МИНОБРНАУКИ РФ

ФГБОУ ВО Тверской государственный технический университет

Кафедра “Программное обеспечение”

Курсовая работа

дисциплина «Теоретическая информатика»

Тема: Правила качественного программирования

Выполнили: студенты группы

«Б.ПИН.РИС-21.06»

Латыпов Тимофей Вадимович

Проверил: Биллиг Владимир Арнольдович

Тверь,2022

**Оглавление**

1. **Введение 3**
2. **Правила качественного программирования**
   1. **Декомпозиция 3**
   2. **Модульность 3**
   3. **Корректность 3**
   4. **Тестируемость 4**
   5. **Устойчивость 4**
   6. **Стилизованность 5**
   7. **Читаемость 5**
   8. **Универсальность 5**
3. **Практика качественного программирования**
   1. **Декомпозиция 6**
   2. **Модульность 7**
   3. **Корректность 12**
   4. **Тестируемость 12**
   5. **Устойчивость 13**
   6. **Стилизованность 13**
   7. **Читаемость 13**
   8. **Универсальность 13**
4. **Вывод 13**

# **1.Введение**

Правила качественного программирования – фундаментальные принципы программирования, определяющие то, как и зачем нужно проектировать структуру программы, алгоритмов, а также общие подходы к разработке.

# **2.Правила качественного программирования**

**2.1. Декомпозиция**

Декомпозиция – мыслительная операция по разделению целого на части. Это общий приём, применяемый при решении различных проблем, состоящий в разделении задачи на множество частных, тривиальных задач, не превосходящих по сложности исходную проблему, с помощью объединения решений которых, можно сформировать решение исходной проблемы в целом.

Программисты часто сталкиваются с очень абстрактными и сложными задачами, которые предполагают работу в команде. При командном подходе к разработке ПО, использование декомпозиции позволяет давать разработчикам конкретные небольшие задачи и постепенно выстраивать общую структуру приложения из небольших частиц. Также это увеличивает продуктивность разработки, так как появляется возможность параллельно решать несколько задач, и совмещать их на дальнейших этапах.

**2.2. Модульность**

Модульность – приём в программировании, заключающийся в разбиении программы на функционально законченные фрагменты, оформляемые в виде отдельного файла или поименованной непрерывной части программы. При таком подходе программа организована как совокупность небольших независимых блоков, структура и поведение которых подчиняется определённым правилам. Причём каждый модуль выполняет одну определённую задачу.

При написании объёмных комплексных решений, программисты сталкиваются с проблемами тестирования и обнаружения ошибок. Модульность как бы проливает свет на внутреннюю работу алгоритма, расставляет всё по своим местам и противостоит неявной логике и запутанности. Она позволяет быстро обнаружить модуль, содержащий ошибку и исправить её.

Модульность так же позволяет проще понимать общую логику решения и то, каким образом достигается результат.

**2.3. Корректность**

Под корректностью программы понимается её соответствие некоторому эталону или совокупности эталонных правил и характеристик. Корректность требует, чтобы программа была корректна на всех уровнях, начиная корректностью текста программы, до комплексной корректности, включающую в себя межмодульные связи, функциональность, динамичность.

Для программистов очень важно, чтобы программа была корректна, так как результат её работы – это результат работы программиста, показатель его навыков, опыта. Не так важно, чтобы приложение было быстрым, сколько чтобы оно работало так, как нужно.

**2.4. Тестируемость**

Тестируемость ПО – степень, в которой программная система или модуль поддерживает тестирование, то есть поиск неисправностей в системе. Тестируемость – очень важный фактор в программировании, так как она позволяет обеспечить другой важный фактор – корректность, и исправить не только текущие ошибки, но и исправить будущие.

Для программиста важно писать такие модули, которые будет легко тестировать, потому что на более поздних стадиях разработки, плохо протестированный модуль, может дать сбой в неординарных ситуациях, что повлечёт за собой глобальные изменения во всей программе, её архитектуре и логике.

Однако тестируемость – понятие, применимое не только к отдельным блокам, модулям в программе, но и ко всему приложению в целом. Для обеспечения тестируемости на ранних этапах разработки тестировщики создают специальные требования к программе, документацию, эталоны, которым обязаны соответствовать все модули программы. Такой строгий подход позволяет в будущем сэкономить огромное количество ресурсов, которые могли бы уйти на исправление множества ошибок и перестройку архитектуры приложения.

**2.5. Устойчивость**

Устойчивость – способность системы сохранять текущее состояние при влиянии внешних воздействий. Если текущее состояние при это не сохраняется, то такое состояние называется неустойчивым. Устойчивость ПО обычно обеспечивается с помощью введения различным форм избыточности, позволяющих иметь дублирующие модули программ, альтернативные программы для одних и тех же задач, осуществлять контроль над процессом исполнения программ.

Для пользователей важно, чтобы программа работала что бы они ни сделали. Программист должен рассчитывать на это и писать устойчивые программы, которые даже при неверных входных данных должны верно их обрабатывать, выводя ошибки или предупреждения.

**2.6. Стилизованность**

Стилизованность – принадлежность к одному стилю, схожесть определённых черт. Стилизованный код – код, написанный в одном стиле, по одним и тем же правилам, на протяжении всей программы.

В программировании очень часто приходится работать в команде, и часто приходится перечитывать ранее написанный код. В таком случае стилизованность – правило хорошего тона программирования. Когда программа написана в одном стиле, она проще, а значит быстрее, воспринимается и вызывает симпатию, так как просто красиво и структурированно выглядит.

Стилизованность проникает во все области программирования, начиная с одинакового стиля названия переменных, методов, классов, и заканчивая табуляцией и организацией файлов в решении.

**2.7. Читаемость**

Читаемость – характеристика, которая определяет, насколько легко вы можете понять некоторые части программы.

Читаемость – ещё одно правило хорошего тона. Читаемость позволяет человеку, не знакомому с программой, быстрее понять как она устроена и, например, помочь написавшему её в решении какой-то проблемы. На читаемость программы влияет множество факторов, начиная с опыта читающего и заканчивая стилем кода. Чтобы писать читаемый код, необходимо соблюдать табуляцию, модульность, стилизованность, декомпозицию, писать комментарии к коду, описывающие работу нетривиальных решений и не использовать чересчур сложных конструкций, таких как множество вложенных друг в друга методов или тернарных операторов.

**2.8. Универсальность**

Универсальность – возможность применения написанного кода в нескольких предметных областях, для решения различных задач.

В программировании универсальность выражается сильнее всего в общедоступных библиотеках – сборников подпрограмм или объектов, которые используются для решения более конкретных задач. Это позволяет не только не использовать один и тот же код повторно, но и формализовать, унифицировать, стандартизировать программирование, чтобы все программисты общались на одном и том же языке.

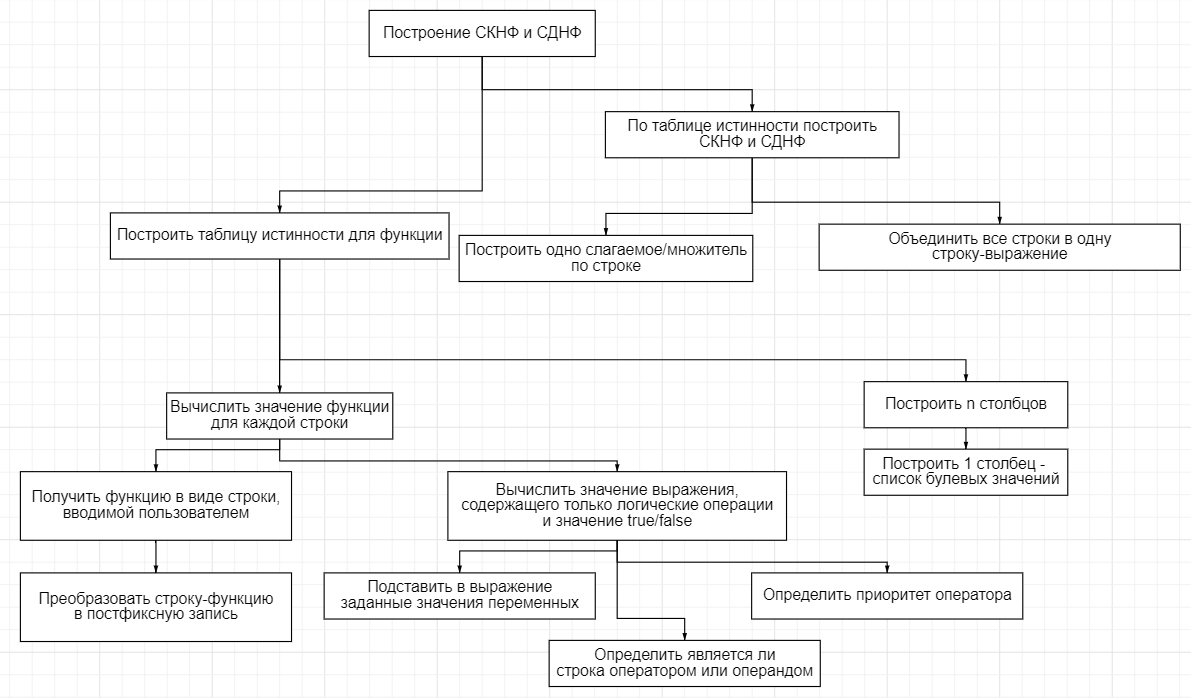
# **3.Практика качественного программирования**

(Латыпов Тимофей)

Рассмотрим применение вышеописанных приёмов на примере проекта «Логические функции». Его целью является построение совершенной дизъюнктивной и конъюнктивной нормальной формы логической функции.

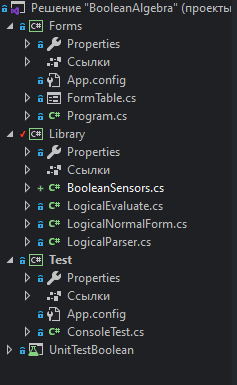
**3.1. Декомпозиция**

Для нашего проекта возможен следующий вариант декомпозиции:

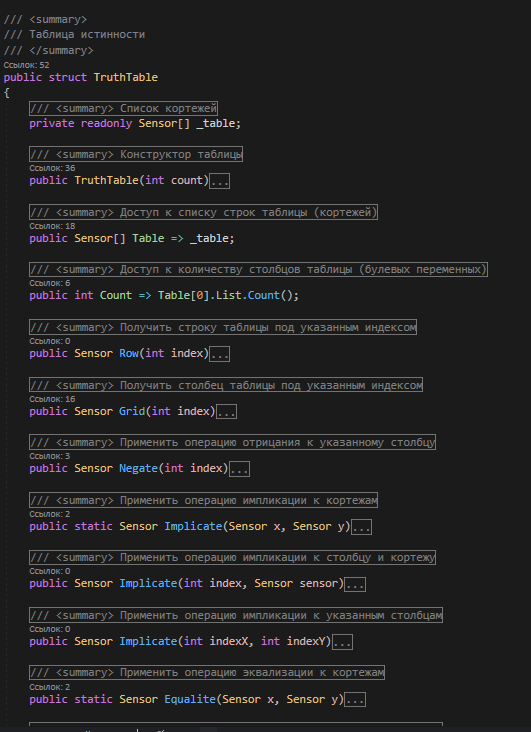


**3.2. Модульность**

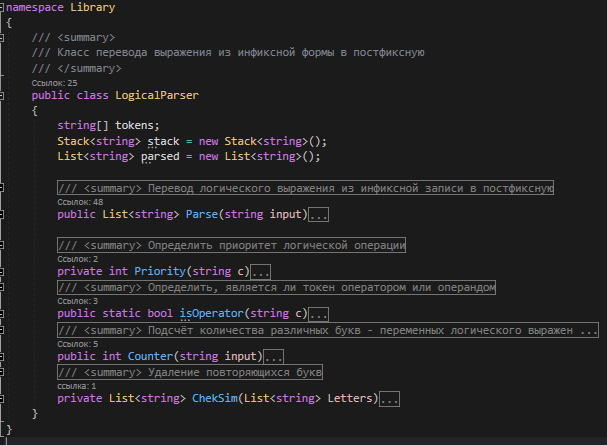
В нашем проекте роли модулей выполняют классы библиотеки классов.



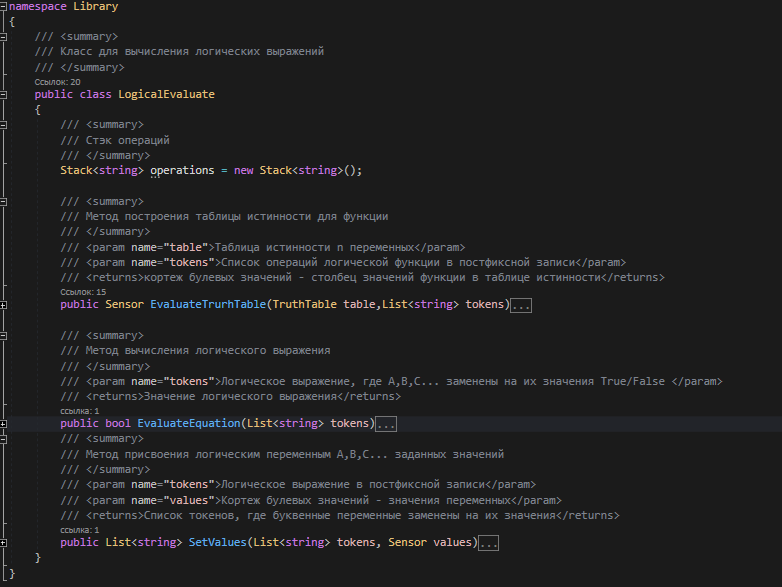
В файле BooleanSensors содержится две структуры: Sensor и TruthTable, которые представляют кортеж булевых значений и таблицу истинности соответственно. Sensor позволяет осуществлять логические операции над n логическими переменными. Основой для TruthTable служит массив типа Sensor, то есть массив кортежей булевых значений – таблицу истинности. Структура содержит методы для построения таблицы истинности n переменных и работы с ней (логические операции, получение строки или столбца по номеру).



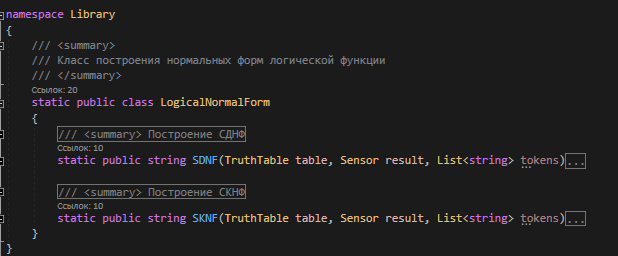
LogicalParser – Класс для перевода строки – логического выражения в постфиксную запись. Он содержит методы определения приоритета логической операции Priority(string c) и метод определения является ли строка оператором или операндом isOperator(string c). Главный метод этого класса – метод перевода логического выражения в постфиксную запись – Parse(string input).



Для вычисления значения логического выражения используется класс Logical Evaluate. В нём содержатся: метод установки булевых значений для переменных, метод расчёта выражения, в которой буквы-переменные заменены на их булевы значения и метод построения таблицы истинности для заданной функции – вычисление значения для каждой строки таблицы.

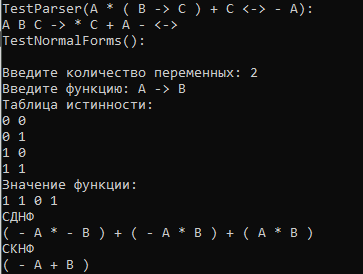


Класс LogicalNormalForm содержит методы для построения СНДФ и СКНФ по таблице истинности функции.



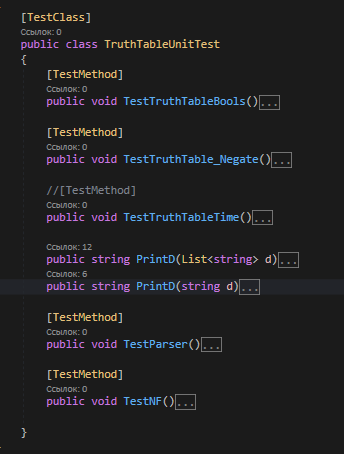
**3.3. Корректность**

Программа корректно переводит логическое выражения в постфиксную запись, строит таблицу истинности для заданной функции и нормальные формы. Действительно, импликацию можно представить как (не А или Б).



**3.4. Тестируемость**

Все методы приложения облагаются юнит-тестами и успешно их проходят



**3.5. Устойчивость**

Наше приложение также достаточно устойчива к изменениям в спецификации. Например, мы захотим вычислять значение не только логических, но и арифметических выражений. Нам необходимо будет лишь добавить новые операнды к методам Parse() и Evaluate(), чтобы программа могла работать с арифметическими выражениями.

**3.6. Стилизованность**

Код программы выдержан в одном стиле – поля классов названы в нижнем регистре, названия методов начинаются с большой буквы и написаны в стиле Camel, где несколько пишутся слитно и каждое новое с большой буквы.

**3.7. Читаемость**

Код программы обладает хорошей читаемостью. Он наполнен большим количеством комментариев, а поля и методы имеют говорящие названия. Также проект имеет прозрачную структуру и все методы идут последовательно в порядке их выполнения.

**3.8. Универсальность**

Классы нашей программы универсальны и могут быть использованы в других приложениях. Так, например, структуры Sensor и TruthTable подходят для любых видов работы с логическими функциями. Перевод логической функции в постфиксную запись может стать хорошей основой для интерпретатора выражений

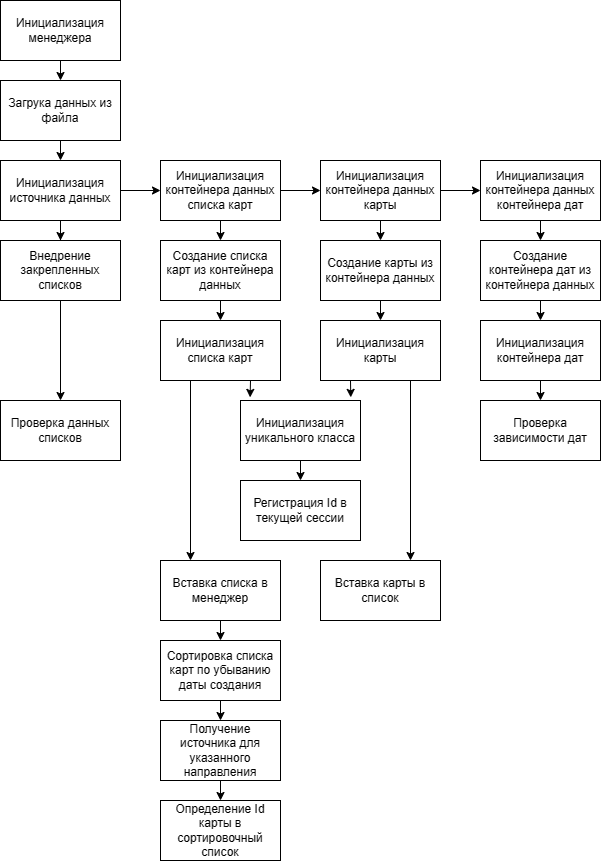
# **3.Практика качественного программирования**

(Симонов Александр)

Рассмотрим применение приёмов качественного программирования на примере проекта «Timelon» - менеджера для создания и управления задачами.

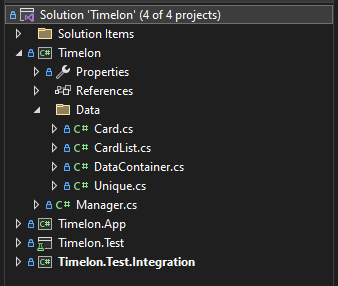
**3.1. Декомпозиция**

В примере будет рассмотрена последовательность инициализации менеджера из данных. Каждый блок – отдельная функция библиотеки. Для нашего проекта возможен следующий вариант декомпозиции:



**3.2. Модульность**

В нашем проекте роли модулей выполняют классы библиотеки.



В файле Manager.cs содержится класс Manager (менеджер), который предоставляет функционал управления списками задач и работы с данными.

// Менеджер

public sealed class Manager

{

// Константы для построения пути к файлу

public const string FileName;

public const string DirectoryName;

// Логика класса-одиночки

private static Manager \_instance;

public static Manager Instance {...}

// Путь до файла

private readonly string \_source;

// Списки карт

private readonly SortedList<int, CardList> \_list;

private readonly List<CardList> \_listEssential;

// Конструкторы

private Manager(){...}

public Manager(string source) {...}

// Доступ к полям

public string Source {...}

public string SourceDirectory {...}

public SortedList<int, CardList> All {...}

// Логика работы со списками карт

public CardList GetList(int id) {...}

public void SetList(CardList list) {...}

public bool RemoveList(int id) {...}

public bool ContainsList(int id) {...}

// Логика работы с файлом и данными

public void SaveData() {...}

private void CreateDataSource() {...}

private void WriteCardListData(List<CardListData> data) {...}

private void LoadData() {...}

private void InjectEssentials() {...}

}

Упрощенное представление файла Manager.cs

DataContainer.cs содержит в себе абстрактный класс DataContainer (контейнер данных), наследники которого могут беспрепятственно быть сериализованы в формат XML и быть сохранены или загружены из файла.

// Контейнер данных

public abstract class DataContainer

{

// Представление объекта в формате XML

public override string ToString() {...}

}

Упрощенное представление файла DataContainer.cs

В файле Unique.cs хранятся абстрактные классы Unique (уникальный класс) и UniqueData (контейнер данных уникального класса), а также интерфейс IUniqueIdentifiable. Наследники класса Unique и интерфейса IUniqueIdentifiable смогут генерировать для себя уникальный идентификатор и хранить в себе этот идентификатор и название. Также наследники смогут быть инициализированы из данных, хранящихся в UniqueData.

// Идентифицируемый вид

public interface IUniqueIdentifiable {...}

// Контейнер данных уникального класса

public abstract class UniqueData : DataContainer

{

// Поля для данных

public int Id;

public string Name;

}

// Уникальный класс

public abstract class Unique<T> where T : IUniqueIdentifiable

{

// Счетчик идентификаторов

private static int \_incrementor;

// Логика генерации идентификаторов

private static void Register(int id) {...}

private static int UniqueId() {...}

// Поля класса

private int \_id;

private string \_name;

// Конструкторы

public Unique(int id, string name) {...}

public Unique(string name) : this(UniqueId(), name) {...}

// Доступ к полям

public int Id {...}

public string Name {...}

// Логика сравнения объектов

public bool Equals(Unique<T> unique) {...}

}

Упрощенное представление файла Unique.cs

Card.cs содержит в себе класс-наследник уникального класса Card, класс-наследник уникального контейнера данных CardData, класс DateTimeContainer и класс-наследник контейнера данных DateTimeContainerData. DateTimeContainer предоставляет возможность хранения и транспортировки нескольких дат, характерных для определенной карты, логику работы с этими датами и возможность конвертации в или из DateTimeContainerData для сериализации. Card содержит в себе основные поля карты и логику конвертации данных в или из CardData.

// Контейнер данных дат

public class DateTimeContainerData : DataContainer

{

// Поля для данных

public DateTime Created;

public DateTime? Updated;

public DateTime? Planned;

}

// Контейнер дат

public class DateTimeContainer

{

// Новый контейнер с текущей датой

public static DateTimeContainer Now {...}

// Логика конвертации из контейнера данных

public static DateTimeContainer FromData(DateTimeContainerData data)

{

return new DateTimeContainer(data.Created, data.Updated, data.Planned);

}

// Поля класса

private DateTime \_created;

private DateTime? \_updated;

private DateTime? \_planned;

// Конструктор

public DateTimeContainer(DateTime created, DateTime? updated, DateTime? planned) {...}

// Логика конвертации в контейнер данных

public DateTimeContainerData ToData() {...}

// Доступ к полям

public DateTime Created {...}

public DateTime? Updated {...}

public DateTime? Planned {...}

// Логика работы с датами

public bool HasUpdated() {...}

public bool HasPlanned() {...}

}

// Контейнер данных карты

public class CardData : UniqueData

{

// Поля для данных

public DateTimeContainerData Date;

public string Description;

public bool IsImportant;

public bool IsCompleted;

}

Упрощенное представление файла Card.cs (часть 1)

// Карта

public class Card : Unique<Card>, IUniqueIdentifiable

{

// Логика конвертации из контейнера данных

public static Card FromData(CardData data) {...}

// Поля класса

private DateTimeContainer \_date;

private string \_description;

private bool \_isImportant;

private bool \_isCompleted;

// Конструкторы

public Card(int id, string name, DateTimeContainer date, string description, bool isImportant, bool isCompleted) : this(id, name, date) {...}

public Card(int id, string name, DateTimeContainer date) : base(id, name) {...}

public Card(string name, DateTimeContainer date = null) : base(name) {...}

// Логика конвертации в контейнер данных

public CardData ToData() {...}

// Доступ к полям

public DateTimeContainer Date {...}

public string Description {...}

public bool IsImportant {...}

public bool IsCompleted {...}

// Переопределение

public override string ToString() {...}

}

Упрощенное представление файла Card.cs (часть 2)

В файле CardList.cs хранятся класс-наследник уникального класса CardList, класс-наследник контейнера уникального класса CardListData и перечисление SortOrder. CardList являет собой список карт, предоставляющий место для хранения карт, логику работы с картами, различные методы поиска и сортировки карт, а также возможность конвертации данных в или из CardListData. SortOrder содержит в себе возможные статусы сортировки.

// Перечисление статусов сортировки

public enum SortOrder

{

Initial,

Unsorted,

Ascending,

Descending

}

// Контейнер данных списка карт

public class CardListData : UniqueData

{

// Поля для данных

public bool IsEssential;

public List<CardData> List;

}

Упрощенное представление файла CardList.cs (часть 1)

// Список карт

public class CardList : Unique<CardList>, IUniqueIdentifiable

{

// Логика конвертации из контейнера данных

public static CardList FromData(CardListData data) {...}

// Поле класса

private bool \_isEssential;

// Хранилища карт

private readonly Dictionary<int, Card> \_pool;

private readonly List<int> \_idListDefault;

private readonly List<int> \_idListImportant;

private readonly List<int> \_idListCompleted;

// Поле класса

private SortOrder \_status = SortOrder.Initial;

// Конструкторы

public CardList(int id, string name, bool isEssential, List<Card> list) : this(id, name, isEssential) {...}

public CardList(int id, string name, bool isEssential = false) : base(id, name) {...}

public CardList(string name) : base(name) {...}

// Логика конвертации в контейнер данных

public CardListData ToData() {...}

// Доступ к полям класса

public bool IsEssential {...}

public Dictionary<int, Card> All {...}

public SortOrder Status {...}

// Логика получения отсортированных списков

public List<Card> GetListDefault(SortOrder order = SortOrder.Descending) {...}

public List<Card> GetListImportant(SortOrder order = SortOrder.Descending) {...}

public List<Card> GetListCompleted(SortOrder order = SortOrder.Descending) {...}

// Логика поиска карт

public List<Card> SearchByContent(string content) {...}

public List<Card> SearchByDateUpdated(DateTime date) {...}

public List<Card> SearchByDateUpdated(DateTime minDate, DateTime maxDate) {...}

// Логика работы с картами

public Card Get(int id) {...}

public void Set(Card card) {...}

public bool Remove(int id) {...}

public bool Contains(int id) {...}

// Логика сортировки

private List<Card> GetListSorted(List<int> source, SortOrder order) {...}

private IEnumerable<KeyValuePair<int, Card>> Source(SortOrder order) {...}

private void Cache(Card card) {...}

private void Sort(SortOrder order) {...}

// Переопределения

public bool Equals(CardList list) {...}

public override string ToString() {...}

}

Упрощенное представление файла CardList.cs (часть 2)

**3.3. Корректность**

Программа корректно инициализирует менеджер из данных в файле, инициализируя и используя при этом все существующие классы библиотеки. Происходит вызов почти всех модулей библиотеки и в результате мы имеем готовый к работе менеджер со всеми картами и списками внутри.

Создание случайных списков карт в менеджере и работа с данными:

<CardListData Id="0">

<Name>Задачи</Name>

<IsEssential>true</IsEssential>

<List />

</CardListData>

<CardListData Id="1">

<Name>Важное</Name>

<IsEssential>true</IsEssential>

<List />

</CardListData>

<CardListData Id="35">

<Name>CDPUNOJDLYLBP</Name>

<IsEssential>false</IsEssential>

<List>

<CardData Id="332">

<Name>AQERJ</Name>

<Date>

<Created>2022-01-10T19:14:04</Created>

<Updated xsi:nil="true" />

<Planned xsi:nil="true" />

</Date>

<Description>HESVUNVLYRXZORGB</Description>

<IsImportant>true</IsImportant>

<IsCompleted>true</IsCompleted>

</CardData>

<CardData Id="333">

<Name>XFGFMKO</Name>

<Date>

<Created>2022-01-20T23:34:21</Created>

<Updated xsi:nil="true" />

<Planned xsi:nil="true" />

</Date>

<Description>EGXMPOUZWZSBNRMHNBARUPIAOD</Description>

<IsImportant>true</IsImportant>

<IsCompleted>false</IsCompleted>

</CardData>

<CardData Id="334">

<Name>FKGLEHR</Name>

<Date>

<Created>2022-10-17T03:06:46</Created>

<Updated xsi:nil="true" />

<Planned xsi:nil="true" />

</Date>

<Description>JZWHEWYUJNTYZINKWPYWYM</Description>

<IsImportant>true</IsImportant>

<IsCompleted>true</IsCompleted>

</CardData>

</List>

</CardListData>

Замер времени выполнения операций списка карт (100000):

Создание случайного списка карт: 00:00:00.7301739

Сортировка в произвольном порядке: 00:00:00.0144784

Сортировка по возрастанию: 00:00:00.0708269

Сортировка по убыванию: 00:00:00.0774236

Поиск по части названия или описания карты: 00:00:00.1010403

Поиск по дате обновления карты: 00:00:00.0121057

Часть вывода в консоль в интеграционном тестировании

**3.4. Тестируемость**

Все методы библиотеки облагаются юнит-тестами и успешно их проходят.

// Модульное тестирование класса Manager

[TestClass]

public class ManagerTest

{

// Путь до директории тестового файла

private const string SampleSource = "sample";

// Идентификаторы списков карт для тестирования

private const int ReservedIdA;

private const int ReservedIdB;

private const int ReservedBadId;

// Экземпляр тестируемого менеджера

private Manager \_manager;

// Доступ к защищенным методам и полям тестируемого менеджера

private PrivateObject \_managerAccessor;

// Списки карт для тестирования менеджера

private readonly CardList \_cardListA;

private readonly CardList \_cardListB;

// Логика возврата тестируемого менеджера к первоначальному состоянию

[TestInitialize()]

public void ResetManager() {...}

// Тест инициализации

[TestMethod]

public void TestConstructor() {...}

// Тесты доступа к полям

[TestMethod]

public void TestInstanceProperty() {...}

[TestMethod]

public void TestSourceDirectoryProperty() {...}

[TestMethod]

public void TestSourceProperty() {...}

[TestMethod]

public void TestAllProperty() {...}

// Тесты логики работы со списками карт

[TestMethod]

public void TestGetList() {...}

[TestMethod]

public void TestSetList() {...}

[TestMethod]

public void TestRemoveList() {...}

[TestMethod]

public void TestRemoveNotExistingList() {...}

[TestMethod]

public void TestContainsList() {...}

// Тесты логики работы с файлом и данными

[TestMethod]

public void TestSaveLoadDataSequence() {...}

[TestMethod]

public void TestCreateDataSource() {...}

[TestMethod]

public void TestInjectEssentials() {...}

}

Упрощенное представление файла ManagerTest.cs

**3.5. Устойчивость**

Например, мы захотим добавить в карту новую дату и работать с ней. Нам необходимо будет лишь добавить поле и логику работы с этой датой в контейнер данных и поле в контейнер данных дат. Все остальные модули библиотеки останутся без изменений.

Или более глобальное изменение: использование базы данных в качестве провайдера. Достаточно всего лишь заменить функционал работы с данными в менеджере на отдельный новый класс со всеми операциями. Контейнеры данных при этом останутся без изменений и продолжат выполнять свою работу в качестве посредников между провайдером и объектом, но уже без необходимости в XML сериализации.

**3.6. Стилизованность**

Код библиотеки выдержан в одном стиле – поля классов названы в нижнем регистре, названия методов начинаются с большой буквы и написаны в стиле CamelCase, где несколько слов пишутся слитно и каждое новое слово начинается с большой буквы.

**3.7. Читаемость**

Код библиотеки обладает хорошей читаемостью. Он наполнен большим количеством комментариев, а поля и методы имеют говорящие названия. Также проект имеет прозрачную структуру и все методы идут последовательно в порядке их выполнения.

**3.8. Универсальность**

Классы нашей библиотеки универсальны и могут быть использованы в других приложениях. Так, например, уникальный класс может быть унаследован чем угодно при необходимости иметь объект с уникальным идентификатором и автоматически генерировать его безопасным образом. Или контейнер данных, представляющий собой полностью открытый объект, позволяющий обезопасить данные основного объекта и иметь при этом возможность сериализации стандартными методами.

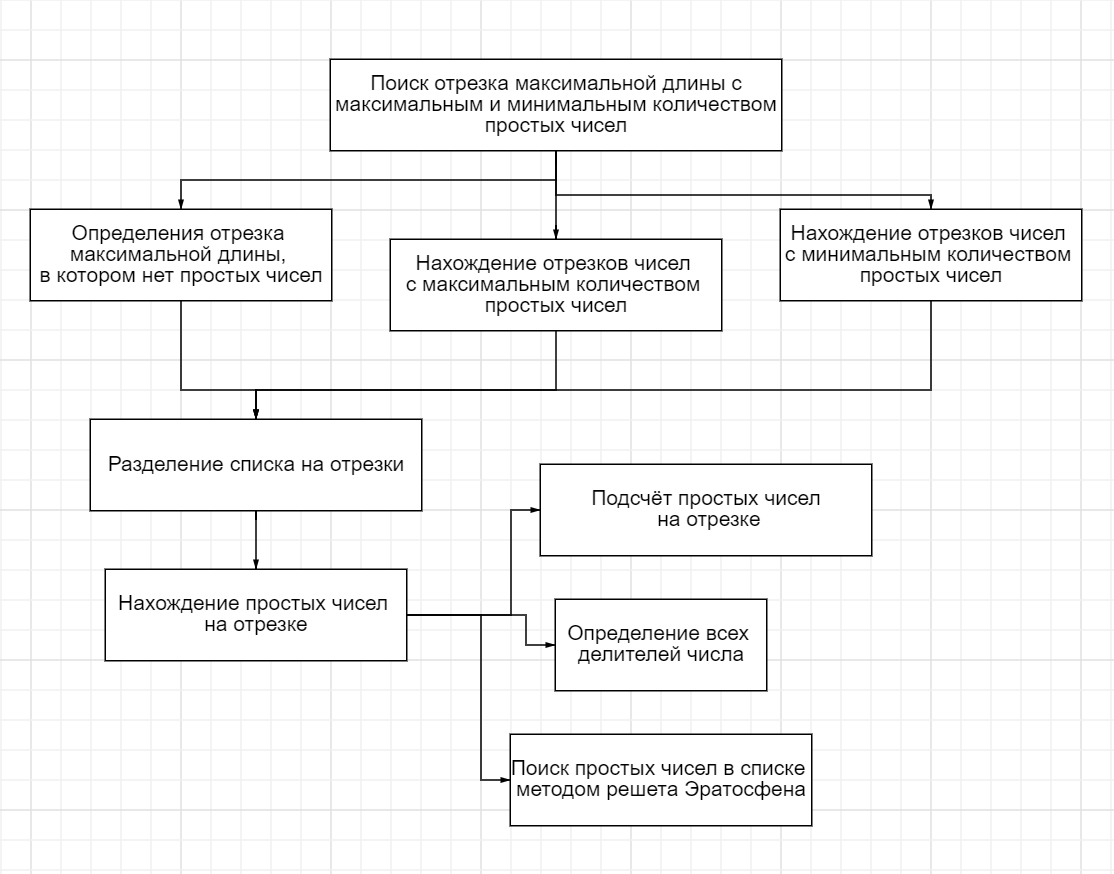
# **3.Практика качественного программирования**

(Громова Полина)

Рассмотрим применение приёмов качественного программирования на примере проекта «Простые числа», предназначенного для нахождения отрезков максимальной длины с максимальным и минимальным количеством простых чисел в заданном интервале.

**3.1. Декомпозиция**

Для нашего проекта возможен следующий вариант декомпозиции:



**3.2. Модульность**

В нашем проекте роли модулей выполняют отдельные классы Primes и ArrayHandler.

/// <summary>

/// Класс обработки массива.

/// Содержит функции заполнения массива и нахождения максимальных десятков

/// с максимальным и минимальным количеством простых чисел

/// </summary>

public class ArrayHandler

{

/// <summary>

/// Функция заполнения списка целыми числами от 1 до N

/// </summary>

/// <param name="N">Входное число, последнее в списке</param>

/// <returns>Список целых чисел от 1 до N</returns>

public static List<int> ArrayFiller(int N)

{

List<int> array = new List<int>();

//Формирование массива целых чисел 1..N

for (int i = 1; i <= N; i++)

{

array.Add(i);

}

array[0] = 0;

return array;

}

/// <summary>

/// Функция подсчёта простых чисел на сегменте

/// </summary>

/// <param name="Primes">Список целых чисел, в котором непростые заменены на ноль</param>

/// <param name="Segment">рассматриваемый сегмент чисел</param>

/// <returns>Список, индексы которого обозначают номер сегмента,

/// а значение - количество простых чисел на нём.</returns>

public static List<int> CountPrimesOnSegment(List<int> Primes,int Segment)

{

List<int> Answer = new List<int>() {0}; //Индекс - номер сегмента, значение - количество простых на сегменте

int index = 0;

for (int i = 0; i < Primes.Count; i++)

{

if (Primes[i] != 0)

Answer[index]++;

if ((i+1) % Segment == 0)

{

index++;

Answer.Add(0);

}

}

return Answer;

}

/// <summary>

/// Функция определения отрезка максимальной длины, в котором нет простых чисел

/// </summary>

/// <param name="NewArray"> Список целых чисел, в котором непростые числа заменены на ноль</param>

/// <returns>Список, содержащий информацию об отрезке

/// [0 Первое число отрезка

/// 1 Последнее число отрезка

/// 2 Количество чисел между первым и последним числом отрезка]

/// </returns>

public static List<int> Segment(List<int> NewArray)

{

//sigma[первое простое число, второе простое число, длина отрезка]

List<int> sigma = new List<int> { 0, 0, 0 };

int CurLen, FirstNumIndex=0;

//Считаем расстояние между ненулевыми элементами списка простых чисел

for (int i = 0; i < NewArray.Count; i++)

if (NewArray[i] != 0)

{

CurLen = i - FirstNumIndex;

if (CurLen > sigma[2]) { sigma[2] = CurLen-1; sigma[0]= FirstNumIndex + 1; sigma[1] = i+1; }

FirstNumIndex = i;

}

return sigma;

}

/// <summary>

/// Функция нахождения максимальных отрезков чисел

/// с максимальным и минимальным количеством простых чисел

/// </summary>

///

/// <param name="array">Список простых чисел,

/// в котором все непростые числа заменены на 0</param>

/// <param name="segment">рассматриваемый отрезок чисел</param>

///

/// <returns>

/// Список, содержащий информацию о максимальных

/// отрезках с максимальным и минимальным количеством простых чисел

///

/// [0 максимальный сегмент с минимальным количеством простых чисел,

/// 1 минимальное количество простых чисел,

/// 2 минимальный сегмент с максимальным количеством простых чисел,

/// 3 максимальное количество простых чисел]

/// </returns>

public static List<int> SplitPrimes(List<int> array, int segment)

{

//Решение проблемы работы с некратным количеством сегментов, умещающихся в N

if (array.Count() % segment == segment - 1 && segment!=1) array.Add(array.Count() + 1);

//minmax[segment с минимумом, минимум, segment с максимумом,максимум]

List<int> minmax = new List<int> { 0, segment, 0, 0 };

int counter = 0; //Счётчик простых чисел в segment

//Проходим по массиву, и каждые segment шагов

//сравниваем counter c максимумом и минимумом

for (int i = 0; i < array.Count; i++)

{

if ((i + 1) % segment == 0)

{

//Проверка на максимум

if (counter >= minmax[3])

{

minmax[2] = i / segment;

minmax[3] = counter;

}

//Проверка на минимум

if (counter <= minmax[1])

{

minmax[0] = i / segment;

minmax[1] = counter;

}

//Обнуление счётчика для нового segment

counter = 0;

}

//Подсчёт простого числа

if (array[i] != 0) counter++;

}

return minmax;

}

}

/// <summary>

/// Класс работы с простыми числами.

/// Содержит функцию поиска простых чисел в списке методом решета Эратосфена

/// </summary>

public class Primes

{

//Список для хранения простых чисел

static List<int> PrimeArray = new List<int>();

/// <summary>

/// Функция определения всех делителей числа N

/// </summary>

/// <param name="N">исходной число</param>

/// <returns>список всех делителей числа N</returns>

public static List<int> GetDividers(int N)

{

List<int> Dividers = new List<int>();

//Ищем делители до квадратного корня

for (int i = 1; i < Math.Sqrt(N); i++)

if (N % i == 0) Dividers.Add(i);

//Если N-квадрат целого числа, добавляем это число к делителям

bool flag = true;

if (Math.Sqrt(N) == (int)Math.Sqrt(N))

{

flag = false;

Dividers.Add((int)Math.Sqrt(N));

}

//Добавляем "зеркальные" относительно корня делители

//В случае, если N-квадрат целого числа, не берём последнее число в Dividers

for (int i = flag?Dividers.Count-1: Dividers.Count-2; i >= 0; i--)

Dividers.Add(N / Dividers[i]);

return Dividers;

}

/// <summary>

/// Функция поиска простых чисел в списке методом решета Эратосфена

/// </summary>

///

/// <param name="InputArray">Список целых чисел от 1 до N</param>

///

/// <returns>

/// Список целых чисел от 1 до N,

/// в котором все непростые числа заменены на 0

/// </returns>

public static List<int> doEratosfen(List<int> InputArray)

{

//Перенос входного списка в объект класса

PrimeArray = InputArray;

//Поиск простых чисел решетом Эратосфена

EratosfenHandleArray();

return PrimeArray;

}

/// <summary>

/// Функция обработки списка целых чисел решетом Эратосфена

/// </summary>

private static void EratosfenHandleArray()

{

int i = 1;

//Для каждого ненулевого элемента списка

//выполняем "выкалывание" по алгоритму решета Эратосфена

while (i < PrimeArray.Count())

{

if (PrimeArray[i] != 0) Step(PrimeArray[i], i);

i++;

}

}

/// <summary>

/// Функция, реализующая "выкалывание" чисел по алгоритму решета Эратосфена

/// </summary>

/// <param name="prime">Стартовое простое число</param>

/// <param name="start">индекс стартового числа в списке</param>

private static void Step(int prime, int start)

{

//Зануляем элементы списка с шагом в значение текущего стартового простого числа

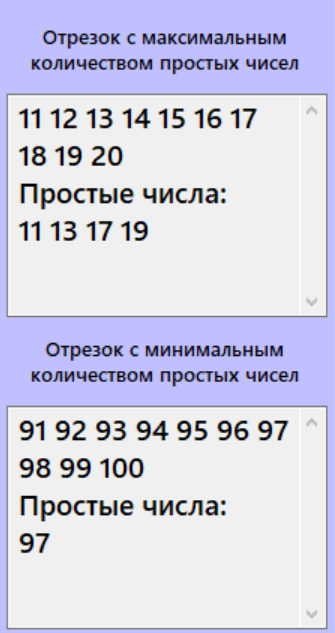
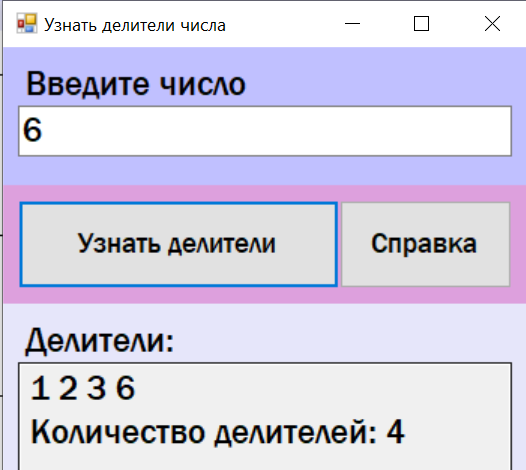
for (int i = start + prime; i < PrimeArray.Count(); i += prime) PrimeArray[i] = 0;

}

}

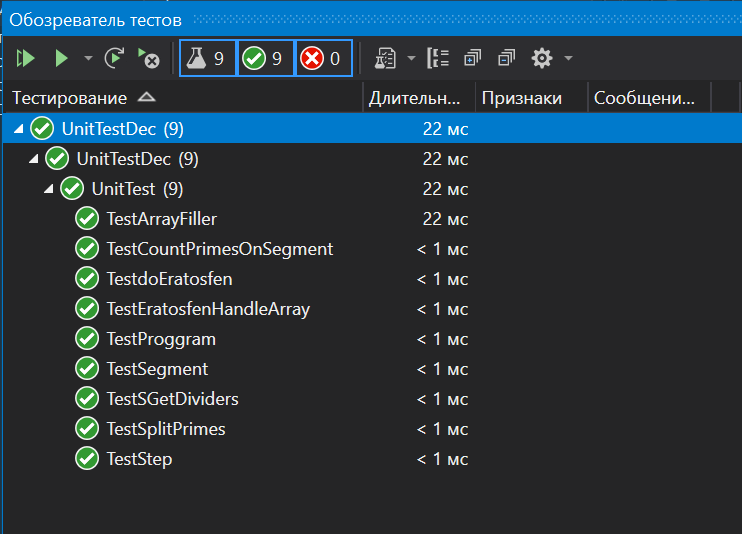
**3.3. Корректность**

Программа корректно производит поиск отрезков максимальной длины с максимальным и минимальным количеством простых чисел. Также исправно подсчитывает все делители числа.

**3.4. Тестируемость**

Все функции классов облагаются юнит-тестами и успешно их проходят.



**3.5. Устойчивость**

Наше приложение также достаточно устойчиво к изменениям. При желании реализовать другие функции связанные с простыми числами, это будет нетрудно реализовать.

**3.6. Стилизованность**

Код программы выдержан в одном стиле – поля классов названы в нижнем регистре, названия методов начинаются с большой буквы и написаны в стиле Camel, где несколько пишутся слитно и каждое новое с большой буквы.

**3.7. Читаемость**

Код программы обладает хорошей читаемостью. Он наполнен достаточным количеством комментариев, а поля и методы имеют говорящие названия. Также проект имеет прозрачную структуру, и все методы идут последовательно в порядке их выполнения.

**3.8. Универсальность**

Классы нашей программы универсальны и могут быть использованы в других приложениях. Так как функции класса Primes могут использоваться для работы с простыми числами, данный класс может быть востребован для других задач, отличающихся от поставленной задачи.

# **4.Вывод**

Благодаря правилам качественного программирования, разработчикам программного обеспечения удаётся успешно проектировать и предоставлять пользователям действительно качественный продукт, который удовлетворяет потребностям не только крупных компаний и обычных пользователей, для которых важно быстродействие и удобство использования, но и таких же программистов, которые будут в дальнейшем заниматься развитием и поддержкой созданного приложения, для чего необходима точная организация и прозрачная архитектура проекта.