим внимательно. При запуске должна стартовать логика, которая проверяет данные на предмет их существования. Какие данные? – те, которые пришли к нам из стора и те, которые в этот стор положил экран списка задач. Если данные были не найдены (указатель остался нулевым), то просто закрываем экран. Если всё хорошо и указатель на данные действительный, мы снимаем копию этих данных и обнуляем указатель в Store. После этого мы можем приступить к запуску задачи. Реализуем отдельный метод, в котором будет содержаться логика обратного отсчёта, после которого мы выставляем в поле задачки триггер на выполнено и спокойно выходим из экрана. При этом, можно добавить возможность ввода обратного отсчёта (от какого числа считать).

Теперь всё готово для начала написания данной логики на языке программирования C++. В этом руководстве будет представлено общее описания синтаксиса, при этом не будет уделено времени разбору отдельных стэйтментов, поскольку их логику работы вы можете найти на сайте ravesli.com.

Сама реализация

Теперь давайте соединим всё вместе. Рассмотрим базовую структуру нашего будущего проекта:

- App – контейнер основной программы
  - App.h
  - App.cpp
- Store – модуль глобального хранилища
  - Store.h
  - Store.cpp
  - State.h
  - State.cpp
  - Action.h
  - Action.cpp
  - ActionTypes.h

Весь код описанный в данном руководстве будет расположен по git репозитории по адресу:
https://github.com/DmitriyODS/laboratory-work-on-VP
Шаблон кода – библиотека, была построена по образцу 18 – ого варианта Л/Р

    - ActionTypes.cpp
- Menu – модуль интерактивного меню
    - Menu.h
    - Menu.cpp
    - ItemMenu.h
    - ItemMenu.cpp
- Models – модуль со всеми сущностями программы
    - Auth
        - Auth.h
        - Auth.cpp
    - Spaceship
        - Spaceship.h
        - Spaceship.cpp
    - LaunchTask
        - LaunchTask.h
        - LaunchTask.cpp
- ConsoleManager – модуль по работе с консолью, который можно реализовать, но не принципиально
    - ConsoleManager.h
    - ConsoleManager.cpp
- consoleUtils – здесь будут лежать функции, которыми мы будем пользоваться во всей нашей программе
    - consoleUtils.h
    - consoleUtils.cpp
- Db – модуль, который отвечает за логику сохранения данных и считывания их обратно
    - Db.h
    - Db.cpp
- global – в этой директории будут лежать глобальные ресурсы, которыми будет пользоваться наше приложение
    - config.h
    - config.cpp
- Screens – в этой директории будут лежать логические единицы нашей программы
    - Auth
        - Auth.h
        - Auth.cpp
    - LaunchTasksList
        - LaunchTasksList.h
        - LaunchTasksList.cpp
    - Launch
        - Launch.h
        - Launch.cpp
    - SpaceshipsList
        - Spaceships.h
        - Spaceships.cpp
    - Main
        - Main.h

        - Main.cpp
    - UsersList
        - UsersList.h
        - UsersList.cpp
- main.cpp – главный файл запуска

Весь код, как было сказано ранее, вы найдёте в архиве – приложении к этому руководству. В нём имеются комментарии, которые поясняют работу того, или иного кода, поэтому разобраться в нём не составит труда.

Ранее было озвучено, что, написав код для одного варианта Л/Р мы с лёгкостью сможем его модернизировать под другой вариант. Далее в руководстве мы рассмотрим, как адаптировать шаблон кода из архива, чтобы решить именно ваш вариант работы.

Адаптация шаблона к вашей задаче

Как вы уже поняли шаблон состоит из нескольких модулей, все они взаимосвязаны и работают. Это, по сути, решённый 18 – ый вариант. Идея в том, что вся структура проекта – масштабируема, а это значит, что вы с лёгкостью можете её перевести на свои рельсы разработки. Начнём адаптацию.

Начнём с адаптации моделей. Переходим в папку Models и создаём там свои подпапки с названиями своих моделей. В этих папках создаём по два файла .cpp и .h, и объявляем свою модель. Образец этого вы уже можете найти в папке Models. Старайтесь свои объекты делать структурами, чтобы избегать сложной логики. Если она у вас всё равно присутствует и её никак нельзя переделать, то оставляйте классы. После того, как вы заполнили папку моделей своими сущностями данных, вам нужно их прописать в CMakeLists.txt, если вы используете CMake.

Для этого, откройте названный файл и перейдите к строке «set(res User/User.cpp User/User.h …», измените содержимое после «res» на свои файлы, которые вы только что написали. Отлично, вы только что в свой код добавили модели! Идём дальше.

Теперь нам нужно определить списки этих моделей, для этого мы должны модифицировать структуру State, которая отвечает за глобальное хранение данных. Переходим по пути: Store/State.h. Здесь вам нужно будет выполнить подключение (include) своих моделей и на их основе создаём поля, типы которых будут – вектор указателей на сущность. Этот тип, вы должны были прописать в моделях данных, как псевдоним, образец уже есть в коде. После создания векторов данных моделей можно закрыть State и считать, что дело сделано. Теперь переходим в Store/ActionTypes.h и прописываем в в enum class константы всех событий, которые будут происходить в ваше программе.

После этого переходим в файл Store/Store.cpp и в методе reducer через switch прописываем обработку созданных вами событий по образцу. Этот метод должен не модифицировать существующее хранилище, а возвращать новое, с изменёнными данными. Также в деструкторе данного класса, следует позаботиться об удалении наборов данных, с которыми вы работаете. В методах: saveState и loadState, вам нужно сохранять и выгружать ваши данные из/в Store через объект Db, логику которого вам следует реализовать на своё усмотрение. С настройкой данных мы закончили, переходим к написанию логики.

Логика нашей программы строится из единиц поведения – экранов, которые лежат по пути Screens. Каждый экран должен находиться в своей поддиректории, а также наследоваться от интерфейса InterfaceScreen. Изучите шаблонный код в примере и напишите по его образцу свои экраны. Дальше в файле CMakeLists.txt в папке Screens, вы должны добавить созданные вами файлы в строчку – «set(res InterfaceScreen.cpp InterfaceScreen.h Auth/Auth.cpp Auth/Auth.h Launch/Launch.cpp Launch/…», после слова res, все остальные экраны, которые присутствуют в шаблоне можете удалить, кроме интерфейсного файла. После этого вам следует создать идентификаторы экранов для вашей программы, чтобы можно было к ним обращаться. Переходим по пути: global/config.h и в этом файле прописываем константные идентификаторы для экранов. После этого осталось только связать логику программы с вашим новым поведением это делать мы будем в классе – App.

Переходим в файл App/App.cpp, здесь, как мы с вами уже понимаем находится вся логика управления нашими рабочими экранами. Здесь же, нужно определить объекты ваших экранов. По сколько объекты ваших экранов являются синглтонами, то определять мы их будем, вызывая соответствующие методы: createScreen. В главном конструкторе App при создании начального объекта State ему передаётся указатель на словарь экранов, ключ которого – это ваш id экрана, который вы прописывали в глобальных переменных, а значение – указатель на экран, который возвращается при вызове статического метода: createScreen. По образцу заполняем словарь своими экранами. В деструкторе объекта App, не забываем удалять созданные объекты экранов из памяти по образцу. Теперь, необходимо задать экран, с которого начнёт запускаться наша программа, это делается в методе init. Получаем у state нужный нам экран по id и помещаем указатель на его в соответствующее поле.

На этом всё, после выполнения вышеперечисленных действий, шаблон вашей программы полностью адаптирован к вашей задаче. Осталось его только запустить.

Все вопросы, которые у вас могут появиться по шаблону, написанию своей логики и т д, вы можете задавать преподавателю, либо авторам данного руководства.