Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Новосибирский государственный технический университет»

Кафедра автоматизированных систем управления

МЕТОДИЧЕСКИЕ УКАЗАНИЯ И ЗАДАНИЯ

к лабораторным работам и практическим занятиям

по дисциплине

Технологии и методы программирования

для студентов, обучающихся по направлениям

09.03.01 – Информатика и вычислительная техника и 09.03.03 – Прикладная информатика

Новосибирск – 2024 г.

ОГЛАВЛЕНИЕ

Оглавление

[ВВЕДЕНИЕ 2](#_Toc156740795)

[ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ 3](#_Toc156740796)

[МОДЕЛИРОВАНИЕ СИСТЕМЫ УПРАВЛЕНИЯ ВИРТУАЛЬНОЙ ПАМЯТЬЮ 4](#_Toc156740797)

[РЕАЛИЗАЦИЯ ПРОГРАММЫ, УПРАВЛЯЕМОЙ ДАННЫМИ 8](#_Toc156740798)

[ПРОГРАММНЫЕ МОДУЛИ МНОГОКРАТНОГО ИСПОЛЬЗОВАНИЯ 12](#_Toc156740799)

[Динамически подключаемые библиотеки 12](#_Toc156740800)

[СЕТЕВОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ СОКЕТОВ 15](#_Toc156740801)

[ТЕСТИРОВАНИЕ ПРОГРАММНЫХ ПРОДУКТОВ 18](#_Toc156740802)

[ТЕСТИРОВАНИЕ МЕТОДАМИ «ЧЕРНОГО ЯЩИКА» 19](#_Toc156740803)

[ТЕСТИРОВАНИЕ МЕТОДАМИ “БЕЛОГО ЯЩИКА” 24](#_Toc156740804)

[ИСПОЛЬЗОВАНИЕ ШАБЛОНОВ 30](#_Toc156740805)

[СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ 36](#_Toc156740806)

[ТРЕБОВАНИЯ ПО ОФОРМЛЕНИЮ ПРОГРАММНОГО КОДА 40](#_Toc156740807)

[ШАБЛОН ТИТУЛЬНОГО ЛИСТА 44](#_Toc156740808)

[АЛГОРИТМ ПЕРЕВОДА СИМВОЛЬНОГО ВЫРАЖЕННИЯ В ПОЛИЗ 45](#_Toc156740809)

[ШАБЛОН РЕАЛИЗАЦИИ ПАТТЕРНА ADAPTER ДЛЯ КЛАССА НА ЯЗЫКЕ C++ 47](#_Toc156740810)

[ШАБЛОН РЕАЛИЗАЦИИ ПАТТЕРНА ADAPTER ДЛЯ ОБЪЕКТА НА ЯЗЫКЕ C++ 49](#_Toc156740811)

[ШАБЛОН РЕАЛИЗАЦИИ ПАТТЕРНА ADAPTER ДЛЯ КЛАССА НА ЯЗЫКЕ C# 51](#_Toc156740812)

[ШАБЛОН РЕАЛИЗАЦИИ ПАТТЕРНА ADAPTER ДЛЯ ОБЪЕКТА НА ЯЗЫКЕ C# 53](#_Toc156740813)

# ВВЕДЕНИЕ

В настоящее время программное обеспечение (ПО) является важнейшей составляющей информационных технологий. От него зависит функционирование подавляющего большинства компьютерной техники, электронных приборов и любое другое интеллектуальное оборудование, оно работает и в детских игрушках и решает сложные задачи по созданию искусственного интеллекта. Несовершенство технологического процесса его создания, неверные решения, принятые при реализации, могут привести к нежелательным результатам: срыву сроков готовности ПО; увеличению финансовых затрат; получению программного продукта, не удовлетворяющего предъявленным требованиям. Отсюда возникает задача подготовки высококвалифицированных специалистов, занятых в ИТ-индустрии и удовлетворяющих требованиям профессиональных стандартов 06.001 Программист, 06.028 Системный программист, 06.004 - Специалист по тестированию в области информационных технологий [5]. Настоящий цикл лабораторных работ и практических занятий должен внести свой вклад в решение этой задачи путём закрепления и расширения навыков программирования сложных программных продуктов. После освоения теоретического материала дисциплины “Технологии и методы программирования” и успешного решения предлагаемых задач, обучающийся должен:

Знать нормативную документацию (стандарты национальные и международные), регламентирующие процесс создания программных продуктов.

Уметь проектировать и реализовывать архитектурные решения программных продуктов, отлаживать и тестировать программные модули.

Владеть распространенными инструментальными средствами программирования.

# ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

В соответствии с [21], программное обеспечение – это программы, процедуры, правила и любая соответствующая документация, относящиеся к работе вычислительной системы. С этим термином связаны другие понятия, используемые в настоящем курсе:

**Программа - д**анные, предназначенные для управления конкретными компонентами системы обработки информации в целях реализации определенного алгоритма [28].

**Прикладная программа** - программа, предназначенная для решения задачи или класса задач в определенной области применения системы обработки информации.

**Программное средство - о**бъект, состоящий из программ, процедур, правил, а также, если предусмотрено, сопутствующих им документации и данных, относящихся к функционированию системы обработки информации.

**Программное обеспечение** - совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ.

**Программный продукт** - программное средство, предназначенное для поставки, передачи, продажи пользователю. Синонимом этого термина можно считать программное обеспечение.

**Пользователь программного средства - ю**ридическое или фактическое лицо, применяющее программное средство или участвующее в деятельности, прямо или косвенно зависящей от функционирования данного программного средства [ГОСТ 28806-90 Качество программных средств. Термины и определения].

# МОДЕЛИРОВАНИЕ СИСТЕМЫ УПРАВЛЕНИЯ ВИРТУАЛЬНОЙ ПАМЯТЬЮ

ЦЕЛЬ РАБОТЫ

Получить представление о механизме управления виртуальной памятью, закрепить навыки программирования на выбранном языке программирования с использованием динамических структур данных и прямого доступа к файлу.

**ОБЩИЕ СВЕДЕНИЯ**

Под виртуальной памятью понимают память, находящуюся вне адресного пространства задачи. С помощью механизма виртуализации можно снять ограничения на объем непосредственно адресуемого пространства, естественно в пределах дополнительной памяти, в качестве которой часто используется внешняя память.

Одна из возможных реализаций виртуальной памяти основана на страничной организации, когда весь объем необходимой памяти разбивается на фрагменты-страницы, располагаемые во внешней памяти. Некоторая часть страниц постоянно находится в оперативной памяти. Система управления по мере необходимости производит замещение (подкачку) недостающих страниц.

Страницы, располагаемые в оперативной памяти, снабжаются набором признаков (атрибутов), позволяющих эффективно управлять процессом замещения. В число атрибутов могут входить:

‑ флаг модификации (признак записи данных на страницу);

‑ флаг защиты от записи (признак запрета/разрешения записи);

‑ счетчик обращений к странице;

‑ счетчик времени пребывания страницы в памяти и т. п.

**ПОСТАНОВКА ЗАДАЧИ И РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ**

Реализовать класс для управления виртуальной памятью. Объект этого класса моделирует массив целого типа очень большой размерности (>10000 элементов).

При создании объекта инициализируется (заполняется 0) файл подкачки (двоичный файл прямого доступа). Файл содержит сигнатуру - два байта, представляющие символы ‘VМ’, и блоки (страницы) размерности 512 байт, Каждая страница состоит из элементов, соответствующих ячейкам моделируемого массива, и содержащих байты значений целого типа. Каждой странице предшествует битовая карта (массив байтов), в которой каждый бит соответствует ячейке моделируемого массива, находящейся на странице. Значение бита 0 означает, что в эту ячейку ничего не записано.

Количество страниц в файле определяется путём выравнивания суммарного объема памяти в байтах моделируемого массива на границу страницы.

Структура класса

{

Файловый указатель виртуального массива.

Буфер страниц – массив структур, соответствующих страницам, находящимся в памяти (>=3).

Конструктор инициализации с параметрами:

строка с именем файла (если отсутствует, имя файла задаётся по умолчанию);

размерность массив – длинное целое.

{

если файл не существует, создаёт файл в режиме rw (читать, писать), записывает сигнатуру и заполняет его нулями (0), иначе открывает его в режиме rw;

считывает заданное количество страниц (>=3), модифицируя атрибуты страниц (абсолютный номер, статус, время записи);

}

Метод определения номера (индекса) страницы в буфере страниц, где находится элемент массива с заданным индексом

{

определяет абсолютный номер страницы (порядковый номер страницы в файле), на которой находится требуемый элемент;

проверяет наличие страницы в памяти;

при отсутствии страницы в памяти выбирает самую старую страницу для замещения;

проверяет статус выбранной страницы (флаг модификации):

если страница модифицирована, выгружает ее в файл на предназначенное ей место;

если необходимо, загружает в память новую страницу вместе с битовой картой;

модифицирует атрибуты страницы:

записывает время обращения к странице, находящейся в памяти;

сбрасывает статус.

возвращает номер (индекс) страницы в буфере страниц или NULL, если обнаружена ошибка.

};

Метод чтения значения элемента массива с заданным индексом в указанную переменную

{

вычисляет номер (индекс) страницы в буфере страниц, на которой находится требуемый элемент;

вычисляет страничный адрес элемента массива с заданным индексом;

копирует значение с этого адреса в указанную переменную;

возвращает результат завершения операции.

}*;*

Метод записи заданного значения в элемент массива с указанным индексом

{

вычисляет номер (индекс) страницы в буфере страниц, на которой находится требуемый элемент;

вычисляет страничный адрес элемента массива с заданным индексом;

записывает значение по этому адресу;

модифицирует атрибуты страницы: статус, время записи, битовую карту;

возвращает результат завершения операции.

};

*Примечание:* можно переопределить операцию [] (квадратные скобки)

};

Структура страницы, находящейся в памяти

{

абсолютный номер страницы (порядковый номер страницы в файле);

статус страницы (флаг модификации) – байт, определяющий статус страницы (0 – страница не модифицировалась, 1 – если была запись);

время записи страницы в память;

битовая карта страницы;

массив значений моделируемого массива, находящихся на странице.

};

Ошибочные ситуации:

‑ недостаточно места на диске для размещения файла;

‑ ошибка файловой операции;

‑ недостаточно оперативной памяти для размещения объекта класса;

‑ индекс элемента выходит за границы массива

Тестирующая программа (консольное приложение):

‑ вводит несколько (2 или 3) значений, попадающих на разные страницы, чтобы обеспечить перемещение страниц;

‑ выводит на экран значения элементов массива до записи в массив и после чтения из массива.

‑ при завершении не уничтожает файл виртуального массива.

Этот файл уничтожается вручную после просмотра дампа файла при защите.

Представление результатов лабораторной работы:

‑ демонстрация тестирующей программы и дампа файла виртуального массива;

‑ отчет по лабораторной работе в электронном виде. Отчет должен содержать:

Титульный лист. Образец смотри приложение Б.

Текст задания.

Описание (можно графическое) всех структур, используемых в системе управления виртуальной памятью.

Исходный код класса и тестирующей программы, оформленный в соответствии с приложением А.

Фрагменты дампа файла виртуального массива. Фрагмент должен показывать адреса и внесённые значения.

Полезные ссылки для программистов:

<https://learn.microsoft.com/ru-ru/>

<https://learn.microsoft.com/ru-ru/dotnet/csharp/>

<https://professorweb.ru/>

<https://metanit.com/>

# РЕАЛИЗАЦИЯ ПРОГРАММЫ, УПРАВЛЯЕМОЙ ДАННЫМИ

**ЦЕЛЬ РАБОТЫ**

Ознакомиться c приёмами разработки программ, управляемых данными.

**ОБЩИЕ СВЕДЕНИЯ**

Программирование, управляемое данными представляет собой метод или даже парадигму программирования, при котором программный код, хотя и отделён от входных данных, но спроектирован таким образом, что логика программы определяется входными данными. В программе, управляемой данными, часть или даже все её свойства устанавливаются во время выполнения, что особенно важно, если программа составляется пользователем или должна им изменяться без перекомпиляции.

При создании программ, управляемых данными, код и структуры данных, на которые он воздействует, четко отделяются друг от друга и проектируются так, чтобы можно было изменять логику программы путем редактирования не кода, а структуры данных. В этом случае данные представляют собой не просто состояние некоторого объекта, а фактически определяют управляющую логику программы.

Программы, управляемые данными, получают некоторые из своих инструкций в форме данных, а не в форме кода. Типичный пример - сообщения об ошибках. Вместо того, чтобы в программе держать таблицу кодов ошибок и сообщений об ошибках, на которые программист ссылается при написании своего кода, программа при запуске считывает таблицу из внешнего источника (файла или базы данных). Другой пример - это конечный автомат. Программист пишет механизм конечного автомата, но состояния и переходы между ними определяются файлом данных, который загружается в приложение.

Хотя дизайн, управляемый данными, действительно предотвращает объединение данных и функциональности, в некоторых случаях утверждается, что программирование, управляемое данными, приводит к плохому объектно-ориентированному дизайну, особенно при работе с более абстрактными данными. Это связано с тем, что объект (сущность), управляемый исключительно данными, определяется способом их представления . Любая попытка изменить структуру объекта немедленно нарушит функции, которые на него возлагаются.

**ПОСТАНОВКА ЗАДАЧИ И РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ**

1. Реализовать класс, конструктор которого создаёт меню, структура которого определяется данными, находящимися во внешнем текстовом файле. Аргументом конструктора будет имя файла (по умолчанию, menu.txt). Меню программы рассматривается как множество деревьев (иерархий), корнями которых являются пункты главного меню (уровень иерархии 0). Следующие уровни иерархии представляют подпункты, при этом количество уровней произвольное для каждого пункта (подпункта). Каждая запись файла меню имеет следующую структуру:

Номер\_уровня\_в\_иерархииПробелНазвание\_пунктаПробелСтатус\_пунктаПробелИмяМетода

Последовательность записей соответствует последовательности пунктов. Статус пункта определяется:

0 – пункт виден и доступен;

1 – пункт виден, но не доступен;

2 – пункт не виден.

ИмяМетода – имя обработчика клика мыши на пункте меню.

Если имя обработчика не указано, это означает, что пункт имеет подменю, которое должно появиться после выбора этого пункта.

1. Внешний вид меню бизнес-приложений (предоставляемая функциональность), как правило, определяется ролью пользователя (его правами на доступ к функциям и данным). Список возможных пользователей будет храниться в файле USERS, записи которого имеют следующую структуру.

#Имя\_пользователяПробелПароль

Название\_пунктаПробелСтатус\_пункта\_для этого пользователя

Другие пункты, доступные пользователю

#Имя\_пользователяПробелПароль

Название\_пунктаПробелСтатус\_пункта\_для этого пользователя

Другие пункты, доступные пользователю

По умолчанию, отсутствующий в списке пункт имеет статус 0 (виден и доступен).

3 Любым текстовым редактором создать требуемые файлы. По умолчанию расширение имени файлов **.txt** .

3. Написать тестовую программу (оконное приложение) для проверки работы класса. При запуске программа открывает окно авторизации (см. рис. 1). Внешний вид окна должен быть как на рисунке. Имя приложения может быть произвольным. Номер версии приложения является параметром проекта.

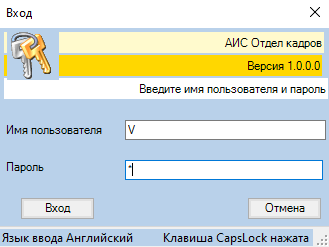


Рисунок 1 – Окно авторизации пользователя

Если авторизация прошла успешно, открывается главное окно программы с меню. (см. рис.2), сформированным для этого пользователя.

4. При выборе мышью пункта срабатывает обработчик пункта или появляется подменю, если имя метода не указано. Обработчик пункта должен вызвать метод с указанным именем, который просто обозначит себя.

5. При демонстрации результата внести изменения в файлы меню:

1) Сменить статус некоторых пунктов (подпунктов).

2) Добавить один или несколько пунктов (подпунктов).

Представление результатов лабораторной работы :

‑ демонстрация тестирующей программы;

‑ отчет по лабораторной работе в электронном виде. Отчет должен содержать:

Титульный лист. Образец смотри приложение Б.

Текст задания.

Исходный текст класса и тестирующей программы, оформленные в соответствии с приложением А.

. Тексты управляющих файлов.

Скриншот экрана для каждого пользователя.

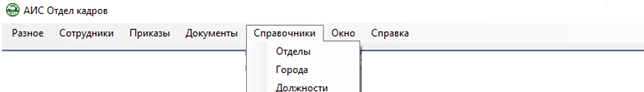


Рисунок 2 - Пример меню оконного приложения

Файл, содержащий описание этого меню, содержит следующие записи:

0 Разное 0 Others

0 Сотрудники 0 Stuff

0 Приказы 0 Orders

0 Документы 0 Docs

0 Справочники 0

1 Отделы 0 Departs

1 Города 0 Towns

1 Должности 0 Posts

0 Окно 0 Window

0 Справка 0

1 Оглавление 0 Content

1 О программе 0 About

Для этого примера список пользователей содержит:

Имя – Boss. Он имеет доступ ко всем функциям (пунктам меню), кроме Справочники.

HR – инспектор отдела кадров. Он имеет полный доступ к функциям Сотрудники, Приказы и Справочники.

OO – сотрудник общего отдела. Он имеет полный доступ к функции Документы и Справочники.

Пункты Разное, Окно и Справка (Оглавление, О программе) доступны всем.

Полезные ссылки для программистов:

<https://learn.microsoft.com/ru-ru/>

<https://learn.microsoft.com/ru-ru/dotnet/csharp/>

<https://professorweb.ru/>

<https://metanit.com/>

# ПРОГРАММНЫЕ МОДУЛИ МНОГОКРАТНОГО ИСПОЛЬЗОВАНИЯ

**ЦЕЛЬ РАБОТЫ**

Ознакомиться с приёмами создания программных модулей многократного использования, научиться применять их при проектировании и разработке программного обеспечения.

**ОБЩИЕ СВЕДЕНИЯ**

Говоря о многократном использовании программного кода, прежде всего, упоминают такие конструкции как **процедуры** и **функции**. С появление объектно-ориентированного подхода к программированию, этот список пополнился **классами**. Использование таких конструкций возможно разными способами:

1. В виде исходного кода, что делает этот код открытым, затрудняет распространение, поскольку требуется перекомпиляция и сборка приложений с новыми модулями.
2. В виде соответствующих библиотек (DLL) или COM-объектов, содержащих готовые к использованию программные компоненты.

## ****Динамически подключаемые библиотеки****

**DLL (Dynamic Link Library, динамически подключаемая библиотека) -** это набор кода и данных, объединенных по смыслу и направленности, которые могут одновременно использоваться несколькими программами.

**DLL** позволяют организовать модульный подход к созданию программ. Иными словами, вместо того, чтобы каждый раз для каждой программы создавать одни и те же возможности (например, писать функции, выполняющие типовые операции), они упаковываются в библиотеки, которые могут использоваться в разных проектах.

**Применение** DLL подразумевает следующие преимущества:

1. Используется меньше ресурсов компьютера. Во-первых, вместо того, чтобы каждая программа копировала одни и те же функциональные модули, используется всего одна библиотека (это позволяет экономить место на жестком диске). Во-вторых, в оперативную память компьютера библиотека загружается только один раз, и все программы вызывают только ее код.

2. Возможность создания приложений модульной архитектуры. Модульность подразумевает возможность создания программ, расширение функциональности которых происходит не за счет корректировки кода, а за счет подключения дополнительных библиотек.

3. Простота модификации. – все изменения локализованы в одном библиотечном модуле и для того, чтобы все программы могли учесть эти изменения, достаточно заменить лишь один файл библиотеки.

Приложение может использовать библиотеки двумя методами — связывание во время загрузки (неявное связывание) и связывание во время работы (явное связывание).

При связывании во время загрузки приложение должно указать на используемые библиотеки.

1. В программах на языке С++, чтобы использовать связывание во время загрузки, необходимо указать объявления импортированных данных, функций и классов C ++ из DLL. Все классы, функции и данные, импортируемые из DLL, должны быть помечены \_\_declspec (dllimport) в файле заголовка. Чтобы сделать код более удобочитаемым, можно определить макрос (#define DllImport \_\_declspec( dllimport ), а затем использовать этот макрос для каждого импортируемого символа:

При построении библиотеки DLL, как правило, создается файл заголовка, который содержит прототипы функций и (или) экспортируемые классы, а также добавляется ключевое слово \_\_declspec(dllexport) в раздел объявлений файла заголовка. Чтобы сделать код более удобочитаемым, можно определить макрос (#define DllExport \_\_declspec( dllexport ),

а затем использовать этот макрос для каждого экспортируемого символа:

// Пример dllimport и dllexport

\_\_declspec( dllimport ) int i;

\_\_declspec( dllexport ) void func().

Эти атрибуты явно определяют интерфейс DLL к модулю, которым может быть исполняемый файл или другая DLL.

1. В программах на языке С#, чтобы использовать связывание во время загрузки, необходимо добавить ссылку на используемую библиотеку.

Принеявном связывании компоновщик подключает все указанные библиотеки, при этом данный способ имеет недостатки и ограничения:

* все подключенные DLL загружаются всегда, даже если в течение работы программа ни разу не обратится ни к одной из них;
* если хотя бы одна из требуемых DLL отсутствует (или DLL не экспортирует хотя бы одной требуемой функции) – работа приложения невозможна, даже если отсутствие этой DLL некритично для исполнения программы.

При явном связывание во время работы приложение вызывает функцию для загрузки необходимой DLL (LoadLibrary для C++ или Assembly.Load для C# ). Это позволяет в программе обрабатывать ситуации с отсутствием DLL. После успешной загрузки DLL можно использовать все экспортируемые функции и классы.

Выбор того или иного способа связывания можно сделать на основании следующего:

1. Если важна начальная скорость запуска приложения, следует использовать связывание во время выполнения.
2. При связывании во время загрузки экспортные функции DLL похожи на локальные функции. Это упрощает вызов этих функций.
3. При связывании во время работы приложение может загружать различные модули по мере необходимости, возможно даже изменяя логику работы.

**ПОСТАНОВКА ЗАДАЧИ И РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ**

Описание класса и все модули, созданные при выполнении работы 2, оформить как DLL. Тестирующая программа должна повторить задание работы 2, используя созданную DLL и разные варианты связывания.

Представление результатов лабораторной работы :

‑ демонстрация тестирующей программы;

‑ отчет по лабораторной работе в электронном виде.

Отчет должен содержать:

Титульный лист. Образец смотри приложение Б.

Текст задания.

Исходный код тестирующей программы (оба варианта) и код библиотеки, оформленные в соответствии с приложением А.

.

Полезные ссылки для программистов:

<https://learn.microsoft.com/ru-ru/>

<https://learn.microsoft.com/ru-ru/dotnet/csharp/>

<https://professorweb.ru/>

<https://metanit.com/>

# СЕТЕВОЕ ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ СОКЕТОВ

**ЦЕЛЬ РАБОТЫ**

Получить навыки создания сетевых приложений на основе технологии Socket.

**ОБЩИЕ СВЕДЕНИЯ**

Сокетом называется специальный объект, создаваемый для отправки и получения данных через сеть. Чтобы две программы могли общаться друг с другом через сеть, каждая из них должна создать сокет. Каждый сокет обладает двумя основными характеристиками: протоколом и адресом, к которым он привязан. Протокол задается при создании сокета и не может быть изменен впоследствии. Адрес сокета задается позже, но обязательно до того, как через сокет пойдут данные.

**Виртуальный сетевой интерфейс**

Все реализации TCP/IP-протокола [] поддерживают **loopback** механизмы, которые реализуют виртуальный сетевой интерфейс исключительно программно и не связаны с каким-либо оборудованием, но при этом полностью интегрированы во внутреннюю сетевую инфраструктуру компьютерной системы. Любой трафик[[1]](#footnote-1), который посылается на этот канал, немедленно принимается тем же самым каналом. Любой трафик, который отправляется с этого интерфейса, но у которого адрес не **loopback**, отбрасывается. В компьютерах таким адресом **loopback** является адрес 127.0.0.1, он фиксированный и изменению не подлежит. Стандартное, официально зарезервированное, доменное имя[[2]](#footnote-2) для этого адреса — **localhost**.

Использование адреса 127.0.0.1 позволяет устанавливать соединение и передавать информацию для программ-серверов, работающих на том же компьютере, что и программа-клиент, независимо от конфигурации аппаратных сетевых средств компьютера (не требуется сетевая карта, модем, и прочее коммуникационное оборудование, интерфейс реализуется при помощи драйвера псевдоустройства в ядре операционной системы). Таким образом, для работы клиент-серверных приложений на одном компьютере не требуется изобретать дополнительные протоколы и дописывать программные модули. Примером может быть запущенный на компьютере веб-сервер и обращение к нему с этого компьютера для веб-разработки на этом компьютере без необходимости выкладывать веб-программу в сеть интернет, пока её разработка не закончена. То есть, если на компьютере, на котором запущен веб-сервер, указать в веб-браузере URL http://127.0.0.1/ или http://localhost/, то он попадает на веб-сайт этого компьютера.

**ПОСТАНОВКА ЗАДАЧИ И РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ**

**Задание 1:** Разработать программу-клиент, которая должна:

* запрашивать у пользователя адрес программы-сервера;
* устанавливать соединение с сервером;
* передавать на сервер имя выбранного каталога;
* принимать ответ от сервера и выводить его на экран;
* закрывать соединение с сервером.

**Задание 2:** Разработать программу-сервер, которая должна:

* ожидать запрос от программ-клиентов на соединение;
* устанавливать соединение с клиентами, при установлении соединения передавать список логических устройств клиенту;
* принимать от клиента данные:

если это имя каталога, передавать клиенту структуру каталога (имена файлов и подкаталогов);

если это имя текстового файла, передавать клиенту содержимое файла.

* сообщения, полученные от клиента, выводить на экран;
* закрывать соединение с клиентом при получении уведомления.

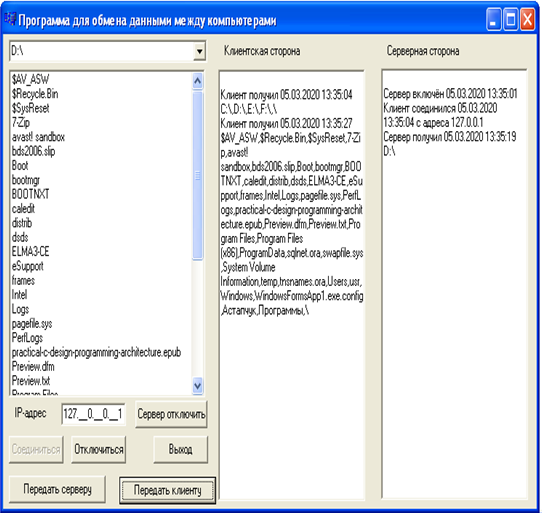
****

Рисунок 1 – Пример интерфейса программы, в которой размещается код клиента и код сервера

Представление результатов лабораторной работы :

‑ демонстрация тестирующей программы;

‑ отчет по лабораторной работе в электронном виде.

Отчет должен содержать:

Титульный лист. Образец смотри приложение Б.

Текст задания.

Снимок экрана при работе программы.

Исходный код модулей программы, оформленный в соответствии с приложением А.

Полезные ссылки для программистов:

<https://learn.microsoft.com/ru-ru/>

<https://learn.microsoft.com/ru-ru/dotnet/csharp/>

<https://professorweb.ru/>

<https://metanit.com/>

# ТЕСТИРОВАНИЕ ПРОГРАММНЫХ ПРОДУКТОВ

**ЦЕЛЬ РАБОТЫ**

Освоить приёмы создания тестов для тестирования программных модулей.

**ОБЩИЕ СВЕДЕНИЯ**

Проблема обнаружения и устранения ошибок обостряется по мере увеличения сложности задач, решаемых программами, что может грозить катастрофами в информационных системах (ИС), выполняющих критические функции управления крупными, дорогими и особенно важными объектами или процессами. Поэтому прежде чем программное обеспечение будет передано пользователям, его необходимо протестировать, чтобы снизить риск ошибок, оказывающих негативное влияние на его функционирование.

**Тестирование программного обеспечения -** проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. В более широком смысле, тестирование - это одна из техник контроля качества, включающая в себя действия по планированию работ, проектированию тестов, выполнению тестирования и анализу полученных результатов, при этом основная роль принадлежит проектированию тестов.

**Контрольный пример (тест)** - набор входных данных, условий выполнения и ожидаемых результатов, разработанный для определенной цели, например для выполнения определенного пути программы или подтверждения соответствия определенному требованию [ГОСТ Р ИСО/МЭК 25051—2017].

Для обнаружения всех ошибок в программе необходимо выполнить *исчерпывающее тестирование*, т.е. тестирование на всех возможных наборах данных (тестах). Для тех же программ, где исполнение команды зависит от предшествующих ей событий, необходимо проверить и все возможные последовательности.

Очевидно, что построение исчерпывающего входного теста для большинства случаев невозможно. Поэтому, обычно выполняется *"разумное" тестирование*, при котором тестирование программы ограничивается прогонами на небольшом подмножестве всех возможных входных данных. Естественно при этом целесообразно выбрать наиболее подходящее подмножество (подмножество с наивысшей вероятностью обнаружения ошибок).

Правильно выбранное подмножества (тест) должно обладать следующими свойствами:

1) уменьшать, причем более чем на единицу число других тестов, которые должны быть разработаны для достижения заранее определенной цели «разумного» тестирования:

2) покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Возможны разные подходы к проектированию тестов. Первый состоит в том, что тесты проектируются на основе внешних спецификаций программ и модулей (функциональное тестирование). В этом случае программа рассматривается как "черный ящик" и такое тестирование имеет целью выяснения обстоятельств, в которых поведение программы не соответствует спецификации.

Второй подход основан на анализе логики программы (стратегия ‘белого ящика’ или структурное тестирование). Суть метода - в проверке каждого пути, каждой ветви алгоритма. При этом внешняя спецификация во внимание не принимается. Другими словами, тестирование белого ящика смещает акцент с вопроса "что должен делать код" на "что фактически делает код".

## ТЕСТИРОВАНИЕ МЕТОДАМИ «ЧЕРНОГО ЯЩИКА»

Стратегия "черного ящика" включает в себя следующие методы формирования тестовых наборов:

* разбиение на классы эквивалентности;
* анализ граничных значений;
* анализ причинно-следственных связей;
* предположение об ошибке.

**Разбиение на классы эквивалентности**

Классом эквивалентности называется множество наборов входных данных (тестов), которые обрабатываются программой по одному алгоритму или приводят к одному результаты.

*Признаки эквивалентности тестов*:

* направлены на поиск одной и той же ошибки;
* если один из тестов обнаруживает ошибку, другие, скорее всего, тоже её обнаружат;
* если один из тестов не обнаруживает ошибку, другие, скорее всего, тоже её не обнаружат;
* тесты используют значения одних и тех же входных данных.

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

1. Выделение классов эквивалентности;
2. Построение тестов.

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза из спецификации) и разбиением его на две или более групп. Каждый тест должен включать по возможности максимальное количество различных входных условий, что позволяет минимизировать общее число необходимых тестов.

Для этого используется таблица следующего вида:

|  |  |  |
| --- | --- | --- |
| Входное условие | Допустимые классы эквивалентности | Недопустимые классы эквивалентности |

Допустимые классы включают правильные данные, недопустимые классы - неверные или недопустимые данные.

Выделение классов эквивалентности является эвристическим процессом, однако при этом существует ряд правил:

1. Если входные условия описывают интервалзначений, трактуемых однозначно, то набор данных должен содержать значения из интервала (допустимый класс) и значения вне интервала (недопустимый класс). Например, если входное условие - «целое данное может принимать значения от 1 до 99», то выделяют один правильный класс 1X99 и два неправильных X<1 и X>99.
2. Если входное условие описывает множество входных значений, среди которых можно выделить группы значений (группа может состоять из одного значения), трактуемых неоднозначно, то определяется правильный класс эквивалентности для каждой группы и один неправильный класс (значение вне множества). Например, если входное условие - «допустимое расширение имени входного файла .doc или .xls или .txt», то выделяют правильные классы и один неправильный (например, «,jpg») .
3. Если входное условие описывает множество входных значений, трактуемых однозначно, то определяется один правильный класс эквивалентности и один неправильный класс (значение вне множества). Например, входное условие - «первым символом идентификатора должна быть буква», то выделяют правильный класс (первый символ - буква) и один неправильный (первый символ - не буква) .

Процесс построения тестов включает в себя:

1. Назначение каждому классу эквивалентности уникального номера.
2. Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых классов эквивалентности, до тех пор, пока все правильные классы не будут покрыты (только не общими) тестами.
3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы не будут покрыты тестами.

Разработка индивидуальных тестов для неправильных классов эквивалентности обусловлено тем, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами.

Недостатком метода эквивалентных разбиения в том, что он не исследует комбинации входных условий.

**Анализ граничных значений.**

*Граничные условия* - это ситуации, возникающие на, выше или ниже границ входных классов эквивалентности. Границы классов эквивалентности порождают набор граничных значений.

Анализ граничных значений отличается от эквивалентного разбиения следующим:

1. Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных условий осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.
2. При разработке тестов рассматриваются не только входные условия (*пространство входов*), но и *пространство результатов*.

Применение метода анализа граничных условий требует определенной степени творчества и специализации в рассматриваемой проблеме. Тем не менее, существует несколько общих правил этого метода:

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений (например, для области входных значений от -1.0 до +1.0 необходимо написать тесты для ситуаций -1.0, +1.0, -1.001 и +1.001).
2. Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих двух значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то проверить 0, 1, 255 и 256 записей.
3. Использовать правило 1 для каждого выходного условия. Причем, важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей. Не всегда также можно получить результат вне выходной области, но, тем не менее, стоит рассмотреть эту возможность.
4. Использовать правило 2 для каждого выходного условия.
5. Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

Анализ граничных условий, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако следует помнить, что определение их связано с большими трудностями, что является недостатком этого метода. Второй недостаток связан с тем, что метод анализа граничных условий не позволяет проверять различные сочетания исходных данных.

**Анализ причинно-следственных связей.**

Метод анализа причинно-следственных связей помогает системно выбирать высокорезультативные тесты. Он дает полезный побочный эффект, позволяя обнаруживать неполноту и неоднозначность исходных спецификаций.

Для использования метода необходимо понимание булевской логики (логических операторов – И (AND), ИЛИ (OR), НЕ (NOT)). Построение тестов осуществляется в несколько этапов.

1) Спецификация разбивается на «рабочие» участки, так как таблицы причинно-следственных связей становятся громоздкими при применении метода к большим спецификациям. Например, при тестировании компилятора в качестве рабочего участка можно рассматривать отдельный оператор языка.

2) В спецификации определяются множество причин и множество следствий. *Причина* есть отдельное входное условие или класс эквивалентности входных условий. *Следствие* есть выходное условие или преобразование системы. Каждым причине и следствию приписывается отдельный номер.

3) На основе анализа семантического (смыслового) содержания спецификации строится таблица истинности, в которой последовательно перебираются все возможные комбинации причин и определяются следствия каждой комбинации причин. Истина обозначается "1". Ложь обозначается "0". Для обозначения безразличных состояний условий применяется обозначение "Х", которое предполагает произвольное значение условия (0 или 1).

Таблица снабжается примечаниями, задающими ограничения и описывающими комбинации причин и/или следствий, которые являются невозможными из-за синтаксических или внешних ограничений. При этом по возможности следует выделять независимые группы причинно-следственных связей в отдельные таблицы.

4) Каждая строка таблицы истинности преобразуется в тест. При этом по возможности следует совмещать тесты из независимых таблиц.

Недостаток метода - неадекватно исследует граничные условия.

**Пример применения методов тестирования «черным ящиком»**

Пусть необходимо выполнить тестирование программы, определяющей прямоугольность треугольника на основе анализа трёх произвольных чисел A, B, C, долженствующих обозначать длины его сторон.

1 Используя метод эквивалентных разбиений**,** получаем два правильных класса эквивалентности (множество целых чисел и множество вещественных чисел, удовлетворяющих условию A2+B2=C2) и два неправильных (положительные числа, неудовлетворяющие условию A2+B2=C2, и числа неположительные). Откуда можно предложить 5 тестов:

1. 3 положительных вещественных числа, удовлетворяющих условию A2+B2=C2;
2. 3 положительных целых числа, удовлетворяющих условию A2+B2=C2;
3. Все три числа =0.
4. Все три числа <0.
5. 3 положительных числа, неудовлетворяющие условию A2+B2=C2

2 По методу граничных условий*:*

* + 1. Все три числа =0
    2. Ввсе три числа =0.001
    3. Ввсе три числа =-0.001

d-e) Три положительных числа, удовлетворяющие условию A2+B2=C2±0.001

3 По методу анализа причинно-следственных связей:

Выделяем группы причинно-следственных связей (определение типа и существования треугольника) и строим таблицы истинности.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A<=0 | B<=0 | C<=0 | A2+B2=C2 | Результат |
| 1 | Х | X | Х | Не треугольник |
| Х | 1 | Х | Х | Не треугольник |
| Х | Х | 1 | Х | Не треугольник |
| 0 | 0 | 0 | 0 | Непрямоугольный |
| 0 | 0 | 0 | 1 | Прямоугольный |

Каждая строка этих таблиц преобразуется в тест., но учитывая, что первые три строки можно свернуть в одну, получаем четыре теста:

* 1. 3 положительных числа, удовлетворяющие условию A2+B2=C2
  2. 3 положительных числа, неудовлетворяющие условию A2+B2=C2
  3. все три числа =0.
  4. все три числа <0.

Порядок выполнения работы:

1. Ознакомиться с теоретическими сведениями по стратегиям тестирования.
2. Для предложенных вариантов разработать программу для каждого варианта.
3. Подготовить для этих программ тесты по методикам стратегии "черного ящика".
4. Предлагаемые тесты свести в таблицу.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер теста | Назначение теста | Значения исходных данных | Ожидаемый результат | Фактический результат | Результат тестирования |

1. Выполнить тестирование. Занести в таблицу результаты. В графу “фактический результат” записываются получаемые при прогоне значения выходных переменных. Если эти значения совпадают с ожидаемыми, то в программе ошибка не обнаруживается и в графе “результат тестирования” записываем “неуспешно”, что может свидетельствовать как об отсутствии ошибок, так и о наличии в программе ошибок, которые данным тестом не были найдены.
2. Сделать вывод о роли тестирования с использованием стратегии "черного ящика" и возможностях его применения. Сформулировать его достоинства и недостатки.

## ТЕСТИРОВАНИЕ МЕТОДАМИ “БЕЛОГО ЯЩИКА”

Стратегия **“**белого ящика**”** или структурное тестирование - это тестирование внутренней структуры, дизайна и кодирования программного решения. В этом типе тестирования программный код виден тестеру.

Методы, с помощью которых реализуется эта стратегия тестирования, включают:

1. Покрытие операторов - этот метод гарантирует, что каждый оператор в коде выполняется хотя бы один раз, чтобы упростить поиск ошибочного кода.
2. Покрытие решений (переходов) - с помощью этого метода проверяется каждый возможный путь или точка принятия решения в программе.
3. Покрытие условий - проверяются все отдельные условия в точках переходов.
4. Покрытие решений/усло­вий - все возможные результаты каждого условия в решении должны выполняться, по крайней мере, один раз, все результаты каждого решения должны выполняться, по крайней мере, один раз и, кроме того, каждой точке входа должно передаваться управление, по крайней мере, один раз.
5. Комбинаторное покрытие условий - все возможные комбинации результатов условия в каждом решении должны выполняться, по крайней мере, один раз.

Примеры реализации разных методов структурного тестирования показаны для фрагмента программы, блок-схема которого приведена на рис. 1.



Рисунок 1 – Блок-схема фрагмента программы

**Метод покрытия операторов**

Целью этого метода тестирования является выполнение каждого оператора программы хотя бы один раз. Для фрагмента на рис.1 можно выполнить каждый оператор, записав один единственный тест, который реализовал бы путь *ace.* Т.е., если бы на входе было: А=2, В=0, Х=3, каждый оператор выполнился бы один раз.

Таблица.1

Результат тестирования методом покрытия операторов

|  |  |  |  |
| --- | --- | --- | --- |
| Тест | Ожидаемый  результат | Фактический  результат | Результат  тестирования |
| A=2, B=0, X=3 | X=2,5 | X=2,5 | неуспешно |

**Метод покрытия решений (покрытия переходов)**

Согласно данному методу, каждое направление перехода должно быть реализовано, по крайней мере, один раз.

Покрытие решений обычно удовлетворяет критерию покрытия операторов. Поскольку каждый оператор лежит на некотором пути, исходящем либо из оператора перехода, либо из точки входа программы, при выполнении каждого направления перехода каждый оператор должен быть выполнен.

Для программы, приведенной на рисунке 1, покрытие решений может быть выполнено двумя тестами, покрывающими пути {*ace, abd*}, либо {*aсd,abe*}. Пути {*aсd,abe*}покроим, выбрав следующие исходные данные: {A=3, B=0, X=3} и {A=2, B=1, X=1}.

Таблица 2

Результат тестирования методом покрытия решений

|  |  |  |  |
| --- | --- | --- | --- |
| Тест | Ожидаемый  результат | Фактический  результат | Результат  тестирования |
| A=3, B=0, X=3 | X=1 | X=1 | неуспешно |
| А=2, В=1, Х=1 | Х=2 | Х=1,5 | Успешно |

**Метод покрытия условий**

Лучшие результаты по сравнению с предыдущими методами может дать метод покрытия условий. В этом случае записывается число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз.

В предыдущем примере имеем четыре условия: {A>1, B=0}, {A=2, X>1}. Для реализации метода требуется достаточное число тестов, такое, чтобы реализовать ситуации, где A>1, A≤1, B=0 и B≠0 в точке *а* и A=2, A≠2, X>1 и X≤1 в точке *в*. Тесты, удовлетворяющие критерию покрытия условий и соответствующие им пути:

1. A=2, B=0, X=4 *ace*
2. A=1, B=1, X=0 *abd.*

Таблица 3

Результаты тестирования методом покрытия условий

|  |  |  |  |
| --- | --- | --- | --- |
| Тест | Ожидаемый результат | Фактический результат | Результат тестирования |
| A=2, B=0, X=4 | X=3 | X=3 | неуспешно |
| A=1, B=1, X=0 | X=0 | X=1 | успешно |

Метод покрытиярешений/усло­вий

Критерий покрытия решений/условий требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз, все результаты каждого решения выполнялись, по крайней мере, один раз и, кроме того, каждой точке входа передавалось управление, по крайней мере, один раз.

##### Два теста метода покрытия условий:

а) A=2, B=0, X=4 *ace*

б) A=1, B=1, X=0 *abd*отвечают и критерию покрытия решений/условий. Это является следствием того, что одни условия приведенных решений скрывают другие условия в этих решениях. Так, если условие А>1 будет ложным, транслятор может не проверять условия В=0, поскольку при любом результате условия В=0, результат решения ((А>1)&(В=0)) примет значение *ложь.* Следовательно, недостатком критерия покрытия решений/условий является невозможность его применения для выполнения всех результатов всех условий.

Другая реализация рассматриваемого примера приведена на рисунке 2. Многоусловные решения исходной программы разбиты на отдельные решения и переходы. Для удовлетворения критерию решений необходимо покрытие тестами всех результатов каждого простого решения. Для этого нужно покрыть пути HILP (тест А=2, В=0, Х=4), HIMKT (тест А=3, В=1, Х=0), HJKT (тест А=0, В=0, Х=0), HJKR (тест А=0, В=0, Х=2).*.*



Рисунок 2 - Реализация фрагмента программы с простыми логическими условиями

Протестировав алгоритм на рисунке 2, нетрудно убедиться в том, что критерии покрытия условий и критерии покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

**Метод комбинаторного покрытия условий**

Критерием, который отчасти более чувствителен к ошибкам в логических условиях, является комбинаторное покрытие условий. Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении выполнялись, по крайней мере, один раз. Набор тестов, удовлетворяющих критерию комбинаторного покрытия условий, удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.

По этому критерию в рассматриваемом примере должны быть покрыты тестами следующие восемь комбинаций:

а) A>1, B=0;

б)A>1, B≠0;

в) A≤1, B=0;

г) А≤1, B≠0;

д) A=2, X>1;

е) A=2, X≤1;

ж) А≠2, X>1;

з) А≠2, X≤1;

Для того чтобы протестировать эти комбинации, необязательно использовать все 8 тестов. Фактически они могут быть покрыты четырьмя тестами:

- A=2, B=0, X=4 {покрывает а, д};

- A=2, B=1, X=1 {покрывает б, е};

- A=0,5, B=0, X=2 {покрывает в, ж};

- A=1, B=0, X=1 {покрывает г, з}.

Результаты тестирования рекомендуется представлять в виде нижеприведенной таблицы 4.

Таблица 4

Итоги тестирования программы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тест | Путь | Ожидаемый  результат | Фактический  результат | Результат  тестирования |
| А=2, В=0, Х=4 | асе | X=3 | X= |  |
| А=2, В=1, Х=1 | аве | X=2 | X= |  |
| А=0, В=0, Х=2 | аве | X=3 | X= |  |
| А=1, В=0, Х=1 | abd | X=1 | X= |  |

В графу “фактический результат” записываются получаемые при прогоне значения выходных переменных. Если эти значения совпадают с ожидаемыми, то в программе ошибка не обнаруживается и в графе “результат тестирования” записываем “неуспешно”, что может свидетельствовать как об отсутствии ошибок, так и о наличии в программе ошибок, которые данным тестом не были найдены.

К преимуществам структурного тестирования программного обеспечения относятся:

1. Оптимизация кода для поиска ошибок.
2. Весь код тестируется, обеспечивая тщательную проверку.
3. Он легко автоматизируется.
4. Тестирование можно начинать на ранних этапах жизненного цикла разработки программного обеспечения, еще до того, как будет доступен графический интерфейс пользователя.

Порядок выполнения работы:

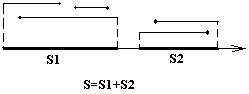
1. Ознакомиться с теоретическими сведениями по стратегиям тестирования.
2. Для программ, разработанных для тестирования по стратегии чёрного ящика, составить блок-схемы алгоритмов. Если размер программы велик, то выделить фрагмент кода, содержащий не менее двух операторов **if**.
3. Подготовить тесты по методам стратегии "белого ящика".
4. Для каждого метода предлагаемые тесты свести в таблицу.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Номер теста | Назначение теста | Значения исходных данных | Ожидаемый результат | Фактический результат | Результат тестирования |

1. Выполнить тестирование. Занести в таблицу результаты. В графу “фактический результат” записываются получаемые при прогоне значения выходных переменных. Если эти значения совпадают с ожидаемыми, то в программе ошибка не обнаруживается и в графе “результат тестирования” записываем “неуспешно”, что может свидетельствовать как об отсутствии ошибок, так и о наличии в программе ошибок, которые данным тестом не были найдены.
2. Сделать вывод о роли тестирования с использованием стратегии "черного ящика" и возможностях его применения. Сформулировать его достоинства и недостатки.

**ЗАДАНИЕ**

1. Разработать программу определения суммарной длины тени, которую отбрасывают на ось абсцисс отрезки, параллельные этой оси и заданные целочисленными абсциссами начала и конца отрезка.



1. Разработать программу перевода символьного арифметического выражения, возможно со скобками, в ПОЛИЗ (польскую инверсную запись). Использовать только целочисленные константы.

Алгоритм перевода выражения в ПОЛИЗ приведён в приложении Д.

**Представление результатов лабораторной работы:**

‑ демонстрация работающей программы;

‑ отчет по лабораторной работе в электронном виде. Отчет должен содержать:

Титульный лист. Образец смотри приложение Б.

Текст задания.

Таблицы тестов для каждого метода.

Исходный код программы, оформленный в соответствии с Приложением А.

Полезные ссылки:

<https://learn.microsoft.com/ru-ru/>

<https://learn.microsoft.com/ru-ru/dotnet/csharp/>

<https://professorweb.ru/>

<https://metanit.com/>

# ИСПОЛЬЗОВАНИЕ ШАБЛОНОВ

**ЦЕЛЬ РАБОТЫ**

Ознакомиться с основными шаблонами проектирования, научиться применять их при проектировании и разработке программного обеспечения.

**ОБЩИЕ СВЕДЕНИЯ**

Одним из основных преимуществ объектно-ориентированного подхода является возможность повторного использования кода. О повторном использовании кода говорят, когда рассматривают реализацию инкапсуляции и наследования. В этом случае классы разрабатывают так, чтобы их можно было использовать не в одном, а в нескольких проектах (инкапсуляция). Удачно определив базовое понятие и реализовав его в виде класса, его можно успешно и многократно использовать, определяя его наследников, реализуя их частные свойства.

Но в объектно-ориентированном подходе можно реализовать не только повторное использование кода, но и повторное использование проектных решений. Типовые проектные решения в объектно-ориентированном проектировании (да и не только [9-16]) называются паттернами (шаблонами).

При создании программных систем перед разработчиками часто встает проблема выбора тех или иных проектных решений. В этих случаях на помощь приходят паттерны. Дело в том, что почти наверняка подобные задачи уже решались ранее и уже существуют хорошо продуманные элегантные решения, составленные экспертами. Паттерны как раз описывают решения таких повторяющихся задач.

Паттерны проектирования упрощают повторное использование удачных про­ектных и архитектурных решений. Представление прошедших проверку време­нем методик в виде паттернов проектирования облегчает доступ к ним со стороны разработчиков новых систем. С помощью паттернов можно улучшить качество документации и сопровождения существующих систем, позволяя явно описать взаимодействия классов и объектов, а также причины, по которым система была построена так, а не иначе. Проще говоря, паттерны проектирования дают разра­ботчику возможность быстрее найти «правильный» путь

**Шаблон проектирования** или паттерн в разработке программного обеспечения — архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях.

Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие именно конечные классы или объекты приложения будут использоваться.

Правильное использование паттернов проектирования дает разработчику ряд неоспоримых преимуществ:

1) Модель системы, построенная в терминах паттернов проектирования, фактически является структурированным выделением тех элементов и связей, которые значимы при решении поставленной задачи.

2) Модель, построенная с использованием паттернов проектирования, более проста и наглядна в изучении, чем стандартная модель.

3) Несмотря на простоту, она позволяет глубоко и всесторонне проработать архитектуру разрабатываемой системы с использованием специального языка.

4) Применение паттернов проектирования повышает устойчивость системы к изменению требований и упрощает неизбежную последующую доработку системы.

5) Роль использования паттернов при интеграции информационных систем организации трудно переоценить.

6) Совокупность паттернов проектирования, по сути, представляет собой единый словарь проектирования, который, будучи унифицированным средством, незаменим для общения разработчиков друг другом.

7) Паттерн, реализованный в нужном месте, в нужное время, может стать настоящим спасителем разработчика.

Однако любой шаблон проектирования может стать палкой о двух концах: если он будет применен не к месту, то это может обернуться катастрофой и создать много проблем в последующем.

В общем случае, паттерн состоит из четырех основных элементов:

1. Имя. Сославшись на него, можно сразу описать проблему проектирова­ния, её решения и их последствия. Присваивание паттернам имен позволяет проектировать на более высоком уровне абстракции. С помощью словаря паттернов можно вести обсуждение с коллегами, упоминать паттерны в до­кументации, в тонкостях представлять дизайн системы. Нахождение хороших имен было одной из самых трудных задач при составлении каталога.

2. Задача. Описание того, когда следует применять паттерн. Необходимо сфор­мулировать задачу и ее контекст. Может описываться конкретная проблема проектирования, например способ представления алгоритмов в виде объектов. Иногда отмечается, какие структуры классов или объектов свидетель­ствуют о негибком дизайне. Также может включаться перечень условий, при выполнении которых имеет смысл применять данный паттерн.

3. Решение. Описание элементов дизайна, отношений между ними, функций каждого элемента. Конкретный дизайн или реализация не имеются в виду, поскольку паттерн – это шаблон, применимый в самых разных ситуациях. Просто дается абстрактное описание задачи проектирования и того, как она может быть решена с помощью некоего весьма обобщенного сочетания эле­ментов (в нашем случае классов и объектов).

4. Результаты – это следствия применения паттерна и разного рода компромиссы. Хотя при описании проектных решений о последствиях часто не упоминают, знать о них необходимо, чтобы можно было выбрать между различ­ными вариантами и оценить преимущества и недостатки данного паттерна. Поскольку в объектно-­ори­ентированном проектировании повторное использование зачастую является важным фактором, то к результатам следует относить и влияние на степень гибкости, расширяемости и переносимости системы.

Выделяют три основных вида шаблонов проектирования: структурные; порождающие; поведенческие.

***Структурные шаблоны*** определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу. К ним относятся:

- Адаптер/Обёртка (Adapter/Wrapper);

- Мост (Bridge);

- Компоновщик (Composite);

- Декоратор (Decorator);

- Фасад (Facade);

- Приспособленец (Flyweight);

- Заместитель (Proxy).

***Порождающие шаблоны*** проектирования, абстрагируют процесс инстанцирования *(*создание экземпляра класса*)*. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов. Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту. К таким шаблонам относятся:

- Абстрактная фабрика (AbstractFactory);

- Строитель (Builder);

- Фабричный метод (Factory Method);

- Прототип (Prototype);

- Одиночка (Singleton).

***Поведенческие паттерны*** ‒ определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение:

- Цепочка обязанностей (ChainOfResponsibility);

- Команда (Command);

- Интерпретатор (Interpreter);

- Итератор (Iterator);

- Посредник (Mediator);

- Хранитель (Memento);

- Наблюдатель (Observer);

- Состояние (State);

- Стратегия (Strategy);

- Шаблонный метод (Template method);

- Посетитель (Visitor).

**ПОСТАНОВКА ЗАДАЧИ И РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ**

Паттерн Adapter принадлежит к структурным паттернам и преобразует интерфейс класса в другой, требуемый интерфейс. В соответствии с форматом описания паттернов паттерн определяется как:

1. Имя: Adapter/Wrapper (Адаптер/Обёртка) - структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.

2. Задача (Проблема): необходимо обеспечить взаимодействие несовместимых интерфейсов или создать единый устойчивый интерфейс для нескольких компонентов с разными интерфейсами.

3. Решение: преобразовать исходный интерфейс компонента к другому виду с помощью промежуточного объекта адаптера, то есть, добавить специальный объект с общим интерфейсом в рамках данного приложения и перенаправить связи от внешних объектов к этому объекту адаптеру.

4. Результаты – классы клиента и адаптируемого класса являются независимыми друг от друга. Адаптер может собирать информацию о вызываемых методах.

Паттерн **Adapter** применяется в следующих случаях:

* если требуется использовать существующий класс, но интерфейс этого класса не отвечает требованиям программы;
* если нужно создать класс, который используется повторно. В этом случае класс должен взаимодействовать с другими, заранее не известными, классами, которые имеют несовместимые с программой интерфейсы;
* если необходимо использовать несколько существующих подклассов в одном классе. Это случай, когда применяется адаптер объектов для приспособления этих подклассов к общему родительскому классу.

На рисунках 1 и 2 схематично изображены особенности реализации паттерна **Adapter**.

Паттерн имеет следующих участников:

**Client**: клиент — это класс или объект, который использует объекты **Target** для реализации своих задач. хочет использовать открытый API **Adaptee**.

Класс **Adaptee** – это класс из другого модуля или сторонней библиотеки, содержащий метод **SpecificRequest(),** который нужно адаптировать к потребностям клиента, использующего метод **Request().** Метод **Request()** объявляется в интерфейсе **Target**, ссылку на который получает **Client,** ничего не знающего об **Adaptee.**

Между интерфейсом **Target** и классом **Adaptee** создается связующий класс **Adapter,** который обеспечивает общий интерфейс между **Adaptee** и **Client**, выполняя необходимые преобразования. Через интерфейс **Target** клиент **Client** обращается к методам, реализованным в классе **Adapter**. **Адаптер**, в свою очередь вызывает методы класса **Adaptee**.

Существует два способа реализации класса **Adapter**:

* способ наследования (рисунок 1). Этот способ получил название адаптер для класса. При этом способе класс **Adapter** наследует методы класса Adaptee и реализует методы интерфейса **Target. Шаблон кода приведён в приложении Г и Д**;
* способ композиции объектов (рисунок 2). Этот способ называется адаптер для объекта. При этом способе класс Adapter использует ссылку (экземляр, объект) типа **Adaptee**. **Шаблон кода приведён в приложении Е и Ж.**

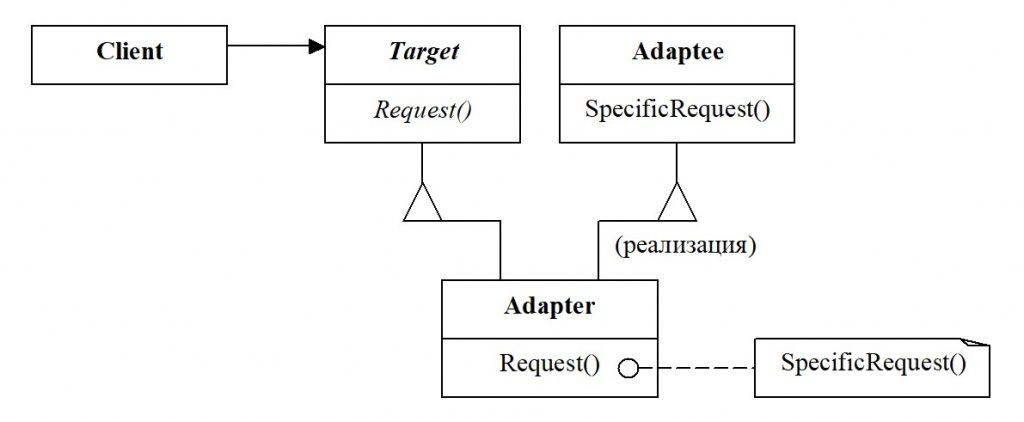
[](https://www.bestprog.net/wp-content/uploads/2020/09/13_01_01_01r.jpg)

Рисунок 1 - Структура паттерна Adapter для класса

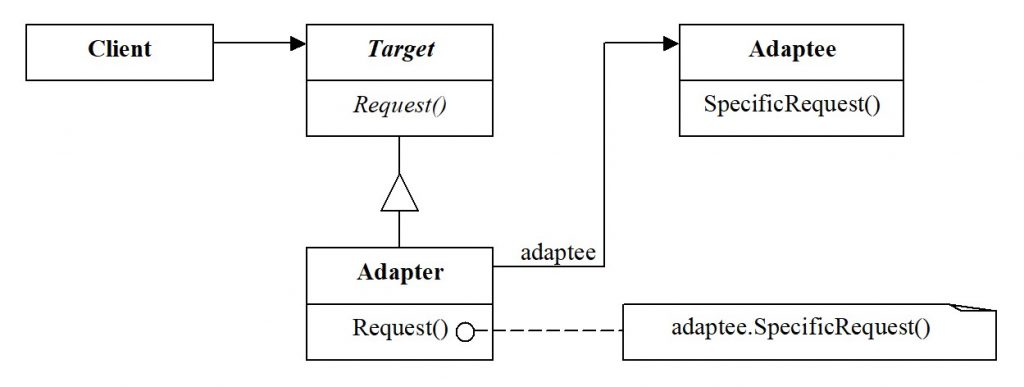
[](https://www.bestprog.net/wp-content/uploads/2020/09/13_01_01_02_.jpg)

Рисунок 2 - Структура паттерна Adapter для объекта

**Задание**

* 1. Познакомиться со списком известных шаблонов проектирования.
  2. Шаблон “Адаптер”. Проект “Часы”. В проекте должен быть реализован адаптер, который дает возможность пользоваться часами со стрелками так же, как и цифровыми часами. Для установки времени цифровые часы требуют формат представления HH:MM, а “Часы со стрелками” должны получать углы поворота стрелок от их текущего положения.
  3. Нарисовать в UML диаграмму классов реализуемой программы (проектирование), используя любой доступный пакет, поддерживающий нотацию UML. Это может быть MS Visio, ArgoUML (https://argouml.ru.uptodown.com/windows), StarUML (https://staruml.io).
  4. Реализовать программу на выбранном языке программирования (реализация). Образцы (шаблоны) программных модулей приведены в приложениях 2 - 5.

Представление результатов лабораторной работы :

‑ демонстрация тестирующей программы;

‑ отчет по лабораторной работе в электронном виде. Отчет должен содержать:

Титульный лист. Образец смотри приложение Б.

Текст задания.

Снимок экрана при работе программы.

Исходный код программы, оформленный в соответствии с Приложением А.

Полезные ссылки для программистов:

<https://learn.microsoft.com/ru-ru/>

<https://learn.microsoft.com/ru-ru/dotnet/csharp/>

<https://professorweb.ru/>

<https://metanit.com/>

# СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

**Основная учебная литература**

1. Гагарина, Л. Г. Технология разработки программного обеспечения: учебное пособие / Л.Г. Гагарина, Е.В. Кокорева, Б.Д. Сидорова-Виснадул ; под ред. Л.Г. Гагариной. — М. : ФОРУМ : ИНФРА-М, 2023. — 400 с.
2. Конструирование программного обеспечения: Учебное пособие / Л. Г.Гагарина, А. В. Гаращенко, Е. Н. Акимова [и др.]. - М. : ФОРУМ : ИНФРА-М, 2023. — 319 с.
3. Иванова Г. С. Технология программирования. Проектирование. Комплексное тестирование. Программная документация. Учебник / Г. С. Иванова – М.: Кнорус, 2022.-336 с.
4. Гниденко, И. Г. Технологии и методы программирования : учебное пособие для вузов / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. — 2-е изд., перераб. и доп. — Москва : Издательство Юрайт, 2023. — 248 с..

**Дополнительная учебная литература**

1. Херманс Ф. Ум программиста. Как понять и осмыслить любой код / Ф. Херманс – СПб.: БХВ, 2023. – 272 с.
2. Орлов С.А. Программная инженерия. Учебник для вузов. 5-е издание обновленное и дополненное / С.А. Орлов - СПб.: Питер, 2022.- 640 c.
3. Вирт Н.Алгоритмы и структуры данных. Новая версия для Оберона / Н.Вирт - М.: ДМК Пресс, 2010.- 272 с.
4. Вирт Н. Алгоритмы + структуры данных = программы / Н.Вирт - М.: Мир, 1985.- 406 с.
5. Нобак, М. Принципы разработки программных пакетов: практическое пособие / М. Нобак; пер. с англ. Д. А. Беликова. - Москва: ДМК Пресс, 2020. - 274 с.
6. Керниган Брайан У. Практика программирования / Брайан У. Керниган, Роб Пайк – М.: Вильямс, 2019.- 228 с.
7. Маклин Холл Гэри Адаптивный код: гибкое кодирование с помощью паттернов проектирования и принципов SOLID / Холл Гэри Маклин - М.: Вильямс, 2019.- 448 с.
8. Константайн Л. Разработка программного обеспечения / Л.  [Константайн](https://www.ozon.ru/person/konstantayn-l-1663927/), Л. [Локвуд](https://www.ozon.ru/person/lokvud-l-1663931/) – СПб.: Питер, 2004.- 592 с.
9. Мартин Р. Чистый код: создание, анализ и рефакторинг [Текст]/Р. Мартин – СПб.: Питер. 2019.- 464 с.
10. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения /Р. Мартин - СПб.: Питер. 2018.- 352 с.
11. Паттерны проектирования / Э.Фримен [и др.] - СПб.: Питер, 2018.- 656 c.
12. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма [и др.]- СПб.: Питер, 2016. – 368 с.
13. Смит Д. Элементарные шаблоны проектирования / Д. Смит - М.: Вильямс, 2013.- 304 c.
14. Мартин Р. Принципы, паттерны и методики гибкой разработки на языке C# / Р. Мартин, М. Мартин - СПб.: Символ-Плюс, 2011.-768 c.
15. Влиссидес Дж. Применение шаблонов проектирования. Дополнительные штрихи.: Пер. с англ. / Дж. Влиссидес—М.: Вильямс, 2003. — 144 с.
16. Шаллоуей А. Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию / А. Шаллоуей, Дж. Р. Тротт – М.: Вильямс, 2002. – 288 с.
17. Ларман К. Применение UML2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ и проектирование. 3-е издание. : Пер. с англ. / К. Ларман — М.: Вильяме, 2008. — 736 с.
18. Тепляков С. Паттерны проектирования на платформе .NET / С. Тепляков — СПб.: Питер, 2015. — 320 с.
19. Бейзер Б. Тестирование чёрного ящика. Технология функционального тестирования программного обеспечения и систем / Б. Бейзер - СПб.: Питер, 2004. - 320 с.
20. Майерс Г. Искусство тестирования программ / Г. Майерс - М.: Финансы и статистика,1982. - 174 с.
21. Тестирование программного обеспечения / Канер С. [и др].- М.: ДиаСофт, 2001. - 538 с.
22. Брауде Э. Технология разработки программного обеспечения: пер. с англ. / Э. Брауде – СПб.: Питер, 2004. – 656 с.

**Нормативные документы**

1. ГОСТ 33707-2016 Информационные технологии (ИТ). Словарь . – Введ. 01.09.2017. – М.: Стандартинформ, 2016. – 206 с.
2. ГОСТ Р ИСО/МЭК ТО 12182-2002 Информационная технология. Классификация программных средств. – Введ. 01.07.2003. – М.: Госстандарт России, 2003. – 16 с.
3. ГОСТ Р ИСО/МЭК ТО 9294-93 Информационная технология. Руководство по управлению документированием программного обеспечения. – Введ. 01.07.1994. – М.: Госстандарт России, 1994. – 20 с.
4. ГОСТ Р ИСО/МЭК 15910-2002 Информационная технология. Процесс создания документации пользователя программного средства. – Введ. 01.07.2003. – М.: Госстандарт России, 2003. – 49 с.
5. ГОСТ 19.301-2000 Единая система программной документации. Программа и методика испытаний. Требования к содержанию и оформлению . – Введ. 01.09.2001. – М.: Госстандарт России, 2001. – 11 с.
6. ГОСТ Р ИСО/МЭК 8631-94 Информационная технология. Программные конструктивы и условные обозначения для их представления . – Введ. 01.07.1995. – М.: Госстандарт России, 1995. – 16 с.
7. ГОСТ 19.201-78 Единая система программной документации. Техническое задание. Требования к содержанию и оформлению . – Введ. 01.01.1980. – Переизд. 01-2010 -М.: М.: Стандартинформ, 2010. – 4 с.
8. ГОСТ 2.105-2019 Единая система конструкторской документации. Общие требования к текстовым документам . – Введ. 29.04.2019. - М.: Стандартинформ, 2021. – 7 с.
9. ГОСТ Р 7.0.100—2018 Библиографическая запись. Библиографическое описание. Общие требования и правила составления . – Введ. 01.07.2019. - М.: Стандартинформ, 2018. – 124 с.
10. ГОСТ Р 7.0.99—2018 Система стандартов по информации, библиотечному и издательскому делу. Реферат и аннотация. Общие требования . – Введ. 01.01.2019. - М.: Стандартинформ, 2018. – 19 с.
11. ГОСТ 19.701-90 Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения . – Взамен ГОСТ 19.002-80, ГОСТ 19.003-80 - Введ. 01.01.1092. – Переизд. 01 2010 г. - М.: Стандартинформ, 2010. – 24 с.
12. ГОСТ Р ИСО 9127-94. Системы обработки информации. Документация пользователя и информация на упаковке для потребительских программных пакетов - Введён 01.07.1095 Переизд. 09.2003. - М.: ИПК Издательство стандартов, 2003. - 8 с.

Приложение А

(справочное)

# ТРЕБОВАНИЯ ПО ОФОРМЛЕНИЮ ПРОГРАММНОГО КОДА

(основаны нарекомендациях Майкрософт для разработки программ и документации на языке C#)

Программный код является одним из артефактов, создаваемых в жизненном цикле программного обеспечения и от того, насколько его структура отражает логику решения задачи, для которой он, собственно и написан, может зависеть качество программного продукта (изучаемость. модифицируемость, модульность, тестируемость, адаптируемость). Кроме того, ротация программистов может вызывать задержки в реализации проекта, поскольку на то, чтобы разобраться в чужом коде уходит в примерно 10 раз больше времени, чем на то, чтобы его изменить.

Для исключения упомянутых проблем, предлагаются стандарты (соглашения) стиля программирования на соответствующих языках [1 – 10 в списке полезных ссылок].

**Стандарт оформления кода** (**стандарт кодирования**, **стиль программирования**) - набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования. Наличие общего стиля программирования облегчает понимание, модификацию и поддержку исходного кода, написанного более чем одним программистом, а также упрощает взаимодействие нескольких человек при разработке программного обеспечения, кроме того, при соблюдении стандарта многие мелкие семантические ошибки (пропущенные или лишние символы и т.д.) становятся видны сразу в процессе разработки.

Учитывая, что функциональность, корректность и эффективность кода главная задача для программиста, возможны отступления от жёстких правил стиля, однако эти отступления должны быть обоснованы и описаны в соответствующей документации или в комментариях в исходном тексте.

**Соглашения о макете**

Для выделения структуры кода и облегчения чтения кода, следует использовать указанные приёмы форматирования:

Отступ в строке по четыре пробела (без табуляции).

Запись только одного оператора в строке.

Запись только одного объявления в строке.

Добавление, по крайней мере, одной пустой строки между определениями методов и свойств.

Каждая фигурная скобка начинается с новой строки (стиль Allman).

Однострочный оператор может быть без фигурных скобок.

Фигурные скобки можно опустить только в том случае, если тело каждого оператор, связанного с if/else if/.../else, расположено на одной строке.

Использовать скобки для ясности предложений в выражениях, как показано в следующем коде.

if ((val1 > val2) && (val1 > val3))

{

//программный код

}

**Соглашения о комментариях**

В зависимости от того, для чего нужны комментарии, их условно делят на два вида.

**Поясняющие** логику кода, использование алгоритмов. Комментировать нужно основные элементы кода, неочевидные решения, сложные бизнес-процессы, тонкости решений и тому подобное. Необходимо минимизировать подобные комментарии за счёт упрощения кода, стремясь сделать его самодокументированным за счёт соблюдения выбранного стиля программирования.

**Документирующие** — обязательные комментарии, содержащие информацию, относящуюся к фрагменту кода. Они располагаются в заголовках модулей, библиотек, процедур и функций и могут включать информацию о проекте, назначении модуля, имя разработчика, номер версии продукта и лицензию на программное обеспечение. Такие комментарии составляют основу технической документации на программный продукт.

В языке C# имеется специальный вид документирующих комментариев – XML-комментарии, начинающиеся с **///**. Документирующие комментарии вводятся перед объявлением таких элементов языка С#, как классы, пространства имен, методы, свойства и события. С помощью документирующих комментариев можно вводить в исходный текст программы сведения о самой программе. При компиляции программы документирующие комментарии к ней могут быть помещены в отдельный XML-файл. Кроме того, документирующие комментарии можно использовать в средстве IntelliSense интегрированной среды разработки Visual Studio [https://learn.microsoft.com/ru-ru/visualstudio/ide/using-intellisense?view=vs-2022].

**Форматирование поясняющих комментариев**

* Комментарий размещается на отдельной строке, а не в конце строки кода.
* Текст комментария начинается с заглавной буквы.
* Текст комментария завершается точкой.
* Между символом комментария **//** и текстом комментария вставляется один пробел.

**Соглашения об именах**

**Идентификатор** — это имя, присваиваемое типу (классу, интерфейсу, структуре, записи, делегату или перечислению), элементу управления, функции, переменной или пространству имен.

Имена следует образовывать от терминов предметной области таким образом, чтобы из имени было понятно назначение.

Идентификаторы должны начинаться с буквы или знака подчеркивания (не рекомендуется).

Односимвольные имена i, j, k и др. можно использовать только как переменные цикла или индексы массива. Не рекомендуется использовать букву l, поскольку её можно спутать с 1 (единицей).

Имена tmp, str и подобные можно использовать только для временных переменных, хранящих промежуточные значения.

Идентификаторы не должны содержать два последовательных символа подчеркивания ( \_ ). Эти имена зарезервированы для создаваемых компилятором идентификаторов.

Имена интерфейсов начинаются с заглавной буквы I.

В именах переменных, описанных как static (private или internal), используйте префикс **s\_** , а для именование потоков префикс **t\_**.

Имена типов атрибутов заканчиваются словом Attribute, но при использовании атрибута в коде этот суффикс можно не указывать.

**Соглашение о написании (использование прописных букв в идентификаторах)**

Существует два способа использования прописных букв в идентификаторах в зависимости от назначения идентификатора.

1. **PascalCasing** - прописной делается первая буква каждого слова (включая акронимы[[3]](#footnote-3) длиной более двух букв, с которых начинается идентификатор). Акронимы из двух букв пишутся полностью прописными. По соглашению программы C# используют PascalCasing для имен типов, пространства имен и всех открытых членов, а также при именовании interface, class, record или struct, полей, свойств, событий, методов и локальных функций, позиционных параметров.
2. **camelCasing** - прописным делается первый символ каждого слова, кроме первого слова. Акронимы из двух букв, с которых начинается идентификатор, названный в соответствии с этим соглашением, пишутся в нижнем регистре полностью. Этот стиль используется при именовании полей, отмеченных как private или internal, параметров.

.**ПОМНИТЕ** - программист пишет код для тех, кто будет читать написанное:

* оценивая профессионализм программиста;
* подхватывая и продолжая работу;
* возвращаясь к прерванной работе после длительного перерыва.

**Полезные ссылки**

1 <https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/coding-style/coding-conventions>

2 Цвалина, Кржиштоф  Инфраструктура программных проектов. Соглашения, идиомы и шаблоны для многократно используемых библиотек .NET (2-е издание): Пер. с англ. / Кржиштоф Цвалина,  Брэд Абрамс. - М.: ООО "И.Д. Вильямс", 2011. — 416 с.

3 Анисимов, А.Е.Требования и рекомендации по оформлению программного кода на языках С и С++ / А.Е. Анисимов. – Ижевск: Издательский центр «Удмуртский университет», 2020. – 48 с.

4 <http://stanford.edu/class/archive/cs/cs106b/cs106b.1158/styleguide.shtml>

5 <https://google.github.io/styleguide/cppguide.html>

6 <https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>

7 <https://www.elma-bpm.ru/KB/article-6651.html>

8 <https://www.csd.uoc.gr/~hy255/reading/c_coding_style.pdf>

9 <https://learn.microsoft.com/ru-ru/dotnet/csharp/>

10 <https://learn.microsoft.com/ru-ru/>

Приложение Б

(справочное)

# ШАБЛОН ТИТУЛЬНОГО ЛИСТА

Министерство науки и высшего образования Российской Федерации

Новосибирский Государственный технический университет

Кафедра автоматизированных систем управления



**Отчет по лабораторной работе №\_\_**

**по дисциплине «Технологии и методы программирования»**

**«**название работы**»**

Выполнил(а)

студент(ка) группы \_\_\_\_\_\_\_\_

Фамилия И.О.

г. Новосибирск 202\_ г

Приложение В

(справочное)

# АЛГОРИТМ ПЕРЕВОДА СИМВОЛЬНОГО ВЫРАЖЕННИЯ В ПОЛИЗ

**ПОЛИЗ** (польская инверсная запись) **–** это аббревиатура названия алгоритма преобразования символьного выражения из инфиксной нотации в постфиксную нотацию (в польскую инверсную запись), в выражение без скобок.

Инфиксная нотация -- это форма математических записей, в которой символ операции стоит между операндами, например, 3 + 4 или 3 + 4 \* (2 - 1).

При использовании постфиксной нотации символ операции стоит после операндов, например, 3 4+ или 3 4 2 1-\*+.

Алгоритм Дейкстра

Алгоритм использует входной массив, содержащий элементы выражения (лексемы) – операнды и символы операций (операторы), выходную строку, а также стек, хранящий ещё не добавленные к выходной строке операторы.

Стек – это структура данных, хранящая совокупность однотипных элементов, которые поступают в стек и выходят из него по правилу «последний вошел - первый вышел». В стек помещаются только символы операций.

Графически алгоритма Дейкстры можно представить следующим рисунком.

Выходная строка

С

Т

Е

К

Входная строка

*Операнды*

*Операции*

*Операции*

1) Если стек пуст, то оператор из входного массива помещается в стек.

2) Операнды из входного массива поступают сразу в выходную строку.

3) Если очередной оператор из входного массива имеет приоритет больший, чем верхний элемент (оператор) в стеке, то очередной оператор помещается в стек.

1. Если очередной оператор из входного массива имеет приоритет не больший, чем верхний элемент (оператор) в стеке, то из стека в выходную строку выталкиваются все операторы с большим или равным приоритетом, после чего очередной оператор помещается в стек.

5) Открывающаяся скобка всегда попадает в стек.

6) Закрывающаяся скобка ни в стек, ни в выходную строку не попадает. Когда она встречается во входном массиве, из стека выталкиваются все символы до первой открывающей скобки включительно. Символы операций выталкиваются в выходную строку, а открывающая скобка пропадает. Если стек закончился раньше, чем встретилась открывающая скобка, это означает, что выражение построено неверно, либо не согласованы скобки.

7) Приоритеты операций определяются таблицей.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Операция | **(** | **)** | **+ -** | **\* :** | **\*\*** | Унарный **+** **-** |
| Приоритет | 0 | 1 | 2 | 3 | 4 | 5 |

**ПРИЛОЖЕНИЕ Г**

(справочное)

# ШАБЛОН РЕАЛИЗАЦИИ ПАТТЕРНА ADAPTER ДЛЯ КЛАССА НА ЯЗЫКЕ C++

#include <iostream>

using namespace std;

// Паттерн Adapter для класса.

// Адаптирует класс Adaptee к классу Target через промежуточный класс Adapter.

// Абстрактный класс Target

class Target

{

public:

virtual void Request() = 0;

};

// Класс, метод которого нужно адаптировать к другой системе.

// В данном случае адаптируется имя метода SpecificRequest() в метод Request()

class Adaptee

{

public:

// Некоторый специфический метод

void SpecificRequest()

{

cout << "Adaptee.SpecificRequest()" << endl;

}

};

// Класс Adapter - использует множественное наследование

class Adapter :public Target, public Adaptee

{

public:

void Request()

{

// Вызывает SpecificRequest() из класса Adaptee

SpecificRequest();

}

};

// Класс Client - получает ссылку (или указатель) на Target

class Client

{

public:

// получает ссылку на Target

void ClientMethod(Target& target)

{

target.Request();

}

// получает указатель на Target

void ClientMethod(Target\* target)

{

target->Request();

}

};

void main()

{

// Демонстрация паттерна Адаптер для класса.

// нужно адаптировать метод SpecificRequest() класса Adaptee к потребностям экземпляра client

Client client;

Target\* target = (Target\*) new Adapter();

// передать клиенту экземпляр adaptee через интерфейс target

client.ClientMethod(target); // Adaptee.SpecificRequest()

}

**ПРИЛОЖЕНИЕ Д**

(справочное)

# ШАБЛОН РЕАЛИЗАЦИИ ПАТТЕРНА ADAPTER ДЛЯ ОБЪЕКТА НА ЯЗЫКЕ C++

#include <iostream>

using namespace std;

// Паттерн Адаптер для объекта.

// Абстрактный класс Target

class Target

{

public:

virtual void Request() = 0;

};

// Класс, в котором метод SpecificRequest() нужно передать клиенту как метод Request()

class Adaptee

{

public:

// Некоторый специфический метод

void SpecificRequest()

{

cout << "Adaptee.SpecificRequest()" << endl;

}

};

// Класс Adapter - использует композицию объектов

class Adapter :public Target

{

private:

Adaptee\* adaptee;

public:

// Конструктор принимает ссылку на adaptee

Adapter(Adaptee\* \_adaptee) :adaptee(\_adaptee)

{}

void Request()

{

// Вызывает SpecificRequest() из класса Adaptee

adaptee->SpecificRequest();

}

};

// Класс Client - получает ссылку (или указатель) на Target

class Client

{

public:

// получает ссылку на Target

void ClientMethod(Target& target)

{

target.Request();

}

// получает указатель на Target

void ClientMethod(Target\* target)

{

target->Request();

}

};

void main()

{

// Демонстрация паттерна Адаптер для объекта

// Экземпляр класса Client, который может получить код метода SpecificRequest() класса Adaptee

Client client;

// Экземпляр, содержащий метод SpecificRequest()

Adaptee adaptee;

// Ссылка на базовый класс, которая будет передаваться клиенту

Target\* target;

// конвертирование adaptee->target через связующий класс Adapter

target = new Adapter(&adaptee);

// Передать клиенту target

client.ClientMethod(target);

}

**ПРИЛОЖЕНИЕ Е**

(справочное)

# ШАБЛОН РЕАЛИЗАЦИИ ПАТТЕРНА ADAPTER ДЛЯ КЛАССА НА ЯЗЫКЕ C#

// Паттерн Адаптер для класса. Решение на языке C#

using System;

using static System.Console;

namespace ConsoleApp10

{

// Адаптирует класс Adaptee к интерфейсу ITarget через промежуточный класс Adapter

// Інтерфейс ITarget

interface ITarget

{

public void Request();

}

// Класс, метод которого нужно адаптировать к другой системе.

// В данном случае адаптируется имя метода SpecificRequest() в метод Request()

class Adaptee

{

// Некоторый специфический метод

public void SpecificRequest()

{

WriteLine("Adaptee.SpecificRequest()");

}

}

// Класс Adapter - реализует интерфейс ITarget и наследует класс Adaptee

class Adapter : Adaptee, ITarget

{

// Реализация метода Request() интерфейса ITarget

public void Request()

{

// В середине метода вызывается метод SpecificRequest() класса Adaptee()

SpecificRequest();

}

}

// Класс Client - получает ссылку на интерфейс ITarget

class Client

{

// некоторый метод класса Client

public void ClientMethod(ITarget target)

{

target.Request();

}

}

class Program

{

static void Main(string[] args)

{

// Демонстрация паттерна Adapter для класса.

// Задание.

// Нужно адаптировать экземпляр класса Adaptee к потребностям экземпляра класса Client.

// 1. Создать экземпляр класса Client

Client client = new Client();

// 2. Создать экземпляр класса Adapter, который ссылается на интерфейс ITarget

ITarget target = new Adapter();

// 3. Передать клиенту target

client.ClientMethod(target);

}

}

}

**ПРИЛОЖЕНИЕ Ж**

(справочное)

# ШАБЛОН РЕАЛИЗАЦИИ ПАТТЕРНА ADAPTER ДЛЯ ОБЪЕКТА НА ЯЗЫКЕ C#

// Паттерн Адаптер для объекта. Решение на языке C#

using System;

using static System.Console;

namespace ConsoleApp10

{

// Адаптирует класс Adaptee к интерфейсу Target через промежуточный класс Adapter

// Интерфейс ITarget

interface ITarget

{

public void Request();

}

// Класс, метод которого нужно адаптировать к другой системе.

// В даном случае адаптируется имя метода SpecificRequest() в метод Request()

class Adaptee

{

// Некоторый специфический метод

public void SpecificRequest()

{

WriteLine("Adaptee.SpecificRequest()");

}

}

// Класс Adapter - реализует интерфейс ITarget

class Adapter : ITarget

{

private Adaptee adaptee; // ссылка на Adaptee

// Конструктор

public Adapter(Adaptee adaptee)

{

// инициализировать внутреннюю ссылку adaptee параметром

this.adaptee = adaptee;

}

// Реализация метода Request() интерфейса ITarget

public void Request()

{

// Всередине метода вызывается метод SpecificRequest() класса Adaptee()

adaptee.SpecificRequest();

}

}

// Класс Client - получает ссылку на интерфейс ITarget

class Client

{

// некоторый метод класса Client

public void ClientMethod(ITarget target)

{

target.Request();

}

}

class Program

{

static void Main(string[] args)

{

// Демонстрация паттерна Adapter для объекта.

// Задание.

// Нужно адаптировать экземпляр класса Adaptee к потребностям экземпляра класса Client.

// 1. Пусть заданы некоторые экземпляры классов Client и Adaptee

Client client = new Client();

Adaptee adaptee = new Adaptee();

// 2. Создать экземпляр класса Adapter, передать экземпляр класса Adaptee

//     в конструктор класса Adapter

ITarget target = new Adapter(adaptee);

// 3. Передать клиенту target

client.ClientMethod(target);

}

}

}

1. Трафик - данные, передаваемые через компьютерную сеть за любой период времени. [↑](#footnote-ref-1)
2. Доменное имя - символьное имя, служащее для идентификации областей, которые являются единицами административной автономии в сети Интернет, в составе вышестоящей по иерархии такой области. Каждая из таких областей называется ***доменом***. Общее пространство имён Интернета функционирует благодаря DNS — системе доменных имён. Доменные имена дают возможность адресации интернет-узлов и расположенным на них сетевым ресурсам (веб-сайтам, серверам электронной почты, другим службам) быть представленными в удобной для человека форме. [↑](#footnote-ref-2)
3. Акроним - аббревиатура, образованная из начальных букв, частей слов или словосочетаний, например, ФИО. [↑](#footnote-ref-3)