ГОСУДАРСТВЕННЫЙ КОМИТЕТ СВЯЗИ, ИНФОРМАТИЗАЦИИ

И ТЕЛЕКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАРШИНСКИЙ ФИЛИАЛ

“Допущен к защите”

Зав. кафедрой “Информационных

технологий”_____Носиров Б.Н.

«_____» _____ 2014 год

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

на тему

«ЭВОЛЮЦИЯ ТЕХНОЛОГИЙ И ЯЗЫКОВ ПРОГРАММИРОВАНИЯ»

Выпускник	_____	Тен С.Н.
Руководитель	_____	доц. Узаков З.У.
Рецензент	_____	Эргашев А.Х.
Консультант по БЖД	_____	доц. О.Д. Рахимов

Карши 2014 г.

ГОСУДАРСТВЕННЫЙ КОМИТЕТ СВЯЗИ, ИНФОРМАТИЗАЦИИ

И ТЕЛЕКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ РЕСПУБЛИКИ УЗБЕКИСТАН

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАРШИНСКИЙ ФИЛИАЛ

ФАКУЛЬТЕТ КОМПЬЮТЕРНОГО ИНЖИНИРИНГА

Направление «5521900 - Информатика и информационные технологии»

«Утверждаю»

Заведующий кафедрой:

_____ *Б.Н.Носиров*

«__»_____2014 год

ЗАДАНИЕ

На выпускную квалификационную работу

Студента *Тен Сергея Николаевича*

1. Тема выпускной квалификационной работы:

Эволюция технологий и языков программирования

2. Утверждена приказом № 37-15, 16 января 2014г. Каршинского филиала ТУИТ
3. Срок сдачи выпускной квалификационной работы «__»_____*2014г*
4. Исходные данные к работе: *Учебно - методические материалы по программированию,Электронные учебники,.*
5. Содержание расчётно-пояснительной записки (список подлежащих рассмотрению работ): *Глава 1 Введение; Глава 2 Исторически этапы*

развития технологий и языков программирования; Глава 3 Объектно – ориентировано технологии и языков программирования; Глава 4 Реализация основных объектов и концепций объектно – ориентированной технологии программирования на языке С++; Глава 5 Безопасность жизнедеятельности; Глава 6 Заключение; Глава 7 Список литературы; Глава 8 Приложение

6. Перечень графического материала, алгоритмов, программ, блок-схем, функциональных схем (подлежащие обязательному выполнению): *Интерфейс клиентской части приложения Алгоритм и принципы работы приложения,скрин-шоты, исходный код.*

7. Консультант по квалификационной работе:

№	Наименование главы	Консультант	Дата принятия задания	Подпись консультанта
1	*Эволюция технологий и языков программирования*	Узаков З.У.	25.01.2014 г.	
2	*Безопасность жизнедеятельности*	Рахимов О.Д.	25.01.2014 г.	

8. Календарный график по выполнению выпускной квалификационной работы

№	Разделы квалификационной работы	Объём квалификационной работы, (стр.)	Отношение к общему объёму, %	Отметка о выполнении задания	Примечание
1	Глава1 Введение	7	6,0	Выполнено	

2	Глава2 Исторически этапы развития технологий и языков программирования	1	0,9	Выполнено	
3	2.1 Технологии и языки программирования низкого уровня Технологии и языки программирования низкого уровня	4	3,4	Выполнено	
4	2.2 Технологии и языки программирования высокого уровня	10	8,9	Выполнено	
5	2.3 Технологии и языки программирования сверх высого уровня	8	6,9	Выполнено	
6	Глава3 Объектно – ориентировано технологии и языков программирования	1	0,9	Выполнено	
7	3.1 Языки объектно – ориентированного программирования	10	8,7	Выполнено	
8	3.2 Основные концепции объектно – ориентированной технологии программирования	3	2,6	Выполнено	
9	3.3 Особенности объектно – ориентированной технологии	14	2,3	Выполнено	

	программирования				
10	Глава4 Реализация основных объектов и концепций объектно – ориентированной технологии программирования на языке С++	2	1,7	Выполнено	
11	4.1 Реализация массивов, структур, объединений и классов	3	2,6	Выполнено	
12	4.2 Реализация концепции полиморфизма	8	7,0	Выполнено	
13	4.2 Реализация концепций наследования	9	7,8	Выполнено	
14	4.3 Реализация концепций инкапсуляции	12	10,5	Выполнено	
15	Глава5 Безопасность жизнедеятельности	1	0,9	Выполнено	
16	5.1 Характеристика условий труда программиста	1	0,9	Выполнено	
17	5.2 Параметры микроклимата	1	0,9	Выполнено	
18	5.3 Шум и вибрация	1	0,9	Выполнено	
19	5.4 Требования к рабочему	1	0,9	Выполнено	

	месту				
20	5.5 Противопожарная безопасность	1	0,9	Выполнено	
21	Глава6 Заключение	3	2,6	Выполнено	
22	Глава8 Приложение	3	2,6	Выполнено	

Руководитель работы: _____ Узаков З.У.

(подпись) (Ф.И.О.)

Дата принятия задания: 25 января 2014 года

(дата)

Студент - выпускник: _____ Тен С.Н.

(подпись) (Ф.И.О.)

Содержание

ВВЕДЕНИЕ

По мере усиления зависимости общества от вычислительных систем надежность и гибкость последних приобретает все большее и большее значение. Одним из последствий этой зависимости является растущее беспокойство о высоком качестве программного обеспечения и возможности его быстрой разработки, реализации, модификации. Используемый в некотором проекте язык программирования в значительной степени определяет скорость разработки и реализации, простоту сопровождения, возможность переноса создаваемого в рамках этого проекта программного обеспечения.

В этой Выпускной квалификационной работе предложена методология сравнения языков программирования. Выпускная квалификационная работа построена на базе некоторого перечня вопросов, лежащих в основе сравнения и оценки языков программирования. Этот перечень вопросов был сформирован, исходя из посылки его последующего применения для сравнения и оценки процедурных языков программирования, таких, например, как языки Ада, Си, Паскаль. С каждым из включенных в этот перечень вопросов ассоциирован ряд конкретных подразделы, ответы на которые необходимы для формирования характеристики некоторого языка программирования. Кроме того, каждый из включенных в этот перечень вопросов сопровождается дополнительной информацией, представляющей собой критерии, которые могут быть использованы для оценки языка программирования на основе уже сформированной его характеристики.

Программирование - это искусство создавать программные продукты, которые написаны на языке программирования. Прогресс компьютерных технологий определил процесс появления новых разнообразных знаковых систем для записи алгоритмов – языков программирования. Смысл

появления такого языка – оснащенный набор вычислительных формул дополнительной информации, превращает данный набор в алгоритм.

Язык программирования Си (английское название — C) создавался как инструментальный язык для разработки операционных систем, трансляторов, баз данных и других системных и прикладных программ. Так же как и Паскаль, Си — это язык структурного программирования, но, в отличие от Паскаля, в нем заложены возможности непосредственного обращения к некоторым машинным командам, к определенным участкам памяти компьютера. Дальнейшее развитие Си привело к созданию языка объектно-ориентированного программирования Си++.

ЭВМ будущего, пятого поколения называют машинами «искусственного интеллекта». Но прототипы языков для этих машин были созданы существенно раньше их физического появления. Это языки ЛИСП и Пролог.

Процесс работы компьютера заключается в выполнении программы, то есть набора вполне определённых команд во вполне определённом порядке. Машинный вид команды, состоящий из нулей и единиц, указывает, какое именно действие должен выполнить центральный процессор. Чтобы задать компьютеру последовательность действий, которые он должен выполнить, нужно задать последовательность двоичных кодов соответствующих команд. Программы в машинных кодах состоят из тысячи команд. Написание таких программ – занятие сложное и утомительное. Программист должен помнить комбинацию нулей и единиц двоичного кода каждой команды, а также двоичные коды адресов данных, используемых при её выполнении. Гораздо проще написать программу на каком-нибудь языке, более близком к естественному человеческому языку, а работу по переводу этой программы в машинные коды поручить компьютеру. Так возникли языки, предназначенные специально для написания программ.

Язык программирования – формальная знаковая система, предназначенная для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и семантических правил, задающих внешний вид программы и действия, которые выполнит исполнитель (компьютер) под ее управлением.

Создатели языков по–разному толкуют понятие язык программирования. К наиболее распространенным утверждениям, признаваемым большинством разработчиков, относятся следующие:

– Функция: язык программирования предназначен для написания компьютерных программ, которые применяются для передачи компьютеру инструкций по выполнению того или иного вычислительного процесса и организации управления отдельными процессами.

– Задача: язык программирования отличается от естественных языков тем, что предназначен для передачи команд и данных от человека компьютеру, в то время как естественные языки используются для общения людей между собой. В принципе, можно обобщить определение «языков программирования» – это способ передачи команд, приказов, четкого руководства к действию, тогда как человеческие языки служат только для обмена информацией.

– Исполнение: язык программирования может использовать специальные конструкции для определения и манипуляции структурами данных и управления процессом вычислений.

Язык программирования чаще всего представлен в виде набора спецификаций, определяющих его синтаксис и семантику. Для многих языков программирования созданы международные стандарты. Специальные организации проводят регулярное обновление спецификаций и формальных определений соответствующего языка, а также продолжают разработку и модернизацию языков программирования.

Со времени создания первых программируемых машин человечество придумало уже более двух с половиной тысяч языков программирования и каждый год их число пополняется новыми. Некоторыми языками умеет пользоваться только небольшое число их собственных разработчиков, другие становятся известны миллионам людей. Профессиональные программисты иногда применяют в своей работе более десятка разнообразных языков программирования.

Язык программирования служит двум связанным между собой целям: он дает программисту аппарат для задания действий, которые должны быть выполнены, и формирует концепции, которыми пользуется программист, размышляя о том, что делать. Первой цели идеально отвечает язык, который настолько "близок к машине", что всеми основными машинными аспектами можно легко и просто оперировать достаточно очевидным для программиста образом. Второй цели идеально отвечает язык, который настолько "близок к решаемой задаче", чтобы концепции ее решения можно было выражать прямо и коротко.

Может показаться удивительным, но конкретный компьютер способен работать с программами, написанными на его родном машинном языке. Существует почти столько же разных машинных языков, сколько и компьютеров, но все они суть разновидности одной идей простые операции производятся со скоростью молнии на двоичных числах.

Можно писать программы непосредственно на машинном языке, хотя это и сложно. На заре компьютеризации(в начале 1950-х г.г.), машинный язык был единственным языком, большего человек к тому времени не придумал. Для спасения программистов от сурового машинного языка программирования, были созданы языки высокого уровня (т.е. немашинные языки), которые стали своеобразным связующим мостом между человеком и машинным языком компьютера. Языки высокого уровня работают через трансляционные программы, которые вводят "исходный код" (гибрид

английских слов и математических выражений, который считывает машина), и в конечном итоге заставляют компьютер выполнять соответствующие команды, которые даются на машинном языке.

Несмотря на то, что в различных источниках делается акцент на те или иные особенности внедрения и применения Объекто-ориентированного программирования, 3 основных (базовых) понятия ООП остаются неизменными. К ним относятся:

1. Наследование (Inheritance)

Наследованием называется возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (прародителя, иногда его называют суперклассом) и добавляя, при необходимости, новые свойства и методы. Набор классов, связанных отношением наследования, называют иерархией. Наследование призвано отобразить такое свойство реального мира, как иерархичность.

2. Инкапсуляция (Encapsulation)

Инкапсуляция — это принцип, согласно которому любой класс должен рассматриваться как чёрный ящик — пользователь класса должен видеть и использовать только интерфейсную часть класса (т. е. список декларируемых свойств и методов класса) и не вникать в его внутреннюю реализацию. Поэтому данные принято инкапсулировать в классе таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а с помощью методов. Принцип инкапсуляции (теоретически) позволяет минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов.

3. Полиморфизм (Polymorphism)

Полиморфизмом называют явление, при котором функции (методу) с одним и тем же именем соответствует разный программный код (полиморфный код)

в зависимости от того, объект какого класса используется при вызове данного метода. Полиморфизм обеспечивается тем, что в классе-потомке изменяют реализацию метода класса-предка с обязательным сохранением сигнатуры метода. Это обеспечивает сохранение неизменным интерфейса класса-предка и позволяет осуществить связывание имени метода в коде с разными классами — из объекта какого класса осуществляется вызов, из того класса и берётся метод с данным именем. Такой механизм называется динамическим (или поздним) связыванием — в отличие от статического (раннего) связывания, осуществляемого на этапе компиляции.

Повышение производительности компьютеров и перемены в составе используемого ПО делают роль языков описания сценариев в создании приложении будущего все более и более важной. Эти языки отличаются от языков программирования системного уровня тем, что их основное назначение – связывать различные компоненты и приложения друг с другом. В них находят применение бестиповые подходы к описанию данных, что позволяет вывести программирование на более высокий уровень и ускорить процесс разработки по сравнению с языками системного уровня.

За прошедшие 15 лет в методологии написания программ для компьютеров произошла радикальная перемена. Она состоит в том, что разработчики перешли от языков программирования системного уровня, таких как C и C++, к языкам описания сценариев, примерами которых могут служить Perl Tcl. Хотя в эту перемену оказалось вовлечено огромное количество людей, лишь немногие из них осознают, что в действительности происходит, и еще меньше найдется таких, кто бы смог объяснить причины.

Объектно-ориентированное программирование - это подход к разработке программного обеспечения, основанный на объектах, а не на процедурах. Этот подход позволяет максимизировать принципы модульности и "сокрытия информации". Объектно-ориентированное

программирование базируется на связывании или инкапсуляции структур данных и процедуры, которая работает с данными в структуре, с модулем.

Актуальность выбранной темы обусловлена тем, что объектно-ориентированный принцип разработки дает много преимуществ и используется многими разработчиками. Например, каждый объект инкапсулирует его структуру данных с процедурой, используемой для работы с экземплярами структуры данных. Это позволяет устранить в коде программы внутренние зависимости, которые могут быстро привести к тому, что этот код будет трудно обслуживать. Объекты могут также наследовать из по рождающего объекта структуры данных и другие характеристики, что позволяет сэкономить усилия и обеспечить прозрачное использование для многих целей больших фрагментов кода.

Данная работа состоит из глав, в которых описана актуальность, цели и причины написания данной работы. Основное внимание я уделил 2,3,4 и 7 главе, где описывается суть моей выпускной квалификационной работе

Глава 2 содержит исторически этапы развития технологий и языков программирования в которых описывается различие между группами технологий языков низкого, высокого и сверх высокого уровней.

Глава 3 содержит описание объектно – ориентированных технологий и языков программирования в которых описывается актуальность ООТЯП

Глава 4 содержит реализацию основных концепций объектно – ориентированных технологий и языков программирования в которых реализуется концепция полиморфизма, наследования и инкапсуляции.

Глава 7 содержит наглядный пример реализации концепций полиформизма, наследования и инкапсуляции.

Глава 1

Исторически этапы развития технологий и языков программирования

Можно сформулировать ряд требований к языкам программирования и классифицировать языки по их особенностям.

Основные требования, предъявляемые к языкам программирования:

наглядность - использование в языке по возможности уже существующих символов, хорошо известных и понятных как программистам, так и пользователям ЭВМ;

единство - использование одних и тех же символов для обозначения одних и тех же или родственных понятий в разных частях алгоритма. Количество этих символов должно быть по возможности минимальным;

гибкость - возможность относительно удобного, несложного описания распространенных приемов математических вычислений с помощью имеющегося в языке ограниченного набора изобразительных средств;

модульность - возможность описания сложных алгоритмов в виде совокупности простых модулей, которые могут быть составлены отдельно и использованы в различных сложных алгоритмах;

однозначность - недвусмысленность записи любого алгоритма. Отсутствие ее могло бы привести к неправильным ответам при решении задач.

В настоящее время в мире существует несколько сотен реально используемых языков программирования. Для каждого есть своя область применения.

По наиболее распространенной классификации все языки программирования делят на языки:

- низкого уровня

- высокого уровня
- сверхвысокого уровня

1.1 Технологии и языки программирования низкого уровня

В группу языков низкого уровня входят машинные языки и языки символического кодирования: Низкоуровневый язык программирования близкий к программированию непосредственно в машинных кодах используемого реального или виртуального (например, Java, Microsoft.NET) процессора. Для обозначения машинных команд обычно применяется мнемоническое обозначение. Это позволяет запоминать команды не в виде последовательности двоичных нулей и единиц, а в виде осмысленных сокращений слов человеческого языка (обычно английских).

Иногда одно мнемоническое обозначение соответствует целой группе машинных команд, выполняющих одинаковое действие над разными ячейками памяти процессора. Кроме машинных команд языки программирования низкого уровня могут предоставлять дополнительные возможности, такие как макроопределения (макросы). При помощи директив есть возможность управлять процессом трансляции машинных кодов, предоставляя возможность заносить константы и литеральные строки, резервировать память под переменные и размещать исполняемый код по определенным адресам. Часто эти языки позволяют работать вместо конкретных ячеек памяти с переменными.

Как правило, использует особенности конкретного семейства процессоров. Общеизвестный пример низкоуровневых языков программирования:

Язык Ассемблера – это символическое представление машинного языка. Он облегчает процесс программирования по сравнению с программированием в машинных кодах.

Программисту не обязательно употреблять настоящие адреса ячеек памяти с размещенными в них данными, участвующими в операции, и вычисляемые результаты, а также адреса тех команд, к которым программа не обращается.

Некоторые задачи, например, обмен с нестандартными устройствами обработки данных сложных структур невозможно решить с помощью языков программирования высокого уровня. Это под силу ассемблеру.

В принципе, язык Ассемблер является машинным языком. И программист реализующий какую-либо задачу на языках высокого уровня, с помощью Ассемблера может определить осмыслено ли решение данной задачи, с точки зрения использования ЭВМ.

Умея разобраться в распечатке языка ассемблера, дает возможность облегчить поиск ошибок в программах, т.к. некоторые языки являются компиляторами Более того, для одного и того же процессора существует несколько видов языка ассемблера. Они совпадают в машинных командах, но различаются набором дополнительных функций (директив и макросов).

Лисп - Один из самых старых языков программирования Фортран был создан в 50-х гг. нашего века. Фортран и подобные ему языки программирования (Алгол, ПЛ/1) предназначались для решения вычислительных задач, возникающих в математике, физике, инженерных расчетах, экономике и т.п. Эти языки в основном работают с числами.

Второй старейший язык программирования Лисп (List Information Symbol Processing), Дж. Маккарти в 1962 г. скорее для работы со строками символов, нежели для работы с числами. Это особое предназначение Лиспа открыло для программистов новую область деятельности, известную ныне, как «искусственный интеллект». В настоящее время Лисп успешно применяется в экспертных системах, системах аналитических вычислений и т.п.

Обширность области возможных приложений Лиспа вызвала появление множества различных диалектов Лиспа. Это легко объяснимо: применение Лиспа для понимания естественного языка требует определенного набора базисных функций, отличных, например, от используемого в задачах медицинской диагностики.

Существование множества различных диалектов Лиспа привело к созданию в начале 80-х гг. Common LISP Комитета, который должен был выбрать наиболее подходящий диалект Лиспа и предложить его в качестве основного. Этот диалект, выбранный Комитетом в 1985г., получил название Common LISP . В дальнейшем он был принят в университетах США, а также многими разработчиками систем искусственного интеллекта, в качестве основного диалекта языка Лисп.

Язык программирования Лисп существенно отличается от других языков программирования, таких, как Паскаль, Си и т.п. Работа с символами и работа с числами как с основными элементами требует разных способов мышления.

Первоначально Лисп был задуман как теоретическое средство для рекурсивных построений, а сегодня он превратился в мощное средство, обеспечивающее программиста разнообразной поддержкой, позволяющей ему быстро строить прототипы весьма и весьма серьезных систем.

Профессор Массачусетского технологического института Дж. Самман заметил, что математическая ясность и предельная четкостьЛиспа – это еще не все. Главное – Лисп позволяет сформулировать и запомнить «идиомы», столь характерные для проектов по искусственному интеллекту.

Низкий уровень подразумевает не качество программ, а уровень детализации инструкций. Так, команда, записанная на Ассемблере, например, MOV AL, OOh, означает, что в регистр процессора (указывается, в какой именно!) надо занести число 0.

Для большинства программ такой подробный и детализированный способ указаний не нужен. И, хотя, в конечном итоге, все исполняемые программы содержат именно такие подробные инструкции, да еще в числовом представлении, человеку удобнее записывать команды более общего плана, на языке, более похожем на человеческий.

Методы программирования для старых компьютеров были громоздкими, медленными и крайне ограниченными. Эти компьютеры программировались путем установки ряда переключателей (включено или выключено), каждый переключатель представлял одну двоичную единицу (или бит), принимая значение 0 или 1. Это сильно ограничивало возможности и занимало много времени.

Следующим этапом было создание хранимых программ. Принцип был таким же – использовался двоичный машинный код, но информация хранилась в памяти компьютера на магнитных сердечниках. После этого был сделан маленький шаг по введению кода в виде, лучше поддающемся управлению: в виде шестнадцатеричных чисел (основание 16), в котором каждый разряд представлялся четырьмя битами. Этот тип программирования был еще подвержен ошибкам, но дела улучшились с появлением ассемблера. Ассемблер позволял записывать программы с помощью мнемонических сокращений, которые представляли команды в виде, более удобном для запоминания: например, ADD представляет код команды сложения двух чисел. Ассемблер использует мнемонические обозначения и преобразует их более или менее один к одному в двоичный код. Преимущество программирования на языке Ассемблер в том, по крайней мере теоретически, что он создает наиболее быстрые и эффективные программы, так как в нем существует прямая связь между кодом программы и конечным машинным кодом. Программирование на языке Ассемблер до сих пор используется для тех приложений, когда важно уменьшить время выполнения программы, а

современные варианты ассемблера даже позволяют использовать объектно-ориентированные конструкции.

К языкам низкого уровня относят:

- машинные языки – языки кодов ЭВМ;
- машино - ориентированные языки – ассемблеры, мнемокоды.

Также к языкам низкого уровня условно можно причислить MSIL, применяемый в платформе Microsoft .NET, Форт, Java байт-код.

1.2 Технологии и языки программирования высокого уровня

ALGOL это язык программирования**,** пер**в**оначально разработанный в 1958 году, который был назван по названию "АЛГОритмический процесс" основной проблемы программирования. Это краткое имя алгоритмического языка. В некотором смысле, это была реакция на Fortran, этот язык был предшественником Паскаля. Он использует слова в качестве ограничителей блоков и был первым, кто использовал пару "begin" и "end". Программирование для начинающих. Существовали три основные версии официального Алгола: Алгол 58, Алгол 60 и Алгол 68. Из них, **язык программирования Алгол 60** был самым влиятельным (Алгол 60 был предком Algol W, который был использован Никлаусом Виртом для разработки Паскаля). Каждая официальная версия Алгола названа в честь года, в котором она была опубликована. Программирование для чайников. Алгол был разработан совместно с Комитетом европейских и американских компьютерных ученых. Этот язык имеет, по крайней мере, три различных синтаксиса: ссылочный синтаксис, синтаксис публикации, и синтаксис исполнения. Для разного синтаксиса разрешено использовать различные имена ключевых слов, разрешено использовать разные разделители целой и дробной части (запятую или точку) для различных стран как основу

программирования.

Язык программирования Алгол **60 б**ыл выпущен в 1960 году Джоном Бэкусом и Питером Наурос, которые служили в комитете, который разработал АЛГОЛ. Algol 60 вдохновил разработчиков многих языков, которые создавались позже; цитата в этой связи Э. Хоара "Алгол оказал громадное влияние на своих преемников". Полная цитата "Этот язык настолько опередил свое время, что намного опередил своих предшественников и очень близок к своим последователям", но афоризм более известен. Иногда ошибочно эти слова приписывают Эдсгеру Дейкстре, что вполне понятно, так как он служил в комитете-разработчике, а также делал не менее известные замечания по поводу языка.

Компьютер B5000 компании Burroughs Corporation и его преемники были и остаются вычислительными машины со стеком данных, предназначенные для программирования на расширенном Алголе; действительно операционные системы, или MCP (Master Control Program), как их называют, написано на расширенном Алголе еще в 1961 году. Unisys Corporation до сих пор предлагает схожие с B5000 компьютеры, на которых и сегодня работает MCP и поддерживает несколько компиляторов для расширенного Алгола.

В официальном языке программирования Algol 60 не были ределены средства ввода / вывода; конкретные реализации языка обязательно должны были добавить какие-то свои, но они варьировались от одной реализации к другой. Алгол-68, возможно, преодолел этот недостаток отсутствия объектов ввода / вывода (или "transput").

Algol 60 разрешал два типа передачи параметров: вызов по значению, и инновационной вызов по имени, от которого было впоследствии отказались языки-преемники. Вызов по имени имел определенные ограничения в отличие от вызова по ссылке, что делает его нежелательной особенностью языкового дизайна. Например, в Алгол 60 невозможно создать процедуру, которая будет менять значения двух параметров, если в качестве фактических параметров, которые передаются будут выступать

целочисленная переменная и массив, индексированный по этой же целочисленной переменной.

Джон Бэкус разработал Бэкуса метод нормальной формы описания языков программирования, специально для Алгола 58. Он был пересмотрен и расширен Питером Науром в метод Бэкуса-Наура для Алгола 60. Алгол 68 был определен с использованием двухуровневой грамматики, котруаю изобрел Адриан ван Вейнгаардена и которая носит его имя.

Фортрана (FORTRAN) — один из первых языков программирования, языков-долгожителей, широко применяемый и сегодня, несмотря на критику и мрачные прогнозы. Название его составлено из первых букв FORmula TRANslating Syst., что означает «система трансляций формул». Язык был разработан группой сотрудников фирмы IBM под руководством Джона Бэкуса. Первый отчет по созданию ФОРТРАНА (версия ФОРТРАН I) был опубликован 10 ноября 1954 г. (последний датируется 1957 г.) Первая версия была реализована на компьютере IBM-704. Для написания первого компилятора потребовалось 18 человеко-лет.

В первые годы язык был встречен скептически, так как ожидали снижения эффективности программ. Однако Д. Бэкус с сотрудниками группы создали такой компилятор, который генерировал программы, по качеству не уступающие составленным вручную на машинном языке.

Язык получил всеобщее признание, в особенности в области научных и инженерных вычислений, для этих целей он используется и поныне. К характерным чертам языка относится сравнительная простота и легкость его изучения, близость записи арифметических выражений к обычной математической записи, возможность работы с комплексными переменными и переменными двойной точности, удобные и развитые операторы ввода-вывода данных. К достоинствам относят также простоту компилятора, эффективность получаемых объектных программ, возможность независимой

компиляции подпрограмм и относительную легкость в обнаружении ошибок в программах.

Наличие многочисленных диалектов не создало серьезных препятствий для широкого обмена программами.

Первая версия языка имела некоторые недостатки и ограничения при ее использовании (например, имена переменных не должны были превышать шесть литер) и представляла лишь ограниченные возможности для определения структур данных. Серьезные недостатки были в управляющих структурах. Точки ветвления приходилось определять метками и следить за тем, чтобы смысл программы не терялся в множестве операторов безусловного перехода.

Небезынтересно отметить, что дальнейшие версии несут в себе отпечаток ФОРТРАНА I и его реализации на ламповой ЭВМ IBM-704. В частности, шестилитерные имена переменных объясняются тем, что длина слова в IBM-704 составляла 64 бит. Максимальное число индексов для элементов массива, равное 3, объясняется наличием в IBM-704 только трех индекс-регистров и т. д.

В 1958 г. появилась усовершенствованная версия языка — ФОРТРАН II, а спустя несколько лет — ФОРТРАН III, которая, однако, не получила распространения.

Наиболее популярной получилась версия под названием ФОРТРАН IV, созданная в 1962 г. В том же году Американский институт стандартов (ANSI) организовал комитет по выработке стандарта ФОРТРАНа, который был принят в 1966 г. Вновь разработанный стандарт получил название ФОРТРАН ANSI или ФОРТРАН-66.

После 1966 г. было разработано еще несколько версий ФОРТРАНА, на основе которых в 1977 г. появилась пересмотренная версия стандартного

языка под названием ФОРТРАН-77. Эта версия не исключает использование старых фортрановских программ, она лишь расширяет возможности языка в операциях
ввода-вывода, в средствах описания данных и подпрограмм, в конструкциях, которые ранее допускали значения только целого типа, и т. п.
В мае 1984 г. комитет X3.13 Американского национального института стандартов рассмотрел предложения по выработке новой стандартной версии языка ФОРТРАН на период 1980 — 1990 гг. под названием ФОРТРАН 8х. В новой версии, совместимой с языком ФОРТРАН-77, предлагаются изменения, касающиеся в основном работы с файлами: процедур открытия и закрытия файлов, доступа к ним и поиска записи в файле с помощью различных операторов, влияния операторов на эффективность работы процессора.
Трансляторы языка ФОРТРАН существуют на всех вычислительных машинах и, в частности, на машинах серии СМ и диалого-вычислительных комплексов (ДВК).

В СССР в 1978 г. разработаны стандарты на языки ФОРТРАН (ГОСТ 23056-78) и базисный ФОРТРАН (ГОСТ 23057-78). Стандарт ФОРТРАН ANSI положен в основу реализации версии ФОРТРАН IV на ЕС ЭВМ.

КОБОЛ - Имя языка составлено из первых букв названия COBOL — COmmon Business — Oriented Language, что дословно означает «общий язык деловой ориентации». Язык этот ориентирован на обработку экономической информации. Он разработан рабочей группой, созданной под эгидой исполнительного комитета по языкам систем обработки данных Кодасил (CODASYL — Conference on Data System Languages).

Работы над первой версией языка завершены в декабре 1959 г., а в феврале 1960 г. опубликован предварительный отчет. На разработку КОБОЛ-

60 потребовалось около 4 человеко-лет. Корни КОБОЛа уходят к ранним, мало известным языкам программирования ФЛОУМАТИК, КОМТРАН и ФАКТ, а АЛГОЛ повлиял на выбор символов для КОБОЛа. Первые компиляторы КОБОЛа созданы в декабре 1960 г. одновременно двумя фирмами RCA и Remington — Rand — Univas.

При разработке КОБОЛа ставились цели — сделать язык машиннонезависимым и приблизить его к естественному языку, с тем чтобы для непрофессионального программиста программы были читаемыми. По утверждению многих пользователей, с этой точки зрения КОБОЛ, возможно, самый лучший в отличие от других языков программирования, имеющих формальный синтаксис. В КОБОЛе взят синтаксис английского предложения, поэтому программы на этом языке легко читать.

КОБОЛ был первым языком, в котором средства описания данных соответствуют процедурным возможностям, и первым языком, в котором введен тип данных «запись», являющийся основной структурой данных. К одной из примечательных особенностей КОБОЛа относится рекурсивное описание данных. Другая его особенность в том, что программы на КОБОЛе разбиваются на части, называемые разделами, причем каждая программа состоит из четырех разделов: идентификации, оборудования (среды), данных и процедур.

Раздел идентификации служит для установления тождественности программы и содержит различные пояснения, необходимые для ее документирования. Раздел оборудования содержит данные об используемом оборудовании, в основном периферийном. Раздел данных содержит информацию о типе и структуре данных, организации и распределении памяти и т. п. Раздел процедур содержит алгоритмы вычислений. В свою очередь, эти основные части программы разбиваются на более простые: секцию, параграф, предложение и слово.

Еще одна отличительная черта КОБОЛа — это его постоянное изменение и совершенствование. Вслед за появлением КОБОЛа-60 в следующем году была опубликована вторая версия под названием «КОБОЛ-61″, которая получила широкое распространение, однако имела некоторую несовместимость с КОБОЛом-60. В 1963 г. была опубликована расширенная версия языка, названная «Расширенный КОБОЛ-60″. Два года спустя появилась новая версия с несколько необычным именем: «КОБОЛ, редакция 1965″. Эта версия в качестве американского национального стандарта утверждена в 1968 г. Однако работы по усовершенствованию языка и выработке новых версий продолжаются.

Измененный американский стандарт КОБОЛа принят в 1974 г. с соответствующим названием — КОБОЛ-74. В настоящие время в Американском национальном институте стандартов (ANSI) заканчивается разработка другого стандарта КОБОЛа, который предусматривает введение ряда новых конструкций в язык и отказ от некоторых редко используемых или неудобных операторов и т.д. Новый стандарт предусматривает также значительное сокращение различных подмножеств языка.

В СССР первые компиляторы с подмножества языка КОБОЛ реализованы в 1968 г. на ЭВМ «Днепр-21″ и «Минск-32″, а в 1977 г. был принят отечественный стандарт на язык программирования КОБОЛ (ГОСТ 22558-77).

Оценивая вклад этого языка в теорию и практику программирования, нельзя не указать на противоречивый характер отношений к нему пользователей, с одной стороны, и специалистов по информатике, с другой. Если среди программистов он получил распространение благодаря своей удобочитаемости, а может быть, и ранней стандартизации, то многие ученые его появление восприняли как ошибку, а его использование как «болезнь», с которой необходимо бороться. Причем некоторые из ученых, в частности известный голландский специалист по программированию Э. Дейкстра, выражали свое негативное отношение к КОБОЛу в довольно резкой форме

(ACM SIGPLAN NOTICE, 1982, v. 17, pp. 13-15). Они возражали против использования английского языка как основы КОБОЛа из-за его несовершенной стилистики. Хорошая читаемость программ, утверждали они, не говорит в пользу КОБОЛа, так как программы с введением многочисленных «шумовых» слов становятся слишком многословными. Из-за большой длины программы компиляторы работают медленно, а кроме того, возникают трудности автоматического обнаружения ошибок во время компиляции.

В заключении отметим: несмотря на то что опыт разработки и применения КОБОЛа, очевидно, мало повлиял на создание языков, появившихся после него (за исключением языка PL/1), все же надо признать, что он оставил заметный след в истории развития языков программирования.

Эти языки позволяют писать программы текстом, похожим на английский язык. Компилятор получает каждую команду и преобразует ее в машинный код. Он позволяет использовать имена (переменные) для представления элементов данных так, что одна и та же программа может быть использована с любыми входными данными. Программы, написанные на языках высокого уровня, более компактны, легче для понимания, а вероятность появления в них ошибок меньше.

Недостаток этих программ в том, что компиляция часто приводит к избыточному коду, содержащему лишние сложные подпрограммы, включенные в конечную исполняемую программу. Это также уменьшает скорость работы программы.

Первым языком программирования высокого уровня считается компьютерный язык Plankalkül разработанный немецким инженером Конрадом Цузе ещё в период 1942-1946г. Однако транслятора для него не существовало до 2000 г. Первым в мире транслятором языка высокого уровня является ПП (Программирующая Программа), он же ПП-1, успешно

испытанный в 1954 г. Транслятор ПП-2 (1955 г., 4-й в мире транслятор) уже был оптимизирующим и содержал собственный загрузчик и отладчик, библиотеку стандартных процедур, а транслятор ПП для ЭВМ Стрела-4 уже содержал и компоновщик (linker) из модулей. Однако, широкое применение высокоуровневых языков началось с возникновением Фортрана и созданием компилятора для этого языка.

Ранние языки высокого уровня были довольно специализированными: Фортран (FORmula TRANslation) был предназначен для использования в научных целях, КОБОЛ (Common business Orientated Language) – для использования в мире бизнеса. Появление в 50-х гг. языка BASIC (Beginners All-purpose Symbolic Instruction Code) закрыло существовавший в языках высокого уровня пробел между языками для науки и для бизнеса. BASIC в равной степени годится для любых задач и в то же время достаточно прост для изучения.

Тем временем были изобретены новые методы программирования, которые привели к новой волне языков высокого уровня. Одним из этих языков, выдержавших проверку временем, является основанный на методах структурного программирования Pascal.

Высокоуровневый язык программирования – язык программирования, разработанный для быстроты и удобства использования программистом. Основная черта высокоуровневых языков — это абстракция, то есть введение смысловых конструкций, кратко описывающих такие структуры данных и операции над ними, описания которых на машинном коде (или другом низкоуровневом языке программирования) очень длинны и сложны для понимания.

Так, высокоуровневые языки стремятся не только облегчить решение сложных программных задач, но и упростить портирование программного обеспечения. Использование разнообразных трансляторов и интерпретаторов

обеспечивает связь программ, написанных при помощи языков высокого уровня, с различными операционными системами и оборудованием, в то время как их исходный код остаётся, в идеале, неизменным.

Такого рода оторванность высокоуровневых языков от аппаратной реализации компьютера помимо множества плюсов имеет и минусы. В частности, она не позволяет создавать простые и точные инструкции к используемому оборудованию. Программы, написанные на языках высокого уровня, проще для понимания программистом, но менее эффективны, чем их аналоги, создаваемые при помощи низкоуровневых языков. Одним из следствий этого стало добавление поддержки того или иного языка низкого уровня (язык ассемблера) в ряд современных профессиональных высокоуровневых языков программирования.

Примеры: C, C++, Visual Basic, Java, Python, PHP, Ruby, Perl, Delphi (Pascal). Языкам высокого уровня свойственно умение работать с комплексными структурами данных. В большинство из них интегрирована поддержка строковых типов, объектов, операций файлового ввода-вывода и т. п.

К языкам высокого уровня относят:

1. проблемно-ориентированные (имеют средства для организа ции структур данных, описания алгоритмов и ориентированы на решение задач определенного класса): Фортран, Алгол, Кобол, Ада и др.;
2. универсальные: Алгол 68, PL/1, Паскаль, QBasic, C, C++, C# и др.;
3. языки проектирования программ (системы программирова ния) – в настоящее время имеют самый высокий уровень аб стракции, они расширяются не как языки описания процесса обработки данных, а как средства описания задач: Visual Basic, Delphi, MS Visual C++, Borland C++ Builder и др.;

4. языки гипертекстовой разметки, такие, как HTML – набор кодов, который вводится в документ для обозначения, напри мер, связей между его частями. Команды HTML обеспечива ют соединение сайтов и главных страниц WWW (Всемирной паутины сети Интернет) при помощи гиперссылок и указывают Web-браузеру (программе навигации) способ расположе ния массивов данных;

5. языки описания сценариев – макросы, в которых объедине ны отдельные команды, управляющие средой в соответствии с их списком – программой: (например состоящие из имено ванных последовательностей совокупности указанных нажа тий клавиш при работе с пакетом Microsoft Office);

6. языки моделирования систем: например, GPSS (General Purpose Simulating System) позволяет автоматизировать при моделировании процесс программирования моделей. Язык построен в предположении, что моделью сложной дискретной системы является описание ее элементов и логических правил их взаимодействия. Для определенного класса моделируемых систем выделяют небольшой набор абстрактных элементов объектов. Набор логических правил ограничен и может быть описан небольшим числом стандартных операций. Комплекс программ, описывающих функционирование объектов и выполняющих логические операции, является основой для создания программной модели систем данного класса.

1.3 Технологии и языки программирования сверх высокого уровня

Алгол 68 - Несмотря на схожесть названия и официальную преемственность по отношению к языку Алгол-60, Алгол-68 унаследовал от него лишь некоторые элементы синтаксиса и существенно отличается от языка-предшественника прежде всего наличием большого числа дополнительных синтаксических средств и изобразительных возможностей. В частности, он

включает встроенные в язык средства организации параллельных вычислений, операции со структурами как с едиными объектами, матричные операции.

Наиболее характерной особенностью синтаксиса Алгола-68 является возможность переопределения синтаксиса и операторов — программист может активно расширять язык в требуемом направлении, создавать собственные операции. Целью включения таких средств в язык было достижение максимальной выразительности и получение возможности удобного описания максимально абстрактных алгоритмов. За эти возможности, а также за наличие мощных операторов для обработки структур и массивов Алгол-68 иногда называют «языком сверхвысокого уровня».

Формально Алгол-68 является процедурным языком программирования, ориентированным на описание последовательности команд, но благодаря развитым средствам описания типов и операций он может быть использован для написания программ практически в любом стиле. Так, в приведённом ниже примере программа на Алголе-68 написана в функциональном стиле.

Эта программа реализует классический алгоритм «Решето Эратосфена» для поиска всех простых чисел меньше 100. nil означает *null pointer* (нулевой указатель) в других языках. Нотация *x* of *y* означает «доступ к *x* как к элементу struct или union *y*».

Ещё одна интересная особенность языка Алгол-68 — его «многоязычность» — в язык заложена возможность использования различных таблиц трансляции, что позволяет для каждого естественного языка определить свой набор ключевых слов Алгола-68. В результате программисты получают возможность писать программы ключевыми словами родного языка. Ниже приведён пример простейшей процедуры на Алголе-68, выполняющей вычисление даты, следующей за переданной в параметре, на двух языках: английском и немецком.

APL (от названия книги A Programming Language) — интерактивный матричный язык программирования, основанный на математической нотации Айверсона.

В 1956 году Кеннет Э. Айверсон, сотрудник Гарвардского университета (впоследствии сотрудник фирмы IBM и профессор названного университета) заявил о разрабатываемом им языке, который был закончен в 1961 г. Впервые этот язык был описан в 1962 году в книге Айверсона «A Programming Language» («Некий язык программирования»). Позднее язык был стандартизирован: ISO 8485:1989 описывает Core APL, ISO/IEC 13751:2001 — Extended APL. Кроме того, предпринимались попытки дополнить его современными возможностями — объектно-ориентированным программированием, работой с базами данных SQL и т.д.

Целью создания APL была разработка компактной системы записи (нотации) для описания алгоритмов прикладной математики. В изобретенной Айверсоном оригинальной нотации присутствует множество специфических соглашений и символов (кроме обычных знаков, в алфавите содержится 58 специфических, например, «сапог», «шапка», «посох», «дно» и т.д., и лигатуры — знаки, которые возникают при наложении друг на друга двух простых символов). Все это предназначено для точной и сжатой формулировки алгоритмов, которые затем можно интерпретировать на различные языки программирования. Таким образом, APL создавался не только как язык программирования, но в первую очередь как язык передачи идей. Первые реализации APL как языка программирования были созданы гораздо позже, чем его первые описания и использования для описаний других систем.

Основные особенности APL:

- краткость: для записи функций и операторов используются одиночные символы. Кроме того, большинство символов используются в двух

смыслах — как унарная и как бинарная функции, при этом эти функции могут быть достаточно непохожи.

- необычный набор символов, разработанный специально для этого языка. В свое время для пишущих машинок создавались специальные расширения, предназначенные для написания программ на APL.
- ориентированность на работу с массивами.
- максимальная нагруженность смыслом всех функций и операторов: все они доступны в множестве вариантов функциональности в зависимости от количества и размерности аргументов (один аргумент или два, скаляры они или массивы и т.д.).
- ориентированность на решение проблемы и описание алгоритма, независимость от архитектуры компьютера и операционной системы.
- интерактивность: интерпретируемый APL стал одной из первых интерактивных сред вычислений, что внесло значительный вклад в его успех.
- высокий уровень абстрактности.
- сложность написания и еще большая сложность чтения кода.

APL предоставляет программисту богатый набор функций — средств обработки данных и их преобразования в другие данные. Использование функций в выражениях подчиняется простому правилу приоритета: правым аргументом бинарной функции (или единственным аргументом унарной) всегда является выражение справа от нее целиком. Приоритет можно контролировать в явном виде при помощи скобок, но чаще всего это не нужно. Это правило упрощает анализ иерархической структуры выражений: функции верхнего уровня всегда находятся слева в строке.

Важным элементом языка являются операторы, примерно соответствующие функциям высшего порядка в других функциональных языках. Операторы получают на вход данные или функции и создают функции. В большинстве реализаций APL набор операторов ограничен и фиксирован —

инструментарий определения новых операторов отсутствует. Это отличает язык от других функциональных языков, включая его наследник J, в которых функции высшего порядка можно создавать без ограничений. Синтаксис использования операторов отличается от синтаксиса функций: операторы выполняются слева направо, а унарные операторы используют выражение слева в качестве аргумента.

Одна команда программы на APL выглядит как последовательность функций и операторов, применяющихся к массивам. В языке есть примитивные типы данных (скаляры) — числа и символы (логические значения моделируются числами 0 для false и 1 для true — прием, впервые примененный в APL). Все структуры данных в языке — массивы: одномерные векторы, двухмерные матрицы и многомерные мультитаблицы. Все функции изначально ориентированы на работу с массивами, хотя и могут применяться к скалярам (с разным эффектом в зависимости от размерности аргументов). Развитая система операций над массивами позволяет во многих случаях избежать использования явных команд управления потоком выполнения программы. Во многих реализациях императивных управляющих структур нет вообще, хотя последние реализации склоняются к их добавлению и использованию для того, чтобы подчеркнуть разделение структур данных и управляющих конструкций.

Для реализации разнородных структур данных используется принцип boxing: любой массив может “заключаться в коробку” и рассматриваться как скаляр. В таком представлении над ним нельзя производить операции, зато его можно включать (вкладывать) в другие массивы. Содержимое такого массива становится доступным после его “извлечения из коробки”. Эта же техника используется для передачи в функции трех и более аргументов (все функции APL могут принимать от 0 до 2 аргументов, включительно).

В настоящее время APL используется в финансовых и математических приложениях, и существует ряд реализаций, но его популярность пошла на убыль с середины 1980-ых годов, когда начали развиваться другие системы математических вычислений. Многие из них создавались под влиянием APL, но оказались более интуитивными и лучше предназначенными для конечного пользователя.

Повышение уровня этих языков произошло за счет введения сверхмощных операций и операторов.

Программу, написанную на языке программирования высокого уровня, ЭВМ не понимает, поскольку ей доступен только машинный язык. Поэтому для перевода программы с языка программирования на язык машинных кодов используют специальные программы – трансляторы.

Существует три вида транслятора: интерпретаторы (это транслятор, который производит пооператорную обработку и выполнение исходного кода программы), компиляторы (преобразует всю программу в модуль на машинном языке, после чего программа записывается в память компьютера и лишь потом исполняется) и ассемблеры (переводят программу, записанную на языке ассемблера, в программу на машинном языке).

Языки программирования также можно разделять на поколения:

– языки первого поколения: машинно- ориентированные с ручным управлением памяти на компьютерах первого поколения.

– языки второго поколения: с мнемоническим представлением команд, так называемые автокоды.

– языки третьего поколения: общего назначения, используемые для создания прикладных программ любого типа. Например, Бейсик, Кобол, Си и Паскаль.

– языки четвертого поколения: усовершенствованные, разработанные для создания специальных прикладных программ, для управления базами данных.

– языки программирования пятого поколения: языки декларативные, объектно–ориентированные и визуальные. Например, Пролог, ЛИСП (используется для построения программ с использованием методов искусственного интеллекта), Си++, VisualBasic, Delphi.

Языки программирования также можно классифицировать на процедурные и непроцедурные. В процедурных языках программа явно описывает действия, которые необходимо выполнить, а результат задается только способом получения его при помощи некоторой процедуры, которая представляет собой определенную последовательность действий.

Среди процедурных языков выделяют в свою очередь структурные и операционные языки. В структурных языках одним оператором записываются целые алгоритмические структуры: ветвления, циклы и т.д. В операционных языках для этого используются несколько операций. Широко распространены следующие структурные языки: Паскаль, Си, Ада, ПЛ/1. Среди операционных языков известны Фортран, Бейсик, Фокал. Непроцедурное(декларативное) программирование появилось в начале 70-х годов 20 века, К непроцедурному программированию относятся функциональные и логические языки.

В функциональных языках программа описывает вычисление некоторой функции. Обычно эта функция задается как композиция других, более простых, те в свою очередь делятся на еще более простые задачи и т.д. Один из основных элементов функциональных языков – рекурсия. Оператора присваивания и циклов в классических функциональных языках нет.

В логических языках программа вообще не описывает действий. Она задает данные и соотношения между ними. После этого системе можно задавать

вопросы. Машина перебирает известные и заданные в программе данные и находит ответ на вопрос. Порядок перебора не описывается в программе, а неявно задается самим языком. Классическим языком логического программирования считается Пролог. Программа на Прологе содержит, набор предикатов–утверждений, которые образуют проблемно–ориентированную базу данных и правила, имеющие вид условий.

Можно выделить еще один класс языков программирования - объектно-ориентированные языки высокого уровня. На таких языках не описывают подробной последовательности действий для решения задачи, хотя они содержат элементы процедурного программирования. Объектно-ориентированные языки, благодаря богатому пользовательскому интерфейсу, предлагают человеку решить задачу в удобной для него форме. Примером такого языка может служить язык программирования визуального общения ObjectPascal.

Первый объектно-ориентированный язык программирования Simula был создан в 1960-х годах Нигаардом и Далом.

Другая классификация делит языки на вычислительные и языки символьной обработки. К первому типу относят Фортран, Паскаль, Алгол, Бейсик, Си, ко второму типу - Лисп, Пролог, Снобол и др.

Языки описания сценариев, такие как Perl, Python, Rexx, Tcl и языки оболочек UNIX, предполагают стиль программирования, весьма отличный от характерного для языков системного уровня. Они предназначаются не для написания приложения с нуля, а для комбинирования компонентов, набор которых создается заранее при помощи других языков. Развитие и рост популярности Internet также способствовали распространению языков описания сценариев. Так, для написания сценариев широко употребляется язык Perl, а среди разработчиков Web-страниц популярен JavaScript.

Глава 2.

Объектно - ориентировано технологии и языков программирования

Объектно – ориентированное программирование или ООП (object-oriented programming) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного типа, использующая механизм пересылки сообщений и классы, организованные в иерархию наследования.

Практически сразу после появления языков третьего поколения (1967 г.) ведущие специалисты в области программирования выдвинули идею преобразования постулата фон Неймана: «данные и программы неразличимы в памяти машины». Их цель заключалась в максимальном сближении данных и программы. Решая поставленную задачу, они столкнулись с задачей, решить которую без декомпозиции оказалось невозможно, а традиционные структурные декомпозиции не сильно упрощали задачу. Усилия многих программистов и системных аналитиков, направленные на формализацию подхода, увенчались успехом.

Были разработаны три основополагающих принципа того, что потом стало называться объектно-ориентированным программированием (ООП):

- наследование;
- инкапсуляция;
- полиморфизм.

Результатом их первого применения стал язык Симула-1 (Simula-1), в котором был введен новый тип – объект. В описании этого типа одновременно указывались данные (поля) и процедуры, их обрабатывающие – методы. Родственные объекты объединялись в классы, описания которых

оформлялись в виде блоков программы. При этом класс можно использовать в качестве префикса к другим классам, которые становятся в этом случае подклассами первого.

Впоследствии Симула-1 был обобщен, и появился первый универсальный ООП-ориентированный язык программирования – Симула-67 (67 – по году создания).

Как выяснилось, ООП оказалось пригодным не только для моделирования (Simula) и разработки графических приложений (SmallTalk), но и для создания большинства других приложений, а его приближенность к человеческому мышлению и возможность многократного использования кода сделали его одной из наиболее бурно используемых концепций в программировании.

Объектно-ориентированный подход помогает справиться с такими сложными проблемами, как уменьшение сложности программного обеспечения; повышение надежности программного обеспечения; обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов; обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

2.1 Языки объектно – ориентированного программирования

Объе́ктно-ориенти́рованное программи́рование (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов. В случае языков с[прототипированием](#) вместо классов используются объекты-прототипы.

ООП возникло в результате развития идеологии процедурного программирования, где данные и подпрограммы (процедуры, функции) их

обработки формально не связаны. Для дальнейшего развития объектно-ориентированного программирования часто большое значение имеют понятия события (так называемое событийно-ориентированное программирование) и компонента (компонентное программирование, КОП).

Взаимодействие объектов происходит посредством сообщений. Результатом дальнейшего развития ООП, по-видимому, будет агентно-ориентированое программирование, где *агенты* — независимые части кода на уровне выполнения. Взаимодействие агентов происходит посредством изменения *среды*, в которой они находятся.

Языковые конструкции, конструктивно не относящиеся непосредственно к объектам, но сопутствующие им для их безопасной (исключительные ситуации, проверки) и эффективной работы, инкапсулируются от них в аспекты (в аспектно-ориентированном программировании). Субъектно-ориентированное программирование расширяет понятие объекта посредством обеспечения более унифицированного и независимого взаимодействия объектов. Может являться переходной стадией между ООП и агентным программированием в части самостоятельного их взаимодействия.

Первым языком программирования, в котором были предложены принципы объектной ориентированности, была Симула. В момент его появления в 1967 году в нём были предложены революционные идеи: объекты, классы, виртуальные методы и др., однако это всё не было воспринято современниками как нечто грандиозное. Тем не менее, большинство концепций были развиты Аланом Кэем и Дэном Ингаллсом в языке Smalltalk. Именно он стал первым широко распространённым объектно-ориентированным языком программирования.

В настоящее время количество прикладных языков программирования (список языков), реализующих объектно-ориентированную парадигму, является наибольшим по отношению к другим парадигмам. В области системного программирования до сих пор применяется парадигма

процедурного программирования, и общепринятым языком программирования является Си. При взаимодействии системного и прикладного уровней операционных систем заметное влияние стали оказывать языки объектно-ориентированного программирования. Например, одной из наиболее распространённых библиотек мультиплатформенного программирования является объектно-ориентированная библиотека Qt, написанная на языке C++.

В центре ООП находится понятие *объекта*. Объект — это сущность, которой можно посылать сообщения и которая может на них реагировать, используя свои данные. Объект — это экземпляр класса. Данные объекта скрыты от остальной программы. Сокрытие данных называется инкапсуляцией.

Наличие инкапсуляции достаточно для объектности языка программирования, но ещё не означает его объектной ориентированности — для этого требуется наличие наследования.

Но даже наличие инкапсуляции и наследования не делает язык программирования в полной мере объектным с точки зрения ООП. Основные преимущества ООП проявляются только в том случае, когда в языке программирования реализован полиморфизм — возможность объектов с одинаковой спецификацией иметь различную реализацию.

ООП имеет уже более чем сорокалетнюю историю, но, несмотря на это, до сих пор не существует чёткого общепринятого определения данной технологии. Основные принципы, заложенные в первые объектные языки и системы, подверглись существенному изменению (или искажению) и дополнению при многочисленных реализациях последующего времени. Кроме того, примерно с середины 1980-х годов термин «объектно-ориентированный» стал модным, в результате с ним произошло то же самое, что несколько раньше с термином «структурный» (ставшим модным после распространения технологии структурного программирования) — его стали искусственно «прикреплять» к любым новым разработкам, чтобы обеспечить

им привлекательность.Бьёрн Страуструп в 1988 году писал, что обоснование «объектной ориентированности» чего-либо, в большинстве случаев, сводится к ложному силлогизму: «X — это хорошо. Объектная ориентированность — это хорошо. *Следовательно*, X является объектно-ориентированным».

Тимоти Бадд пишет:

Роджер Кинг аргументированно настаивал, что его кот является объектно-ориентированным. Кроме прочих своих достоинств, кот демонстрирует характерное поведение, реагирует на сообщения, наделён унаследованными реакциями и управляет своим, вполне независимым, внутренним состоянием.

По мнению Алана Кея, создателя языка Smalltalk, которого считают одним из «отцов-основателей» ООП, объектно-ориентированный подход заключается в следующем наборе основных принципов (цитируется по вышеупомянутой книге Т. Бадда).

1. *Всё является объектом.*
2. *Вычисления осуществляются путём взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некоторое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщение — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.*
3. *Каждый объект имеет независимую память, которая состоит из других объектов.*
4. *Каждый объект является представителем класса, который выражает общие свойства объектов (таких, как целые числа или списки).*
5. *В классе задаётся поведение (функциональность) объекта. Тем самым*

все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.

6. *Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определённого класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.*

Таким образом, программа представляет собой набор объектов, имеющих состояние и поведение. Объекты взаимодействуют посредством сообщений. Естественным образом выстраивается иерархия объектов: программа в целом — это объект, для выполнения своих функций она обращается к входящим в неё объектам, которые, в свою очередь, выполняют запрошенное путём обращения к другим объектам программы. Естественно, чтобы избежать бесконечной рекурсии в обращениях, на каком-то этапе объект трансформирует обращённое к нему сообщение в сообщения к стандартным системным объектам, предоставляемым языком и средой программирования.

Устойчивость и управляемость системы обеспечивается за счёт чёткого разделения ответственности объектов (за каждое действие отвечает определённый объект), однозначного определения интерфейсов межобъектного взаимодействия и полной изолированности внутренней структуры объекта от внешней среды (инкапсуляции).

Определить ООП можно и многими другими способами.

Появление в ООП отдельного понятия **класса** закономерно вытекает из желания иметь множество объектов со сходным поведением. Класс в ООП — это в чистом виде абстрактный тип данных, создаваемый программистом. С этой точки зрения объекты являются значениями данного абстрактного типа, а определение класса задаёт внутреннюю структуру значений и набор

операций, которые над этими значениями могут быть выполнены. Желательность иерархии классов (а значит, наследования) вытекает из требований к повторному использованию кода — если несколько классов имеют сходное поведение, нет смысла дублировать их описание, лучше выделить общую часть в общий родительский класс, а в описании самих этих классов оставить только различающиеся элементы.

Необходимость совместного использования объектов разных классов, способных обрабатывать однотипные сообщения, требует поддержки полиморфизма — возможности записывать разные объекты в переменные одного и того же типа. В таких условиях объект, отправляя сообщение, может не знать в точности, к какому классу относится адресат, и одни и те же сообщения, отправленные переменным одного типа, содержащим объекты разных классов, вызовут различную реакцию.

Отдельного пояснения требует понятие обмена сообщениями. Первоначально (например, в том же Smalltalk) взаимодействие объектов представлялось как «настоящий» обмен сообщениями, то есть пересылка от одного объекта другому специального объекта-сообщения. Такая модель является чрезвычайно общей. Она прекрасно подходит, например, для описания параллельных вычислений с помощью *активных объектов*, каждый из которых имеет собственный поток исполнения и работает одновременно с прочими. Такие объекты могут вести себя как отдельные, абсолютно автономные вычислительные единицы. Посылка сообщений естественным образом решает вопрос обработки сообщений объектами, присвоенными полиморфным переменным — независимо от того, как объявляется переменная, сообщение обрабатывает код класса, к которому относится присвоенный переменной объект. Данных подход реализован в языках программирования Smalltalk, Ruby, Objective-C, Python.

Однако общность механизма обмена сообщениями имеет и другую сторону — «полноценная» передача сообщений требует дополнительных

накладных расходов, что не всегда приемлемо. Поэтому во многих современных объектно-ориентированных языках программирования используется концепция «отправка сообщения как вызов метода» — объекты имеют доступные извне методы, вызовами которых и обеспечивается взаимодействие объектов. Данный подход реализован в огромном количестве языков программирования, в том числе C++, Object Pascal, Java, Oberon-2. Однако, это приводит к тому, что сообщения уже не являются самостоятельными объектами, и, как следствие, не имеют атрибутов, что сужает возможности программирования. Некоторые языки используют гибридное представление, демонстрируя преимущества одновременно обоих подходов — например, CLOS, Python.

Концепция виртуальных методов, поддерживаемая этими и другими современными языками, появилась как средство обеспечить выполнение нужных методов при использовании полиморфных переменных, то есть, по сути, как попытка расширить возможности вызова методов для реализации части функциональности, обеспечиваемой механизмом обработки сообщений.

Как уже говорилось выше, в современных объектно-ориентированных языках программирования каждый объект является значением, относящимся к определённому классу. Класс представляет собой объявленный программистом составной тип данных, имеющий в составе:

Поля данных

Параметры объекта (конечно, не все, а только необходимые в программе), задающие его состояние (свойства объекта предметной области). Иногда поля данных объекта называют свойствами объекта, из-за чего возможна путаница. Физически поля представляют собой значения (переменные, константы), объявленные как принадлежащие классу.

Методы

Процедуры и функции, связанные с классом. Они определяют действия, которые можно выполнять над объектом такого типа, и которые сам объект может выполнять.

Классы могут наследоваться друг от друга. Класс-потомок получает все поля и методы класса-родителя, но может дополнять их собственными либо переопределять уже имеющиеся. Большинство языков программирования поддерживает только единичное наследование (класс может иметь только один класс-родитель), лишь в некоторых допускается множественное наследование — порождение класса от двух или более классов-родителей. Множественное наследование создаёт целый ряд проблем, как логических, так и чисто реализационных, поэтому в полном объёме его поддержка не распространена. Вместо этого в 1990-е годы появилось и стало активно вводиться в объектно-ориентированные языки понятие интерфейса. Интерфейс — это класс без полей и без реализации, включающий только заголовки методов. Если некий класс наследует (или, как говорят, реализует) интерфейс, он должен реализовать все входящие в него методы. Использование интерфейсов предоставляет относительно дешёвую альтернативу множественному наследованию.

Взаимодействие объектов в абсолютном большинстве случаев обеспечивается вызовом ими методов друг друга.

Инкапсуляция обеспечивается следующими средствами

Контроль доступа

Поскольку методы класса могут быть как чисто внутренними, обеспечивающими логику функционирования объекта, так и внешними, с помощью которых взаимодействуют объекты, необходимо обеспечить скрытость первых при доступности извне вторых. Для этого в языки вводятся специальные синтаксические конструкции, явно задающие область видимости каждого члена класса. Традиционно это модификаторы public, protected и private, обозначающие, соответственно, открытые члены класса,

члены класса, доступные только из классов-потомков, и скрытые, доступные только внутри класса. Конкретная номенклатура модификаторов и их точный смысл различаются в разных языках.

Методы доступа

Поля класса в общем случае не должны быть доступны извне, поскольку такой доступ позволил бы произвольным образом менять внутреннее состояние объектов. Поэтому поля обычно объявляются скрытыми (либо язык в принципе не позволяет обращаться к полям класса извне), а для доступа к находящимся в полях данным используются специальные методы, называемые методами доступа. Такие методы либо возвращают значение того или иного поля, либо производят запись в это поле нового значения. При записи метод доступа может проконтролировать допустимость записываемого значения и, при необходимости, произвести другие манипуляции с данными объекта, чтобы они остались корректными (внутренне согласованными). Методы доступа называют ещё аксессорами (от англ. *access* — доступ), а по отдельности — геттерами (англ. *get* — чтение) и сеттерами (англ. *set* — запись)[6].

Свойства объекта

Псевдополя, доступные для чтения и/или записи. Свойства внешне выглядят как поля и используются аналогично доступным полям (с некоторыми исключениями), однако фактически при обращении к ним происходит вызов методов доступа. Таким образом, свойства можно рассматривать как «умные» поля данных, сопровождающие доступ к внутренним данным объекта какими-либо дополнительными действиями (например, когда изменение координаты объекта сопровождается его перерисовкой на новом месте). Свойства, по сути, не более чем синтаксический сахар, поскольку никаких новых возможностей они не добавляют, а лишь скрывают вызов методов доступа. Конкретная языковая реализация свойств может быть разной. Например, в C# объявление свойства непосредственно содержит код методов доступа, который вызывается только при работе со свойствами, то

есть не требует отдельных методов доступа, доступных для непосредственного вызова. В Delphi объявление свойства содержит лишь имена методов доступа, которые должны вызываться при обращении к полю. Сами методы доступа представляют собой обычные методы с некоторыми дополнительными требованиями к сигнатуре.

Полиморфизм реализуется путём введения в язык правил, согласно которым переменной типа «класс» может быть присвоен объект любого класса-потомка её класса.

ООП ориентировано на разработку крупных программных комплексов, разрабатываемых командой программистов (возможно, достаточно большой). Проектирование системы в целом, создание отдельных компонентов и их объединение в конечный продукт при этом часто выполняется разными людьми, и нет ни одного специалиста, который знал бы о проекте всё.

Объектно-ориентированное проектирование состоит в описании структуры и поведения проектируемой системы, то есть, фактически, в ответе на два основных вопроса:

- Из каких частей состоит система.
- В чём состоит ответственность каждой из частей.

Выделение частей производится таким образом, чтобы каждая имела минимальный по объёму и точно определённый набор выполняемых функций (обязанностей), и при этом взаимодействовала с другими частями как можно меньше.

Дальнейшее уточнение приводит к выделению более мелких фрагментов описания. По мере детализации описания и определения ответственности выявляются данные, которые необходимо хранить, наличие близких по поведению агентов, которые становятся кандидатами на реализацию в виде классов с общими предками. После выделения компонентов и определения

интерфейсов между ними реализация каждого компонента может проводиться практически независимо от остальных (разумеется, при соблюдении соответствующей технологической дисциплины).

Большое значение имеет правильное построение иерархии классов. Одна из известных проблем больших систем, построенных по ООП-технологии — так называемая *проблема хрупкости базового класса*. Она состоит в том, что на поздних этапах разработки, когда иерархия классов построена и на её основе разработано большое количество кода, оказывается трудно или даже невозможно внести какие-либо изменения в код базовых классов иерархии (от которых порождены все или многие работающие в системе классы). Даже если вносимые изменения не затронут интерфейс базового класса, изменение его поведения может непредсказуемым образом отразиться на классах-потомках. В случае крупной системы разработчик базового класса просто не в состоянии предугадать последствия изменений, он даже не знает о том, как именно базовый класс используется и от каких особенностей его поведения зависит корректность работы классов-потомков.

2.2 Основные концепции объектно – ориентированной технологии программирования

Основные концепции объектно-ориентированного программирования (ООП) являются объект и класс. Понятие «объект» может быть определено следующим образом.

Объект - это нечто, обладающее состоянием, поведением и идентичностью.

В качестве еще одного примера может быть приведен интерфейсный объект, например кнопка. Кнопка однозначно выделяется из всего прочего интерфейса - т.е обладает идентичностью. Ее состояние - надпись на кнопке, цвет, размер, форма и т.п. Ее поведение - способность быть нажатой - отобразить этот процесс на экране и предать сообщение об этом событии тому объекту, который должен на него реагировать.

Идентичность - это такое свойство объекта, которое отличает его от всех других объектов.

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

Поведение - это то, как объект действует и реагирует; поведение выражается в терминах изменения состояния объекта и передачи сообщений. Структура и поведение схожих объектов определяет для них общий класс. Например класс графических интерфейсных элементов - кнопок, переключателей, окон ввода данных, и т.п. Другой пример - если бы мы занялись моделированием экосистемы, то могли бы выделить классы растений, насекомых, рыб, млекопитающих и т.д. Может быть дано следующее определение понятия «класс». Класс - это некое множество объектов, имеющих общую структуру и общее поведение.

Термины «экземпляр класса» и «объект» взаимозаменяемы. Классы вступают между собой в некоторое отношение, называемое иерархией наследования. Наследование - это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других классов (множественное наследование).

Для примера рассмотрим программу моделирующую работу цеха. На верхнем уровне иерархии наследования могут быть выделены такие классы, как работник, станок, помещение. Различные категории работников в свою очередь образуют классы, которые наследуют структуру и поведение класса работник (например все работники должны приходить на работу в одно время), но и добавляют что-то свое (выполняют разные категории работ). То же самое относится к станкам и помещениям - станки разного типа и помещения разного назначения.

Предлагается следующее определение термина «объектно-ориентированное программирование»:

Объектно-ориентированное программирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является объектом определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части:

1 ООП использует в качестве базовых элементов объекты, а не алгоритмы,

2 каждый объект является экземпляром какого-либо определенного класса

3 классы организованы иерархически.

Программа будет объектно-ориентированной только при соблюдении всех трех указанных условий.

Для разработки объектно-ориентированных программ используются объектно-ориентированные языки программирования. Объектно-ориентированный язык программирования может быть определен как язык, имеющий средства хорошей поддержки объектно-ориентированного стиля программирования, т.е. при разработке программ на этом языке удобно пользоваться таким стилем программирования. Если написание программ в стиле ООП требует специальных усилий или оно невозможно совсем, то этот язык не является объектно-ориентированным.

Язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:

- поддерживаются объекты, т.е. абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним;
- объекты относятся к соответствующим типам (классам);
- типы (классы) могут наследовать атрибуты супертипов (суперклассов).

Согласно этому определению такие языки как C++ и Object Pascal являются объектно-ориентированными.

Дальнейшее изложение ООП ведется на примере языка Object Pascal, реализованном в Borland Delphi. В некоторых случаях для сравнения

рассматривается реализация объектно-ориентированного подхода в языке C++.

2.3 Особенности объектно – ориентированной технологии программирования

Во многих учебниках выделяют такие основные идеи ООП как наследование, инкапсуляция и полиморфизм. Заключаются они примерно в следующем:

Наследование. механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами. Позволяет строить иерархии классов Возможность выделять общие свойства и методы классов в один класс верхнего уровня (родительский). Классы, имеющие общего родителя, различаются между собой за счет включения в них различных дополнительных свойств и методов.

Класс, от которого произошло наследование, называется базовым или родительским (англ. base class). Классы, которые произошли от базового, называются потомками, наследниками или производными классами (англ. derived class).

В некоторых языках используются абстрактные классы. Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. То есть от абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник вуза», от которого наследуются классы «аспирант», «профессор» и т. д. Так как производные классы имеют

общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза».

Наследование в C++:

```
class A{    //базовый класс
};

class B : public A{    //public наследование
};

class C : protected A{    //protected наследование
};

class Z : private A{    //private наследование
};
```

В C++ существует три типа наследования: public, protected, private. Спецификаторы доступа членов базового класса меняются в потомках следующим образом:

Если класс объявлен как базовый для другого класса со спецификатором доступа public, тогда public члены базового класса доступны как public члены производного класса, protected члены базового класса доступны как protected члены производного класса.

Если класс объявлен как базовый для другого класса со спецификатором доступа protected, тогда public и protected члены базового класса доступны как protected члены производного класса.

Если класс объявлен как базовый для другого класса со спецификатором доступа private, тогда public и protected члены базового класса доступны как private члены производного класса.

Одним из основных преимуществ public-наследования является то, что указатель на классы-наследники может быть неявно преобразован в указатель на базовый класс, то есть для примера выше можно написать:

```
A* a = new B();
```

Эта интересная особенность открывает возможность динамической идентификации типа (RTTI).

инкапсуляция. (лат. *in capsula*) — механизм языка программирования, предоставляющий возможность обрабатывать несколько единиц данных как одну единицу. Является важным инструментом любого промышленного языка.

В одних языках (например, C++) термин тесно пересекается (вплоть до отождествления) с сокрытием, в других (например, ML) эти понятия абсолютно независимы. В некоторых языках (например, Smalltalk или Python) сокрытие отсутствует, хотя возможности инкапсуляции развиты хорошо.

В языках, поддерживающих замыкания, инкапсуляция рассматривается как понятие не присущее исключительно объектно-ориентированному программированию. Также, реализации абстрактных типов данных (например, модули) предлагают похожую на инкапсуляцию модель сокрытия данных. Свойства и методы класса делятся на доступные из вне (опубликованные) и недоступные (защищенные). Защищенные атрибуты нельзя изменить, находясь вне класса. Опубликованные же атрибуты также называют интерфейсом объекта, т. к. с их помощью с объектом можно взаимодействовать. По идеи, инкапсуляция призвана обеспечить надежность программы, т.к. изменить существенные для существования объекта атрибуты становится невозможно.

```
class A
```

```
{
 public:
   int a, b; //данные открытого интерфейса
   int ReturnSomething(); //метод открытого интерфейса
 private:
   int Aa, Ab; //скрытые данные
   void Do_Something(); //скрытый метод
};
```

Класс A инкапсулирует свойства Aa, Ab и метод Do_Something(), представляя внешний интерфейс ReturnSomething, a, b.

Целью инкапсуляции является обеспечение согласованности внутреннего состояния объекта. В C# для инкапсуляции используются публичные свойства и методы объекта. Переменные, за редким исключением, не должны быть публично доступными. Проиллюстрировать инкапсуляцию можно на простом примере. Допустим, нам необходимо хранить вещественное значение и его строковое представление (например, для того, чтобы не производить каждый раз конвертацию в случае частого использования). Пример реализации без инкапсуляции таков:

```
class NoEncapsulation
    {
        public double ValueDouble;
        public string ValueString;
    }
```

При этом мы можем отдельно изменять как само значение *Value*, так и его строковое представление, и в некоторый момент может возникнуть их несоответствие (например, в результате исключения). Пример реализации с использованием инкапсуляции:

```
class EncapsulationExample
    {
        private double valueDouble;
```

```
        private string valueString;

        public double ValueDouble
        {
            get { return valueDouble; }
            set
            {
                valueDouble = value;
                valueString = value.ToString();
            }
        }

        public string ValueString
        {
            get { return valueString; }
            set
            {
                double tmp_value = Convert.ToDouble(value); //здесь может
возникнуть исключение
                valueDouble = tmp_value;
                valueString = value;
```

Здесь доступ к переменным *valueDouble* и *valueString* возможен только через свойства *ValueDouble* и *ValueString*. Если мы попытаемся присвоить свойству *ValueString* некорректную строку и возникнет исключение в момент конвертации, то внутренние переменные останутся в прежнем, согласованном состоянии, поскольку исключение вызывает выход из процедуры.

Полиморфизм. Существует несколько принципиально различных видов полиморфизма, два из которых были описаны Кристофером Стрэчи в1967 году.

Если функция описывает разные реализации (возможно, с различным поведением) для ограниченного набора явно заданных типов и их комбинаций, это называется ситуативным полиморфизмом (ad

hoc polimorphism). Ситуативный полиморфизм поддерживается во многих языках посредством перегрузки функций и методов.

Если же код написан отвлеченно от конкретного типа данных и потому может свободно использоваться с любыми новыми типами, имеет место параметрический полиморфизм. Джон С.Рейнольдс, и независимо от него Жан-Ив Жирар формально описали эту нотацию как развитие лямбда-исчисления (названную полиморфным лямбда-исчислением или Системой F). Параметрический полиморфизм широко используется в статически типизируемых функциональных языках программирования. В объектно-ориентированном сообществе программирование с использованием параметрического полиморфизма называется обобщённым программированием.

Полиморфизм является фундаментальным свойством системы типов. Обычно различают статическую неполиморфную типизацию (потомки Алгола и BCPL), динамическую типизацию (потомки Lisp, Smalltalk, APL) и статическую полиморфную типизацию (потомкиML). В наибольшей степени применение ad hoc полиморфизма присуще при неполиморфной типизации, параметрического — при полиморфной. Параметрический полиморфизм и динамическая типизация намного существеннее, чем ситуативный полиморфизм, повышают коэффициент повторного использования кода, поскольку определенная единственный раз функция реализует без дублирования заданное поведение для бесконечного множества вновь определяемых типов, удовлетворяющих требуемым в функции условиям.

Некоторые языки совмещают различные формы полиморфизма, порой сложным образом, что формирует самобытную идеологию в них и влияет на применяемые методологии декомпозиции задач. Например, в Smalltalk любой класс способен принять сообщения любого типа, и либо обработать его самостоятельно (в том числе посредством интроспекции),

либо ретранслировать другому классу — таким образом, несмотря на широкое использование перегрузки функций, формально любая операция является неограниченно полиморфной и может применяться к данным любого типа.

В объектно-ориентированном программировании полиморфизм подтипов (или полиморфизм включения) представляет собой концепцию в теории типов, предполагающую использование единого имени (идентификатора) при обращении к объектам нескольких разных классов, при условии, что все они являются подклассами одного общего надкласса (суперкласса). Полиморфизм подтипов состоит в том, что несколько типов формируют подмножество другого типа (их базового класса) и потому могут использоваться через общий интерфейс.

Параметрический полиморфизм позволяет определять функцию или тип данных обобщённо, чтобы значения могли обрабатываться идентично вне зависимости от их типа. Параметрический полиморфизм делает язык более выразительным, сохраняя полную статическую типобезопасность

Параметрический полиморфизм повсеместно используется в функциональном программировании, где он обычно обозначается просто как «полиморфизм». Следующий пример демонстрирует параметризованный тип «список» и две определённые на нём параметрически полиморфные функции:

```
data List a = Nil | Cons a (List a)

length :: List a -> Integer
length Nil = 0
length (Cons x xs) = 1 + length xs

map :: (a -> b) -> List a -> List b
map f Nil = Nil
map f (Cons x xs) = Cons (f x) (map f xs)
```

Концепция параметрического полиморфизма применима как к данным, так и к функциям. Функция, принимающая на входе или порождающая значения разных типов, называется полиморфной функцией. Тип данных, используемый как обобщённый (например,список элементов произвольного типа), называется полиморфным типом данных. Параметрически полиморфная функция использует аргументы на основе поведения, а не значения, апеллируя лишь к необходимым ей свойствам аргументов, что делает её применимой в любом контексте, где тип объекта удовлетворяет заданным требованиям поведения. Таким образом, реализуется концепция параметричности. Параметрический полиморфизм связывается с подтипами[en] данных понятиями связанной квантификации и ковариантности/контравариантности (или полярности) конструкторов типов.

Параметрический полиморфизм также доступен в некоторых императивных (в частности, объектно-ориентированных) языках программирования, где для его обозначения обычно используется термин «обобщённое программирование»:

```
struct segment { int start; int end; };

int seg_cmpr( struct segment *a, struct segment *b )
{ return abs( a->end - a->start ) - abs( b->end - b->start ); }

int str_cmpr( char **a, char **b )
{ return strcmp( *a, *b ); }

struct segment segs[] = { {2,5}, {4,3}, {9,3}, {6,8} };
char* strs[] = { "three", "one", "two", "five", "four" };

main()
{
    qsort( strs, sizeof(strs)/sizeof(char*), sizeof(char*),
                (int (*)(void*,void*))str_cmpr );
```

```
    qsort( segs, sizeof(segs)/sizeof(struct segment), sizeof(struct
segment),
                  (int (*)(void*,void*))seg_cmpr );
    ...
}
```

Здесь одна функция обрабатывает массивы элементов любого типа, для которого определена функция сравнения. Полиморфное поведение обеспечивается здесь не на уровне типов, а за счёт передачи всего необходимого поведения явным образом через нетипизированные указатели — другими словами, за счёт бестипового подмножества языка.

```
template <typename T> T max(T x, T y)
{
    if (x < y)
        return y;
    else
        return x;
}

int main()
{
    int a = max(10,15);
    double f = max(123.11, 123.12);
    ...
}
```

Здесь определение функции на уровне исходного кода по-прежнему единственно, но в результате компиляции порождается такое количество перегруженных мономорфных функций, с каким количеством типов данных функция вызывается в программе. Другими словами, параметрический полиморфизм на уровне исходного кода транслируется в ad hoc полиморфизм на уровне целевой платформы. Такое превращение

называется мономорфизацией (англ. *monomorphizing*). Мономорфизация повышает быстродействие (точнее, делает полиморфизм «бесплатным»), но вместе с тем увеличивает размер выходного машинного кода. В данном случае мономорфизация неизбежна из-за отсутствия поддержки полиморфизма в системе типов языка (полиморфный язык здесь являетсястатической[en] надстройкой над мономорфным ядром языка). Более развитые системы типов (такие как Хиндли — Милнер) делают мономорфизацию опциональной, то есть позволяют выбирать между сохранением единственности тела полиморфной функции и размножением мономорфных тел. Например, для языка Standard ML компилятор SML/NJ сохраняет тела функций единственными, в то время как MLton выполняет полную мономорфизацию программы. В таких языках увеличение размера машинного кода может оказываться незначительным за счёт других выводимых из системы типов свойств (так, в MLton увеличение кода за счёт мономорфизации не превышает 30 %)

Кристофер Стрэчи избрал термин «ситуативный полиморфизм» для описания полиморфных функций, вызываемых для аргументов разных типов, но реализующих различное поведение в зависимости от типа аргумента (называемых также перегруженными функциямии перегруженными операторами). Термин «ad hoc» (лат. *спонтанный, сделанный под текущую задачу*) в этом смысле не несёт уничижительного подтекста — он означает лишь, что этот вид полиморфизма не является фундаментальным свойством системы типов языка. В следующем примере функции `Add` выглядят как реализующие один и тот же функционал над разными типами, но компилятор определяет их как две совершенно разные функции

```
program Adhoc;

function Add( x, y : Integer ) : Integer;
begin
```

```
    Add := x + y
end;

function Add( s, t : String ) : String;
begin
    Add := Concat( s, t )
end;

begin
    Writeln(Add(1, 2));
    Writeln(Add('Hello, ', 'World!'));
end.
```

В динамически типизируемых языках ситуация может быть более сложной, так как выбор требуемой функции для вызова может быть осуществлён только во время исполнения программы.

Ситуативный полиморфизм иногда используется совместно с параметрическим. В языке Haskell стремление предоставить одновременно полиморфизм и перегрузку привело к необходимости введения принципиально нового понятия — Классы типов[en] (не следует путать с классом в объектно-ориентированном программировании). С одной стороны, это позволяет существенно повысить выразительность программ, с другой чрезмерное их использование может приводить к сбоям механизма выведения типов, что вынуждает программистов отказываться от его использования, явно декларируя типы функций.

Некоторые языки представляют идею подтипов для ограничения спектра типов, применимых в определённом частном случае параметрического полиморфизма. В этих языках полиморфизм подтипов (обычно называемый также динамическим полиморфизмом) позволяет функции, определённой на типе T, также корректно исполняться для аргументов, принадлежащих типу S, являющемусяподтипом T (в соответствии с принципом подстановки Барбары Лисков). Такое отношение типов обычно записывается как S <: T.

При этом тип T называется надтипом (или супертипом) для S, что обозначается как T :> S.

Например, если имеются типы Number, Rational и Integer, связанные отношениями Number :> Rational и Number :> Integer, то функция, определённая на типе Number, также сможет принять на вход аргументы типов Integer или Rational, и её поведение будет идентичным. Действительный тип объекта может быть скрыт как «чёрный ящик», и предоставляться лишь по запросу идентификации объекта. На самом деле, если тип Number является абстрактным, то конкретного объекта этого типа даже не может существовать (см.абстрактный тип данных, абстрактный класс). Данная иерархия типов известна — особенно в контексте языка Scheme — как числовая башня[en], и обычно содержит большее количество типов.

Объектно-ориентированные языки программирования реализуют полиморфизм подтипов посредством наследования, то есть определения подклассов. В большинстве реализаций каждый класс содержит т. н. виртуальную таблицу — таблицу функций, реализующих полиморфную часть интерфейса класса — и каждый объект (экземпляр класса) содержит указатель на виртуальную таблицу своего класса, по согласованию с которой производится вызов полиморфного метода. Такой механизм используется в случаях:

- позднего связывания, где виртуальные функции не связаны до момента вызова;
- одиночной диспетчеризации (то есть одноаргументного полиморфизма), где виртуальные функции связываются посредством простого просмотра виртуальной таблицы по первому аргументу (данному экземпляру класса), так что динамические типы остальных аргументов не учитываются.

То же применимо и к большинству остальных популярных объектных моделей. Некоторые, однако, такие как CLOS, предоставляютмножественную диспетчеризацию, в которой метод полиморфен по всем аргументам.

В следующем примере коты и псы являются подтипами животных. Процедура определена для животных, но также будет работать корректно, получив на входе один из подтипов:

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() { return "Meow!"; }
}

class Dog extends Animal {
    String talk() { return "Woof!"; }
}
  public class MyClass {

    public static void write(Animal a) {
        System.out.println(a.talk());
    }
     public static void main(String args[]) {
        write(new Cat());
        write(new Dog());
    }
}
```

В связи со своими особенностями объектно-ориентированное программирование имеет ряд преимуществ перед структурным (и др.) программированием. Выделим некоторые из них:

1. Использование одного и того же программного кода с разными данными. Классы позволяют создавать множество объектов, каждый из которых имеет собственные значения атрибутов. Нет потребности вводить

множество переменных, т.к объекты получают в свое распоряжение индивидуальные так называемые пространства имен. Пространство имен конкретного объекта формируется на основе класса, от которого он был создан, а также от всех родительских классов данного класса. Объект можно представить как некую упаковку данных.

2. Наследование и полиморфизм позволяют не писать новый код, а настраивать уже существующий, за счет добавления и переопределения атрибутов. Это ведет к сокращению объема исходного кода.

ООП позволяет сократить время на написание исходного кода, однако ООП всегда предполагает большую роль предварительного анализа предметной области, предварительного проектирования. От правильности решений на этом предварительном этапе зависит куда больше, чем от непосредственного написания исходного кода.

Глава 3.

Реализация основных объектов и концепций объектно – ориентированной технологии программирования на языке C++

Сложность современного программного обеспечения требует от разработчиков владения наиболее перспективными технологиями его создания. Одной из таких технологий на настоящий момент является объектно-ориентированное программирование (ООП), применение которого позволяет разрабатывать программное обеспечение повышенной сложности за счет улучшения его технологичности (лучших механизмов разделения данных, увеличения повторяемости кодов, использования стандартизованных интерфейсов пользователя и т.д.). Чтобы технологически грамотно использовать ООП, необходимо хорошо понимать его основные концепции и научиться мыслить при разработке программы в понятиях ООП.

Несмотря на то, что в различных источниках делается акцент на те или иные особенности внедрения и применения ООП, 3 основных (базовых) понятия ООП остаются неизменными. К ним относятся:

Наследование (**Inheritance**)

Инкапсуляция (**Encapsulation**)

Полиморфизм (**Polymorphism**)

Эти понятия, как три кита, лежат в основе мира ООП и, конечно же, будут рассмотрены мною подробнейшим образом. Наряду с ними будет рассмотрен и ряд других понятий и определений.

А для начала рассмотрим пару мнений, которые в какой-то степени помогут Вам прояснить сложившуюся вокруг поставленного вопроса ситуацию.

Итак, первое из них имеет некоторый академический оттенок:

ООП позволяет разложить проблему на связанные между собой задачи. Каждая проблема становится самостоятельным объектом, содержащим свои собственные коды и данные, которые относятся к этому объекту. В этом

случае исходная задача в целом упрощается, и программист получает возможность оперировать с большими по объему программами.

В этом определении ООП отражается известный подход к решению сложных задач, когда мы разбиваем задачу на подзадачи и решаем эти подзадачи по отдельности. С точки зрения программирования подобный подход значительно упрощает разработку и отладку программ.

Не очень понятно ? Тогда попробуем зайти с другой стороны и упростим описание. Начнем с общеизвестного примера.

Вы сидите за обеденным столом и вам понадобилась соль, которая находится на другом конце стола и вы не можете до нее дотянуться. Обидно, но чтобы не оставаться без соли Вы просите своего друга передать Вам соль. Ура ! Вы получили то, что хотели.

Что Вы сделали для того, чтобы получить соль ? Просто сказали: "Передай мне, пожалуйста, соль".

Однако вместо этой простой фразы можно было бы сказать: "Пожалуйста, убери твою правую руку со стакана воды. Затем протяни ее влево на такое-то расстояние(или пока она не коснется солонки). Возьми баночку на которой написано "Соль". Подними ее. Сделай плавное движение правой рукой по дуге в направлении меня. Остановись когда твоя рука коснется моей. Затем подожди пока пока мои пальцы возьмут ее и только затем отпусти солонку и верни руку в исходной положение"

Неправда ли, звучит странно и как-то даже глупо. Да, для нормального человека это звучит глупо. Но эти два описания одного и того же действия хорошо иллюстрируют разницу объектно-ориентированного (первый вариант действий) и обычного структурного (второй, многошаговый вариант) подхода к программированию.

Так в чем же принципиальная разница ?

Разница заключается в том, что объектно-ориентированный подход с просьбой передать солонку оставляет за объектом (Вашим другом) право решать как отреагировать и что сделать в ответ на поступившую просьбу. А

Вы можете даже не знать (к примеру, не видеть) как Ваш друг (объект) выполнил Вашу просьбу. А зачем Вам это знать ? Вы в стандартной форме поставили перед ним задачу (сделали вызов) и получили ответ.

При процедурном же развитии события, второй вариант описания этого же действия, необходимо описать каждый шаг, каждое движение для достижения конечного результата.

Этот простой пример иллюстрирует только один из принципов ООП, суть которого: "Объект отвечает за все действия, которые он производит в ответ на запрос клиента".

К сожалению, этот пример не может показать всей полноты преимуществ использования ООП, но и с помощью него можно почувствовать глубину отличий применения ООП и структурного программирования для решения даже таких простых задач.

3.1 Реализация массивов, структур, объединений и классов

Очень часто перед трейдером стоит задача сохранения массивов данных (список тикеров позиций, открытых этим экспертом, и т.д.) в глобальных переменных.

Конечно, можно все данные хранить в переменных эксперта, но при перезапуске клиентского терминала они будут потеряны, поэтому лучше критические данные, которые нельзя потерять ни в коем случае, хранить не в памяти, а сразу в глобальных переменных.

Функции для работы с такими массивами:

AddItem()

DeleteItem()

GetItem()

Count()

Search()

BSearch

Sort()

В каждой из этих функций мы также будем использовать "критические секции". А иначе не избежать ситуации, когда данные будут испорчены, если в момент выполнения этих функций будет осуществлен доступ к ним из нескольких советников одновременно.

Предположим, что наш глобальный массив имеет имя, которое содержится в переменной global_array_name.

Тогда:

Количество элементов массива будет находится в глобальной переменнойglobal_array_name+"Count".

Глобальная переменная с именем global_array_name+"Lock" используется в качестве критической секции при вызове функций для работы с этим массивом.

Элементы массива будут находится в переменных global_array_name+"1",global_array_name+"2" и т.д.

Если в переменной global_array_name+"IsSorted" находится ненулевое значение, то массив отсортирован. В противном случае - неотсортированный.

Теперь мы знаем достаточно, чтобы создать необходимые функции для работы глобальным массивом. В следующем выпуске я расскажу о функции Count().

В структурном программировании понималось разделение алгоритмов, где каждый алгоритм выполнял один из этапов общего процесса. Основой этого принципа является проектирование "сверху вниз".

Можно рассматривать проблему и с другой стороны. Попытайтесь разделить некую систему по признаку принадлежности ее элементов различным абстракциям данной проблемной области.

Оба подхода могут, решая одну и ту же проблему, делать это по-разному. Во втором случае мир представляет собой совокупность взаимодействующих

объектов. Каждый объект в такой системе моделирует поведение объекта реального мира.

Encapsulation (сокрытие данных) - ключевое понятие при работе с объектами. Формально инкапсуляцией считается объединение данных и операций над ними в одном пакете и сокрытие данных от других объектов. Данные в объекте называются instance fields (поля экземпляра), а функции и процедуры, выполняющие операции над данными - methods (методы). Конкретный объект (экземпляр класса) имеет определенные значения полей. Множество этих значений называется текущим состоянием (state) объекта. Применение любого метода к некоторому объекту может изменить его состояние.

Реализация прикладной программной системы, спроектированной с помощью объектно-ориентированной методологии (например, методологии OMT), на языке C++ начинается с определения классов, разработанных на этапе проектирования, на этом языке. При этом желательно сохранение имен и, по возможности, других обозначений, введенных на этапе проектирования. Рассмотрим в качестве примера, как реализовать на языке C++ класс Window,.

В определении класса на языке C++ и атрибуты, и методы называются *членами* этого класса; их определения могут следовать в тексте определения класса в произвольном порядке. Члены класса могут быть общедоступными (public), или приватными (private); вне класса определен доступ только к его общедоступным членам, а приватные члены доступны только методам своего класса. В рассматриваемом примере все атрибуты являются приватными, а все методы (кроме метода add_to_selections) - общедоступными, так что прочитать или изменить значение каждого атрибута можно только с помощью соответствующего метода; это рекомендуемая, хотя и не обязательная дисциплина программирования на

языке C++ (определение всех атрибутов класса как приватных называется *инкапсуляцией* данных).

3.2 Реализация концепции полиморфизма

Формализуем понятие полиморфизма применительно к объектно-ориентированному подходу. Под полиморфизмом будем иметь в виду возможность оперирования объектами без однозначной идентификации их типов.

Наметим концепции, объединяющие функциональный и объектно-ориентированный подходы к программированию с точки зрения полиморфизма.

Как было отмечено в ходе исследования функционального подхода к программированию, концепция полиморфизма предполагает в части реализации отложенное связывание переменных со значениями. При этом во время выполнения программы происходят так называемые «ленивые» или, иначе,«замороженные» вычисления. Таким образом, означивание языковых идентификаторов выполняется по мере необходимости.

В случае объектно-ориентированного подхода к программированию теоретический и практический интерес при исследовании концепции полиморфизма представляет отношение наследования, прежде всего, в том смысле, что это отношение порождает семейства полиморфных языковых объектов.

С точки зрения практической реализации концепции полиморфизма в языке программирования C# в форме полиморфных функций особое значение для исследования имеет механизм интерфейсов.

Напомним, что реализация полиморфизма при функциональном подходе к программированию основана на оперировании функциями переменного типа.

Для иллюстрации исследуем поведение встроенной SML-функции hd (от слова «head» – голова), которая выделяет «голову» (первый элемент) списка, вне зависимости от типа его элементов. Применим функцию к списку из целочисленных элементов:

```
hd [1, 2, 3];
val it = 1: int
```

Получим, что функция имеет тип функции из списка целочисленных величин в целое число:

```
int list -> int
```

В случае списка из значений истинности та же самая функция

```
hd [true, false, true, false];
val it = true: bool
```

возвращает значение истинности, т.е. имеет следующий тип:

```
bool list -> bool
```

Наконец, для случая списка кортежей из пар целых чисел

```
hd [(1,2)(3,4),(5,6)];
val it = (1,2) : int*int
```

получим тип

```
((int*int)list -> (int*int))
```

В итоге можно сделать вывод, что функция hd имеет тип

```
(type list) -> type
```

где type – произвольный тип, т.е. функция hd полиморфна.

Проиллюстрируем сходные черты и
особенности реализации концепции полиморфизма при функциональном и объектно-ориентированном подходе к программированию следующим фрагментом программы на языке C#:

```
void Poly(object o) {
    Console.WriteLine(o.ToString());
```

```
}
```

Как видно, приведенный пример представляет собой описание полиморфной функции Poly, которая выводит на устройство вывода (например, на экран) произвольный объект o, преобразованный к строковому формату (o.ToString()).

Рассмотрим ряд примеров применения функции Poly:

```
Poly(25);
Poly("John Smith");
Poly(3.141592536m);
Poly(new Point(12,45));
```

Заметим, что независимо от типа аргумента (в первом случае это целое число 25, во втором – символьная строка "John Smith", в третьем – вещественное число π=3.141592536, в четвертом – объект типа Point, т.е. точка на плоскости с координатами (12,45)) обработка происходит единообразно и, как и в случае с языком функционального программирования SML, функция генерирует корректный результат.

Как видно из приведенных выше примеров, концепция полиморфизма одинаково применима как к функциональному, так и к объектно-ориентированному подходу к программированию. При этом целью полиморфизма является унификация обработки разнородных языковых объектов, которые в случае функционального подхода являются функциями, а в случае объектно-ориентированного – объектами переменного типа. Отметим, что для реализации полиморфизма в языке объектно-ориентированного программирования C# требуется четкое представление о ряде понятий и механизмов.

Естественно, говорить о полиморфизме можно только с учетом понятия типа. Типы определяют интерфейсы объектов и их реализацию. Переменные, функции и объекты в изучаемых в данном курсе языках программирования также рассматриваются как типизированные элементы. Необходимо также напомнить, что важное практическое значение при реализации полиморфизма в языке C# имеет механизм интерфейсов (под

которыми понимаются чисто абстрактные классы с поддержкой полиморфизма, содержащие только описания без реализации). Для реализации концепции множественного наследования необходимо принять ряд дополнительных соглашений об интерфейсах.

Сложно говорить о полиморфизме и в отрыве от концепции наследования, при принятии которой классы и типы объединяются в иерархические отношения частичного порядка из базовых классов (или, иначе, надклассов) и производных классов (или, иначе, подклассов). При этом существуют определенные различия между наследованием интерфейсов как частей, отвечающих за описания классов, и частей, описывающих правила реализации.

Еще одним значимым механизмом, сопряженным с полиморфизмом, является так называемое отложенное связывание (или, иначе, «ленивые» вычисления), в ходе которых значения присваиваются объектам (т.е. связываются с ними) по мере того как эти значения требуются во время выполнения программы.

При вычислении с вызовом по значению (call-by-value) все выражения должны быть означены до вычисления операции аппликации. Заметим, что формализация стратегии вычислений с вызовом по значению возникла в числе первых моделей computer science в виде абстрактной SECD-машины П.Лендина.

При вычислении с вызовом по имени (call-by-name) до вычисления операции аппликации необходима подстановка термов вместо всех вхождений формальных параметров до означивания. Стратегию вычислений с вызовом по значению иначе принято называть вызовом по ссылке (call-by-reference).

Наконец, при вычислении с вызовом по необходимости (call-by-need) ранее вычисленные значения аргументов сохраняются в памяти компьютера только в том случае, если необходимо их повторное использование. Именно эта стратегия лежит в основе «ленивых» (lazy), «отложенных»

(delayed) или«замороженных» (frozen) вычислений, которые принципиально необходимы для обработки потенциально бесконечных структур данных.

Рассмотрим более подробно особенности стратегии вычислений с вызовом по значению (call-by-value, CBV).

Прежде всего, в случае вызова по значению формальный параметр является копией фактического параметра и занимает выделенную область в памяти компьютера.

Кроме того, фактический параметр в случае вызова по значению является выражением.

Проиллюстрируем особенности использования стратегии вызова по значению следующим фрагментом программы на языке C#:

```
void Inc(int x) {
    x = x + 1;
}
void f() {
    int val = 3;
    Inc(val);
    // val == 3
}
```

Как видно из приведенного примера, фрагмент программы на языке C# реализует ввод значений valпосредством функции f. Функция Inc является функцией следования (т.е. прибавления единицы для целых чисел).

Отметим, что, несмотря на применение функции Inc к аргументу val, значение переменной val, как явствует из комментария

```
// val == 3,
```

остается неизменным. Это обусловлено тем, что при реализации стратегии вызова по значениюформальный параметр является копией фактического.

Рассмотрим более подробно особенности стратегии вычислений с вызовом по имени, или, иначе, по ссылке (call-by-reference, CBR).

Прежде всего, в случае вызова по имени формальный параметр является подстановкой (alias)фактического параметра и не занимает отдельной области в памяти компьютера. При реализациданной стратегии вычислений вызывающей функции передается адрес фактического параметра.

Кроме того, фактический параметр в случае вызова по имени должен быть не выражением, а переменной. При этом формальный параметр является копией фактического.

Проиллюстрируем особенности использования стратегии вызова по имени следующим фрагментом программы на языке C#:

```
void Inc(ref int x) {
    x = x + 1;
}
void f() {
    int val = 3;
    Inc(ref val);
    // val == 4
}
```

Как видно из приведенного примера, фрагмент программы на языке C# реализует преобразование значений val посредством функции f. Функция Inc является функцией следования.

Отметим, что вследствие применения функции Inc к аргументу val, значение переменной val, как явствует из комментария

```
// val == 4,
```

изменяется. Это обусловлено тем обстоятельством, что при реализации стратегии вызова по именформальный параметр является подстановкой фактического.

Рассмотрим более подробно особенности стратегии вычислений с вызовом по необходимости (call-by-need, CBN).

В целом, данная стратегия вычислений сходна с вызовом по имени (или ссылке, call-by-reference, CBR), однако ее реализация имеет две характерные особенности.

Во-первых, в случае вызова по необходимости значение фактического параметра не передается вызывающей функции, т.е. не происходит связывания переменной со значением.

Во-вторых, данная стратегия вычислений неприменима до того, как означивание может быть произведено, т.е. значение фактического параметра может быть вычислено.

Проиллюстрируем особенности использования стратегии вызова по необходимости следующим фрагментом программы на языке C#:

```
void Read (out int first, out int next) {
    first = Console.Read();
    next = Console.Read();
}
void f() {
    int first, next;
    Read(out first, out next);
}
```

Как видно из приведенного примера, фрагмент программы на языке C# реализует ввод значений first inext посредством функции Read co стандартного устройства ввода. Функция f может быть означена по необходимости, по мере поступления аргументов.

Исследовав особенности реализации различных стратегий вычислений в языке программирования C#, рассмотрим концепцию полиморфизма в соотнесении с механизмом так называемых абстрактных классов.

Абстрактные классы при объектно-ориентированном подходе (в частности, в языке программирования C#) являются аналогами полиморфных функций в языках функционального программирования (в частности, в языке SML) и используются для реализации концепции полиморфизма. Методы, которые реализуют абстрактные классы, также называются абстрактными и являются полиморфными.

Перечислим основные особенности, которыми обладают абстрактные классы и методы в рамках объектно-ориентированного подхода к программированию.

Прежде всего, отметим то обстоятельство, что абстрактные методы не имеют части реализации(implementation).

Кроме того, абстрактные методы неявно являются виртуальными (т.е. как бы оснащенными описателем virtual).

В том случае, если внутри класса имеются определения абстрактных методов, данный класс необходимо описывать как абстрактный. Ограничений на количество методов внутри абстрактного класса в языке программирования C# не существует.

Наконец, в языке программирования C# запрещено создание объектов абстрактных классов (какконкретизаций или экземпляров).

Проиллюстрируем особенности использования абстрактных классов следующим фрагментом программы на языке C#:

```
abstract class Stream {
    public abstract void
        Write(char ch);
    public void WriteString(string s)
    {
        foreach (char ch in s)
            Write(s);
    }
}

class File : Stream {
    public override void Write(char ch)
    {
        ...
        write ch to disk
        ...
    }
}
```

Как видно из приведенного примера, фрагмент программы на языке C# представляет собой описаниеабстрактных классов Stream и File, реализующих потоковую запись (метод Write) данных в форме символьных строк ch.

Заметим, что описание абстрактного класса Stream реализовано явно посредством зарезервированного слова abstract. Оператор foreach ... in реализует последовательную обработку элементов.

Необходимо обратить внимание на то обстоятельство, что поскольку в производных классах необходимо замещение методов, метод Write класса File оснащен описателем override.

Подводя итоги рассмотрения основных аспектов концепции полиморфизма в объектно-ориентированном подходе к программированию и особенностей реализации этой концепции применительно к языку программирования C#, кратко отметим достоинства полиморфизма.

Прежде всего, к преимуществам концепции полиморфизма следует отнести унификацию обработки объектов различной природы. В самом деле, абстрактные классы и методы позволяют единообразно оперировать гетерогенными данными, причем для адаптации к новым классам и типам данных не требуется реализации дополнительного программного кода.

Кроме того, важным практическим следствием реализации концепции полиморфизма для экономики программирования является снижение стоимости проектирования и реализации программного обеспечения.

Еще одно достоинство полиморфизма – возможность усовершенствования стратегии повторного использования кода. Код с более высоким уровнем абстракции не требует существенной модификации при адаптации к изменившимся условиям задачи или новым типам данных.

Важно также отметить, что идеология полиморфизма основана на строгом математическом фундаменте (в частности, в виде формальной системы

ламбда-исчисления), что обеспечивает интуитивную прозрачность исходного текста для математически мыслящего программиста, а также верифицируемость программного кода.
Наконец, концепция полиморфизма является достаточно универсальной и в равной степени применима для различных подходов к программированию, включая функциональный и объектно-ориентированный.

3.3 Реализация концепций наследования

Под наследованием в дальнейшем будем понимать свойство производного объекта сохранять поведение родительского объекта. Под поведением будем иметь в виду для математического объекта его атрибуты и операции над ним, а для языкового объекта ООП – поля и методы.

Таким образом, применительно к языку программирования концепция наследования означает, что свойства и методы базового класса равно применимы к производным от него классам. Заметим, что дочерний объект не обязательно наследует все без исключения атрибуты и операции родительского, а лишь некоторые из них. Такой подход характерен как для классов объектов в целом, так и для отдельных их конкретизаций, или, иначе, экземпляров.

Теоретическая концепция наследования удовлетворительно моделируется посредством отношения(или, точнее, иерархии) частичного порядка. Существует целый ряд формализаций наследования, но наиболее адекватными и концептуально ясными являются графические модели. Среди них следует выделить уже упомянутые ранее подходы: фреймовую нотацию Руссопулоса и диаграммы Хассе.

Отношение частичного порядка обладает следующими теоретически интересными и практически полезными свойствами.

Во-первых, оно является рефлексивным, т.е. любой объект языка программирования или формальной модели предметной области находится в отношении частичного порядка с самим собой. Формальная запись свойства рефлексивности для отношения частичного порядка ISA выглядит следующим образом:

$\forall$ a: a ISA a.

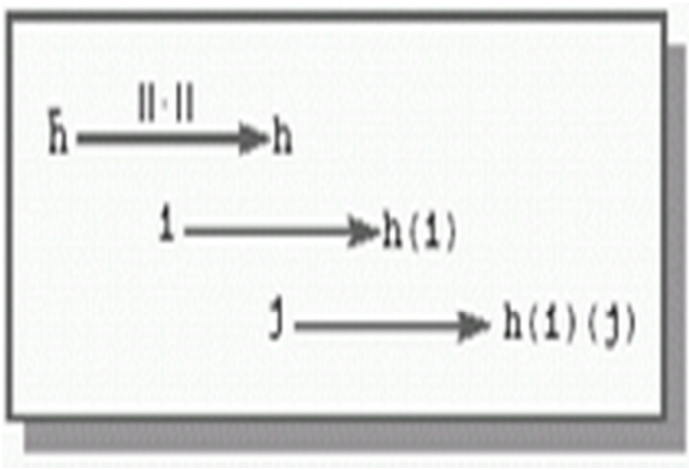
Другой важной особенностью отношения частичного порядка является транзитивность. Суть транзитивности состоит в том, что если существует цепочка из (трех) объектов, связанных отношением частичного порядка, то первый и последний элементы этой цепочки можно связать тем же отношением. Это свойство гарантирует построение на множестве графа отношения частичного порядка в виде «решетки». Формальная запись свойства транзитивности для отношения частичного порядка выглядит следующим образом:

$\forall$ a,b,c: a ISA b, b ISA c $\Rightarrow$ a ISA c.

Наконец, еще одним фундаментальным свойством отношения частичного порядка как модели наследования является анти симметричность. Это свойство разрешает наследование только в одном направлении (и запрещает его в противоположном). Формальная запись свойства анти симметричности для отношения частичного порядка выглядит следующим образом:

$\forall$ a,b : a ISA b $\Rightarrow$ NOT (b ISA a).

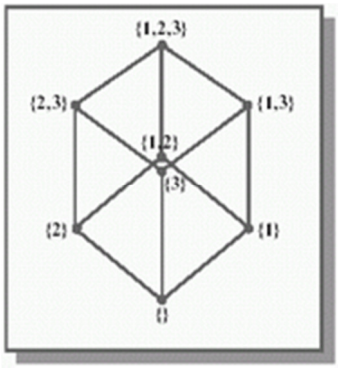
По завершении исследования свойств отношения частичного порядка, перейдем к рассмотрению формализаций, моделирующих наследование. Проиллюстрируем графическую интерпретацию отношения частичного порядка на примере фреймовой нотации Руссопулоса. Рассмотрим следующий фрейм, который связывает отношением частичного порядка понятия «сущность» (THING), «юридическое лицо» (LEGAL.PERSON), «учреждение» (INSTITUTION), «работодатель» (EMPLOYER), «кадровое агентство» (RECRUITER), «физический объект» (PHYSICAL.OBJECT), «одушевленный объект» (ANIMATE.OBJECT), «человек» (PERSON), «мужчина» (MALE.PERSON) и «женщина» (FEMALE.PERSON).



Пример фрейма.

Как видно из структуры фрейма, понятия (или, точнее, концепты) изображены в овалах. Направленные ISA-дуги соединяют понятия, образуя иерархию с направлением в сторону увеличения уровня общности (абстракции), например, от понятия «мужчина» к понятию «сущность». Рефлексивные и транзитивные дуги опущены для удобства восприятия; их без труда можно восстановить. Ориентированность дуг характеризует анти симметричность отношения частичного порядка: любая из дуг может иметь только одну стрелку со стороны увеличения уровня абстракции.

Другой продуктивной формализацией, моделирующей, в частности, наследование, является диаграмма Хассе. Проиллюстрируем использование диаграмм Хассе для графической интерпретации отношения между элементами класса или домена.



Пример диаграммы Хассе.

Диаграмма Хассе состоит из точек, которые представляют элементы множества (точнее, домена или класса), а также из соединяющих их линий,

которые представляют собой отношения между элементами класса или домена (в данном случае интерпретируется отношение частичного порядка).

Данный пример иллюстрирует отношение IS IN («является подмножеством») между множествами {},{1}, {2}, {3}, {1,2}, {1,3}, {2,3} и {1,2,3}.

Заметим, что в случае графической интерпретации отношения частичного порядка с помощьюдиаграммы Хассе свойство антисимметричности рассматриваемого отношения было бы отображено в явном виде.

Исследовав основные свойства отношения наследования и способы его наглядной формализации, рассмотрим, каким образом эта теоретическая концепция реализуется в языках объектно-ориентированного программирования, и, в частности, в языке C#.

В простейшем случае язык программирования C# поддерживает единичное наследование. Проиллюстрируем реализацию механизма единичного наследования фрагментом программы на языке C#.

```
class A {
// базовый класс
   int a;
   public A() {...}
   public void F() {...}
}

class B : A {
// подкласс (наследует свойства класса A,
// расширяет класс A)
   int b;
   public B() {...}
   public void G() {...}
}
```

Применительно к языку программирования C# наследование есть отношение между классами. Данный пример содержит описание более общего (находящегося выше по ISA-иерархии) класса A и его подкласса – класса B. Заметим, что находящийся выше по ISA-иерархии класс принято называть базовым, а находящийся ниже – производным (или подклассом).

Подкласс B наследует свойства класса A, и, кроме того, возможно, расширяет его новыми (по сравнению с классом A) свойствами.

Приведенный фрагмент программы на языке C# иллюстрирует простейший случай наследования, а именно, единичное наследование.

Рассмотрим подробнее, каким образом производится реализация механизма наследования.

Производный класс B наследует свойство a и метод F() базового класса A. Кроме того, к свойствам производного класса B добавляется новое по сравнению с базовым классом A свойство b, а к методам– новый по сравнению с базовым классом A метод G().

Заметим, что операторы конкретизации элементов базовых классов (которые в языке C# называют конструкторами) не наследуются производными классами. Отметим также, что некоторые из наследуемых методов могут игнорироваться. В ходе единичного наследования производный подкласс может наследовать свойства единственного базового класса. Множественное наследование, демонстрирующее гибкость механизмов наследования в языке C#, реализуется на основе интерфейсов. Хотя производный класс в языке программирования C# может наследовать свойства базового класса, он не может наследовать свойств структуры. Реализация механизма наследования (в том числе множественного) для структур в языке программирования C# осуществляется (как и в случае с классами) посредством интерфейсов.

В случае неявного задания базового класса производный класс наследует свойства наиболее общего класса Microsoft .NET, известного как объект (object) и аналогичного концепту THING («сущность») в рассмотренном ранее примере фреймовой нотации.

Для иллюстрации множественного наследования приведем фрагмент программы на языке C#, содержащей описание интерфейса:

```
public interface IList :
   ICollection, IEnumerable
{
   int Add (object value);
   // методы
   bool Contains (object value);
   bool IsReadOnly {
      get;
```

```
    }
    // свойство
    object this [int index]{
       get;
       set;
    }
    // индексатор
}
```

Из приведенного примера видно, что фрагмент программы на языке C# представляет собой описание общедоступного интерфейса IList, наследующего в иерархии элементы интерфейсов ICollection иIEnumerable, а также содержащего методы Add и Contains, свойство IsReadOnly и объект-индексатор.

После визуального знакомства с кодом на языке программирования C#, описывающим интерфейс, приведем более общее словесное определение этого механизма, имеющего принципиальное значение.

Интерфейсом называется чисто абстрактный класс (с поддержкой полиморфизма), содержащий только описания без реализации.

Концепция множественного наследования предполагает возможность наследования одним концептом языка свойств сразу нескольких языковых концептов. При этом в языке программирования C# принципиально допустимо множественное наследование, но область действия его ограничена интерфейсами. Множественное наследование классов в языке программирования C# недопустимо, однако неявно реализуемо посредством интерфейсов. Так, классы и структуры языка программирования C# могут реализовывать множественные интерфейсы.

Интерфейсы могут содержать методы, свойства, индексаторы и события. Однако интерфейсы не могут содержать полей, констант, конструкторов, деструкторов, операторов, а также вложенных типов.

Элементы интерфейса неявно являются виртуальными (т.е. общедоступными и абстрактными) объектами и описываются в языке C# как public abstract (или virtual).

Интерфейсы, будучи принципиально полиморфными (т.е. динамически означиваемыми) объектами, не могут содержать статических элементов.

Наконец, свойства одного интерфейса могут быть расширены посредством другого интерфейса.

Рассмотрим более подробно важнейшие свойства интерфейсов как механизма реализации концепции и множественного наследования в языке программирования C#.

Как уже отмечалось, концепция множественного наследования в языке программирования C# неприменима в явной форме для классов. В случае классов существует возможность только единичного наследования, т.е. производный класс может наследовать свойства единственного базового класса. Механизм интерфейсов, таким образом, представляет собой неявную возможность реализации концепции множественного наследования для языка программирования C#, поскольку для классов в языке допустима реализация множественных интерфейсов.

В отношении наследования структур в языке программирования C# наблюдается определенное сходство с классами. В частности, несмотря на то обстоятельство, что структуры не имеют возможности наследовать свойства типов, для них, как и для классов, допустима реализация множественных интерфейсов.

При этом произвольный элемент интерфейса, будь то метод, свойство или индексатор, должен непременно быть либо реализован непосредственно в базовом классе, либо унаследован от него.

Кроме того, заметим, что реализованные в языке программирования C# методы интерфейса могут быть описаны либо как виртуальные (virtual), либо как абстрактные (abstract). Таким образом, интерфейс может быть реализован посредством абстрактного класса. В то же время не допускается реализация замещенных интерфейсов по аналогии с замещенными в производных классах методами, которые описываются посредством ключевого слова override языка программирования C#.

Концепция наследования, рассмотренная нами применительно к объектно-ориентированному подходу к программированию, вполне может быть распространена и на случай систем и сред, поддерживающих проектирование и реализацию программного обеспечения.

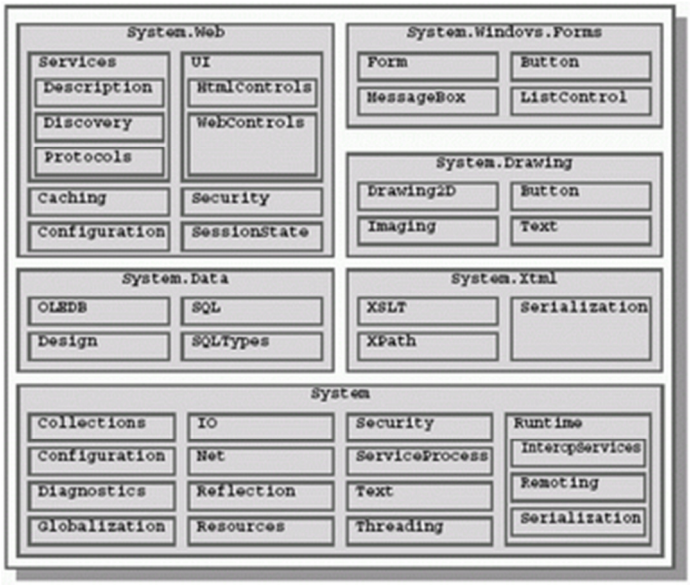
В этой связи не является исключением и среда интегрированной разработки приложений Microsoft .NET, классы которой не имеют существенных различий с классами языка программирования C#.

Как видно из схемы, описывающей основные пространства имен Microsoft .NET Framework, организация их является вполне иерархической и реализует классическую схему единичного наследования.

Наиболее общим концептом иерархии является пространство имен System, характеризующее конфигурацию среды Microsoft .NET Framework и содержащее, в частности, параметры среды выполнения приложений, удаленной обработки данных, процессов, безопасности, ввода-вывода, системной конфигурации и др.

Среди подпространств пространства имен System можно выделить такие пространства имен, какSystem.Web, System.Windows.Forms, System.Data, System.Drawing и System.Xml, которые описывают такие характеристики среды Microsoft .NET Framework, как конфигурации веб-сервисов, формы ввода-вывода данных, форматы представления данных, графических подсистем и т.д.

Данная схема иллюстрирует перечень классов Common Type System среды проектирования и реализации программного обеспечения Microsoft .NET применительно к языку программирования C#.



Иерархия классов .NET Framework.

В частности, все многообразие типов можно разделить на предопределенные (заранее заданные системой программирования) и определенные пользователем (user defined) типы.

К последним относятся перечисления, массивы, классы, интерфейсы и делегаты (указатели на функцию).

Предопределенные типы делятся на ссылочные типы (объекты и символьные строки) и типы-значения (встроенные – короткие и длинные целые со знаком и без знака, а также числа с плавающей точкой – с одинарной и двойной точностью).

Напомним, что все без исключения типы языков программирования SML и C# однозначно отображаются в систему типизации Microsoft .NET, верхним иерархическим элементом которой является пространство имен System.

Подводя итоги обсуждения наследования, одной из основополагающих концепций объектно-ориентированного подхода к программированию, кратко резюмируем положительные аспекты явления и вытекающие из них практические возможности.

Прежде всего, концепция наследования устраняет необходимость многократного явного указаниясвойств и методов для производных объектов. Одного взгляда на схему пространств имен Microsoft .NET Framework достаточно, чтобы понять значимость этого свойства. Заметим, что на схеме

показана лишь часть верхних «этажей» многоуровневой иерархии Кроме того, следует отметить, что наследование открывает возможности для гибкого определенияуровня абстракции предметной области. Можно оперировать единственным, достаточно абстрактным, концептом System (или THING, и, спускаясь по ISA-иерархии, обращаться к терминам все более конкретных концептов и сущностей (или, в терминологии языка программирования C#, классов и объектов).

Таким образом, концепция наследования предоставляет возможность моделирования сколь угодно сложной предметной области посредством ISA-иерархии концептов (или классов языка программирования C#), моделирующих объекты реального мира.

Итак, наследование прозрачный и унифицированный подход к построению классов (которые в терминологии языка программирования C# принято называть производными) по заданной совокупности свойств и методов так называемых базовых классов.

Заметим, что в языке программирования C# допустимо наследование свойств и методов как от единственного класса (единичное), так и от нескольких классов (множественное). Последний вид наследования реализуется посредством механизма интерфейсов.

3.4 Реализация концепций инкапсуляции

Формализуем понятие инкапсуляции в рамках объектно-ориентированного подхода к программированию.

В неформальной постановке вопроса под инкапсуляцией будем понимать доступность объекта исключительно посредством его свойств и методов.

Другими словами, концепция инкапсуляции призвана обеспечивать безопасность проектирования и реализации программного обеспечения на основе локализации манипулирования объектом в областях его полей и методов.

Иначе говоря, свойствами объекта можно оперировать исключительно посредством его методов. Это замечание касается как свойств, явно определенных в описании объекта, так и свойств, унаследованных данным объектом от другого (других).

Практическая важность концепции инкапсуляции для современных языков объектно-ориентированного программирования (в том числе и для языка C#) определяется следующими фундаментальными свойствами.

Прежде всего, реализация концепции инкапсуляции обеспечивает совместное хранение данных (или, иначе, полей) и функций (или, иначе, методов) внутри объекта.

Как следствие, механизм инкапсуляции приводит к сокрытию информации о внутреннем «устройстве» объекта данных (или, в терминах языков ООП, свойств и методов объекта) от пользователя того или иного объектно-ориентированного приложения.

Таким образом, пользователь, получающий программное обеспечение как сервис, оказывается изолированным от особенностей среды реализации.

Рассмотрев определение понятия инкапсуляции (пока на интуитивном уровне), перейдем к описанию формализаций этой фундаментальной для объектно-ориентированного программирования концепции на основе уже известных нам формальных теорий computer science.

Оказывается, что понятие инкапсуляции вполне адекватно формализуется посредством таких теоретических формальных систем, как ламбда-исчисление А.Черча и комбинаторная логика Х.Карри.

При этом при интерпретации концепции инкапсуляции в терминах формальной системы ламбда-исчисления, в роли объектов выступают ламбда-термы, в роли свойств – связанные переменные, а в роли методов – свободные переменные.

В случае же интерпретации концепции инкапсуляции в терминах формальной системы комбинаторной логики в роли объектов выступают

комбинаторы, в роли свойств – переменные, а в роли методов – комбинаторы.

Поясним более подробно реализацию механизма сокрытия информации посредством концепцииинкапсуляции.

Рассмотрим в достаточно обобщенном виде схему взаимодействия объекта и данных.

Вначале представим схему такого рода для традиционного императивного языка программирования (например C и Pascal). При этом в нашем рассуждении под термином «объект» будем понимать произвольный объект языка программирования, безотносительно к концепции объектно-ориентированного программирования.

В этом случае объявления данных и процедуры обработки данных отделены друг от друга. Зачастую в ранних языках программирования описания языковых объектов и процедуры манипулирования ими выделены в особые разделы.

В этой связи принципиальным недостатком ранних императивных языков программирования является то обстоятельство, что доступ к данным может быть получен методами, которые изначально не предназначались разработчиками приложений для манипулирования этими данными. Вообще говоря, управление объектом может осуществляться посредством произвольной процедуры или функции, и у среды проектирования и реализации программного обеспечения нет возможности централизованного управления этим процессом. Такой подход к программированию, безусловно, является довольно непоследовательным и весьма небезопасным.

В отличие от предыдущей схемы, при объектно-ориентированном подходе к проектированию и реализации программного обеспечения взаимодействие объектов языка программирования и конкретизирующих его данных реализуется принципиально иным образом. По существу, объект и данные составляют единое и неделимое целое.

Прежде всего, определение (или описание свойств классов) и процедуры манипулирования этимисвойствами (или методы) для каждого объекта языка программирования при объектно-ориентированном подходе хранятся совместно.

Кроме того, среда проектирования и реализации программного обеспечения (например, Microsoft Visual Studio .NET) не предоставляет иных возможностей доступа к объекту, кроме как посредством методов, изначально предназначенных для манипулирования данным объектом.

Таким образом, концепция инкапсуляции в языках объектно-ориентированного программирования служит целям обеспечения единообразия определения, хранения и работы с информацией о языковых объектах.

Заметим, что инкапсуляция является безусловно необходимым требованием для каждого объекта.

В то же время в практике программирования степень инкапсуляции объекта (в широком значении этого слова) определяется его описанием, а также описанием порождающих его объектов (в действительности это, как правило, описания базовых и производных классов).

В этой связи в языках объектно-ориентированного программирования вводится понятие области видимости как степени доступности произвольного языкового объекта.

Применительно к языку программирования C# области видимости объектов подразделяются на следующие виды.

Общедоступные объекты описываются с помощью зарезервированного слова public и характеризуются доступностью из произвольного места программы, для которого определено пространство имен с описанием рассматриваемого объекта. При этом элементы интерфейсов и перечислений являютсяобщедоступными (public) объектами по умолчанию.

С другой стороны, типы, описанные в составе пространств имен в форме классов, структур, интерфейсов, перечислений или делегатов являются по

умолчанию видимыми из сборки с описанием объекта и описываются с помощью зарезервированного слова internal.

Наконец, элементы классов и структур, в частности, поля, методы, свойства и вложенные типы являются по умолчанию доступными из описаний соответствующих классов или структур и описываются с помощью зарезервированного слова private.

Проиллюстрируем обсуждение использования модификаторов областей видимости объектов (public иprivate) следующим содержательным примером фрагмента программы на языке C#:

```
public class Stack{
    private int[] val;
    // private используется
    // и по умолчанию
    private int top;
    // private используется
    // и по умолчанию
    public Stack(){
        ...
    }
    public void Push(int x){
        ...
    }
    public int Pop(){
        ...
    }
}
```

Как видно из приведенного примера, фрагмент программы на языке C# содержит описание класса стека Stack, реализованного на основе массива целочисленных элементов (поле val). Голова стека представляет собой целочисленное значение (поле top). Над стеком определены операцииинициализации (метод Stack), а также вставки (метод Push) и удаления (метод Pop) элемента.

Как явствует из примера, класс Stack и все манипулирующие его объектами методы (Stack, Push и Pop) являются общедоступными (public), тогда как поля являются доступными локально (private), т.е. только из описания данного класса.

Заметим, что в приведенном выше примере фрагмента программы с описанием областей видимостииспользовались только базовые возможности модификаторов видимости языка программирования C#.

Оказывается, что язык программирования C# располагает механизмами реализации дополнительных (или расширенных) по сравнению с базовыми областей видимости объектов.

Рассмотрим более подробно особенности основных типов расширенных областей видимости объектов в языке программирования C#.

К числу расширенных областей видимости следует отнести доступность языковых объектов как непосредственно из класса с описанием объекта, так и из его производных классов. В данном случае для описания языкового объекта используется зарезервированное слово protected.

Кроме того, для описания доступности языкового объекта лишь из сборки с его описанием используется зарезервированное слово internal.

Наконец, для описания доступности языкового объекта непосредственно из класса с описанием данного объекта, его производных классов, а также из сборки с описанием данного объекта используется зарезервированное слово protected internal.

Проиллюстрируем обсуждение использования модификаторов расширенных областей видимостиобъектов (protected и internal) следующим содержательным примером фрагмента программы на языке C#:

```
class Stack {
    protected int[] values =
        new int[32];
    protected int top = -1;
    public void Push(int x) {
        ...
    }
    public int Pop() {
        ...
    }
}
class BetterStack : Stack {
    public bool Contains(int x) {
        foreach (int y in values)
            if(x==y)
              return true;
            return false;
        }
}
class Client {
    Stack s = new Stack();
    ...
    s.values[0];
    ...
    // ошибка при компиляции!
```

```
}
```

Как видно из приведенного примера, фрагмент программы на языке C# содержит описание класса стека Stack (для хранения 32 целочисленных элементов и значением -1 в вершине), а также реализованного на его основе усовершенствованного класса стека BetterStack, дополнительно реализующего повторяющиеся элементы стека. В отличие от предыдущего примера, все поля класса стека Stack доступны как из данного класса, так и из классов, производных от него, поскольку описаны посредством ключевого слова protected.

После обсуждения использования (степеней) инкапсуляции применительно к классам в целом, рассмотрим особенности приложения данной концепции к таким фундаментальным объектам языка программирования C# как поля и константы.

Инициализация не является безусловно необходимой для полей в языке программирования C#. Тем не менее, доступ к полям и методам изначально запрещен (что соответствует использованию по умолчанию модификатора доступа private). В случае структуры поля инициализации не подлежат.

Простейшей иллюстрацией описания поля в языке программирования C# является следующий пример, содержащий определение класса C с целочисленным полем value:

```
class C {
    int value = 0;
}
```

В случае константы инициализация также не является необходимым требованием. Тем не менее, значение константы должно быть вычислимым в процессе компиляции.

Простейшей иллюстрацией описания константы в языке программирования C# является следующий пример, содержащий определение константы size, представляющей собой целочисленное значение двойной длины (long):

```
const long size = ((long)int.MaxValue+1)/4;
```

Заметим, что фрагмент

```
...(long) ...
```

в правой части присваивания представляет собой явное преобразование типов языковых объектов.

Особым случаем реализации концепции инкапсуляции в языке объектно-ориентированного программирования C# является механизм полей, предназначенных только для чтения. Для описания такого рода полей в языке C# используется зарезервированное слово readonly.

Основные свойства полей, предназначенных только для чтения, сводятся к следующему. Прежде всего, такие поля необходимо инициализировать непосредственно при описании или в конструкторе класса. Кроме того, значение поля, предназначенного только для чтения, не обязательно должно быть вычислимым в ходе компиляции, а может быть вычислено во время выполнения программы. Наконец,поля, предназначенные только для чтения, занимают предназначенные для них области памяти (что до определенной степени аналогично статическим объектам).

Проиллюстрируем основные свойства полей, предназначенных только для чтения, следующими примерами фрагментов программ на языке C#. Описание поля выглядит следующим образом:

```
readonly DateTime date;
```

Доступ к полю изнутри класса организуется по краткому имени объекта:

```
... value ... size ... date ...
```

Доступ к полю из других классов (на примере объекта с как конкретизации класса C) реализуется с указанием полного имени объекта:

```
c = new C();
... c.value ... c.size ... c.date ...
```

Рассмотрим подробнее особенности реализации механизмов доступа к статическим полям иконстантам в языке программирования C#.

Прежде всего, необходимо отметить следующее фундаментальное положение, характеризующеестатические поля и константы. Поскольку изменения свойств объектов, динамических по своей природе и порожденных динамическими классами, не затрагивают статических полей и констант, последние принадлежат классам, а не объектам.

Проиллюстрируем это высказывание следующим фрагментом программы на языке C#:

```
class Rectangle {
    static Color defaultColor;
    //однократно для класса

    static readonly int scale;
    //однократно для класса
    //статические константы
    //недопустимо использовать

    int x, y, width,height;
    //однократно для объекта
    ...
}
```

Как видно из приведенного примера, при описании класса Rectangle со статическими полями Color iscale и динамическими полями x, y, width и height, статическими являются поля, инвариантные по отношению к классу, а динамическими – представляющие собой «неподвижную точку» относительно объекта.

Доступ к статическим полям изнутри класса по краткому имени и из других классов по полному имени соответственно реализуется по аналогии с полями только для чтения:

```
... defaultColor ... scale ...
... Rectangle.defaultColor ...
    Rectangle.scale ...
```

Продолжим иллюстрацию механизмов реализации концепции инкапсуляции в языке программирования C# следующим примером фрагмента программы:

```
class C {
    int sum = 0, n = 0;
    public void Add (int x){
        // процедура
        sum = sum + x; n++;
    }
    public float Mean() {
        // функция
        // (возвращает значение)
        return (float)sum / n;
    }
}
```

Приведенный пример представляет собой описание класса C, содержащего целочисленные поля sum un, а также методы Add и Mean для суммирования и вычисления среднего значения, реализованные в форме процедуры и функции, соответственно.

Заметим, что методы Add и Mean класса C описаны как общедоступные и, следовательно, могут быть доступны из любого места программного проекта при указании полного имени, тогда как поля sum и n, в силу умолчаний, принятых в языке программирования C#, являются доступными локально (private).

Заметим попутно, что функция (в терминологии языка C#) отличается от процедуры тем, что обязательно возвращает значение. Для определения процедуры, не возвращающей значения, в C# используется зарезервированное слово void.

Проиллюстрируем сказанное об областях видимости объектов языка программирования C# фрагментами программ.

Предположим, что в предыдущем примере, описывающем общедоступный класс C, содержащий целочисленные поля sum и n, а также методы Add и Mean, необходимо организовать доступ к элементам класса изнутри, а также из других классов.

Очевидно, что в силу общедоступности реализации класса C такое задание принципиально осуществимо.

При реализации доступа к элементам класса C изнутри данного класса достаточно указать краткие имена объектов:

```
this.Add(3);
float x = Mean();
```

В то же время, при реализации механизма доступа к элементам класса C из других классов (внешних по отношению к данному) необходимо указывать полные имена объектов:

```
C c = new C();
c.Add(3);
float x = c.Mean();
```

Манипулирование статическими полями и методами в языке программирования C# осуществляется аналогично рассмотренным выше случаям статических полей и констант.

Например, в случае, если класс, содержащий статические элементы данных, определяется посредством следующего описания:

```
class Rectangle {
    static Color defaultColor;
    public static void ResetColor(){
        defaultColor = Color.white;
    }
}
```

доступ к элементам данных изнутри класса осуществляется посредством указания краткого имениобъекта класса, например:

```
ResetColor();
```

а доступ к элементам данных из других классов – посредством указания полного имени объекта класса, например:

```
Rectangle.ResetColor();
```

По результатам исследования концепции инкапсуляции и ее применения в языках объектно-ориентированного программирования можно сделать следующие выводы о принципиальных преимуществах рассмотренной концепции.

Прежде всего, реализация концепции инкапсуляции приводит к унификации представления, а следовательно, и моделирования сколь угодно сложных предметных областей.

Кроме того, механизм инкапсуляции обеспечивает четкий, недвусмысленный, прямолинейный подход к моделированию предметной области за счет объединения объектов с данными.

Именно в силу последнего обстоятельства становится принципиально возможной реализация гибкого управления уровнем абстракции как для данных, так и для метаданных.

Наконец, концепция инкапсуляции гарантирует более высокую степень безопасности результирующего кода и его защищенности от несанкционированных действий или случайных ошибок пользователя.

Глава 4.

Безопасность жизнедеятельности

Безопасность жизнедеятельности - это комплекс мероприятий, направленных на обеспечение безопасности человека в среде обитания, сохранение его здоровья, разработку методов и средств защиты путем снижения влияния вредных и опасных факторов до допустимых значений, выработку мер по ограничению ущерба в ликвидации последствий чрезвычайных ситуации.

Охрана здоровья трудящихся, обеспечение безопасности условий труда, ликвидация профессиональных заболеваний и производственного травматизма составляет одну из главных забот человеческого общества. Обращается внимание на необходимость широкого применения прогрессивных форм научной организации труда, сведения к минимуму ручного, малоквалифицированного труда, создания обстановки, исключающей профессиональные заболевания и производственный травматизм.

На рабочем месте должны быть предусмотрены меры защиты от возможного воздействия опасных и вредных факторов производства. Уровни этих факторов не должны превышать предельных значений, оговоренных правовыми, техническими и санитарно-техническими нормами. Эти нормативные документы обязывают к созданию на рабочем месте условий труда, при которых влияние опасных и вредных факторов на работающих либо устранено совсем, либо находится в допустимых пределах.

4.1 Характеристика условий труда программиста

В настоящее время компьютерная техника широко применяется во всех областях деятельности человека. При работе с компьютером человек подвергается воздействию ряда опасных и вредных производственных

факторов: электромагнитных полей (диапазон радиочастот: ВЧ, УВЧ и СВЧ), инфракрасного и ионизирующего излучений, шума и вибрации, статического электричества и др.

Работа с компьютером характеризуется значительным умственным напряжением и нервно-эмоциональной нагрузкой операторов, высокой напряженностью зрительной работы и достаточно большой нагрузкой на мышцы рук при работе с клавиатурой ЭВМ. Большое значение имеет рациональная конструкция и расположение элементов рабочего места, что важно для поддержания оптимальной рабочей позы человека-оператора.

В процессе работы с компьютером необходимо соблюдать правильный режим труда и отдыха. В противном случае у персонала отмечаются значительное напряжение зрительного аппарата с появлением жалоб на неудовлетворенность работой, головные боли, раздражительность, нарушение сна, усталость и болезненные ощущения в глазах, в пояснице, в области шеи и руках.

4.2 Параметры микроклимата

Параметры микроклимата могут меняться в широких пределах, в то время как необходимым условием жизнедеятельности человека является поддержание постоянства температуры тела благодаря терморегуляции, т.е. способности организма регулировать отдачу тепла в окружающую среду. Принцип нормирования микроклимата – создание оптимальных условий для теплообмена тела человека с окружающей средой.

Вычислительная техника является источником существенных тепловыделений, что может привести к повышению температуры и снижению относительной влажности в помещении. В помещениях, где установлены компьютеры, должны соблюдаться определенные параметры микроклимата. В санитарных нормах СаНПиН 2.2.4.548-96 «Гигиена труда и

микроклимата помещений», установлены величины параметров микроклимата, создающие комфортные условия..

Объем помещений, в которых размещены работники вычислительных центров, не должен быть меньше 19,5м^3/человека с учетом максимального числа одновременно работающих в смену. Нормы подачи свежего воздуха в помещения, где расположены компьютеры.

Для обеспечения комфортных условий используются как организационные методы (рациональная организация проведения работ в зависимости от времени года и суток, чередование труда и отдыха), так и технические средства (вентиляция, кондиционирование воздуха, отопительная система).

В нашем случае обеспечивать комфортные условия работы специалиста будет кондиционер. Кондиционер – это автоматизированная вентиляционная установка, которая поддерживает в помещении заданные параметры микроклимата.

4.3 Шум и вибрация

Шум ухудшает условия труда оказывая вредное действие на организм человека. Работающие в условиях длительного шумового воздействия испытывают раздражительность, головные боли, головокружение, снижение памяти, повышенную утомляемость, понижение аппетита, боли в ушах и т. д. Такие нарушения в работе ряда органов и систем организма человека могут вызвать негативные изменения в эмоциональном состоянии человека вплоть до стрессовых. Под воздействием шума снижается концентрация внимания, нарушаются физиологические функции, появляется усталость в связи с повышенными энергетическими затратами и нервно-психическим напряжением, ухудшается речевая коммутация. Все это снижает работоспособность человека и его производительность, качество и

безопасность труда. Длительное воздействие интенсивного шума [выше 80 дБ(А)] на слух человека приводит к его частичной или полной потере.

Уровень шума на рабочем месте математиков-программистов и операторов видеоматериалов не должен превышать 50дБА, а в залах обработки информации на вычислительных машинах - 65дБА. Для снижения уровня шума стены и потолок помещений, где установлены компьютеры, облицовываются звукопоглощающими материалами.

4.4 Требования к рабочему месту

Рабочее место и взаимное расположение всех его элементов должно соответствовать антропометрическим, физическим и психологическим требованиям. Большое значение имеет также характер работы. В частности, при организации рабочего места программиста соблюдаются следующие основные условия: оптимальное размещение оборудования, входящего в состав рабочего места и достаточное рабочее пространство, позволяющее осуществлять все необходимые движения и перемещения.

Главными элементами рабочего места программиста являются стол и кресло. Основным рабочим положением является положение сидя.

Рабочая поза сидя вызывает минимальное утомление программиста. Рациональная планировка рабочего места предусматривает четкий порядок и постоянство размещения предметов, средств труда и документации. То, что требуется для выполнения работ чаще, расположено в зоне легкой досягаемости рабочего пространства.

Моторное поле - пространство рабочего места, в котором осуществляются двигательные действия человека.

Максимальная зона досягаемости рук - это часть моторного поля рабочего места, ограниченного дугами, описываемыми максимально вытянутыми руками при движении их в плечевом суставе.

Оптимальная зона - часть моторного поля рабочего места, ограниченного дугами, описываемыми предплечьями при движении в локтевых суставах с опорой в точке локтя и с относительно неподвижным плечом.

Документация, необходимая при работе - в зоне легкой досягаемости ладони – **в**, а в выдвижных ящиках стола - литература, неиспользуемая постоянно.

4.5 Противопожарная безопасность

Пожар может возникнуть в любом помещении. Для тушения пожара, а также для обеспечения безопасности работников на предприятиях, должны быть предусмотрены определенные средства пожаротушения.

Аппараты пожаротушения подразделяют на стационарные установки и огнетушители (ручные до 10 л. и передвижные или стационарные объемом свыше 25 л.).

Стационарные установки предназначены для тушения пожаров в начальной стадии их возникновения без участия людей. Их монтируют в зданиях и сооружениях, а также для защиты наружных технологических установок. По применяемым огнетушащим средствам их подразделяют на водные, пенные, газовые, порошковые и паровые. Стационарные установки могут быть автоматическими и ручными с дистанционным пуском.

Огнетушители по виду огнетушащих средств подразделяют на жидкостные, углекислотные, химпенные, воздушно–пенные, хладоновые, порошковые и комбинированные.

В качестве такого средства пожаротушения можно выбрать химический ОХП-10 и углекислотные ОУ-2, ОУ-3 ОУ-5, ОУ-8

огнетушители, которые применяются для тушения пожаров электроустановок, находящихся под напряжением.

Заключение

В этой выпускной квалификационной работе я попытался раскрыть тему «Эволюция технологий и языков программирования» и представить актуальность этой темы в современной жизни. В главах выше особое внимание нужно обратить на главы 2, 3, 4, 7 так как там представлена основная информация доказывающая нужной данной темы в современной мире.

Информационные технологии прочно вошли в нашу жизнь. Применение ЭВМ стало обыденным делом, хотя совсем ещё недавно рабочее место, оборудованное компьютером, было большой редкостью. Информационные технологии открыли новые возможности для работы и отдыха, позволили во многом облегчить труд человека. Современное общество вряд ли можно представить без информационных технологий. Перспективы развития вычислительной техники сегодня сложно представить даже специалистам. Однако, ясно, что в будущем нас ждет нечто грандиозное. И если темпы развития информационных технологий не сократятся, то это произойдет очень скоро.
С развитием информационных технологий растет прозрачность мира, скорость и объемы передачи информации между элементами мировой системы, появляется еще один интегрирующий мировой фактор. Это означает, что роль местных традиций, способствующих самодостаточному инерционному развитию отдельных элементов, слабеет. Одновременно усиливается реакция элементов на сигналы с положительной обратной связью. Интеграцию можно было бы только приветствовать, если бы ее следствием не становилось размывание региональных и культурно-исторических особенностей развития.
Информационные технологии вобрали в себя лавинообразные достижения электроники, а также математики, философии, психологии и экономики.

Образовавшийся в результате жизнеспособный гибрид ознаменовал революционный скачок в истории информационных технологий, которая насчитывает сотни тысяч лет. Современное общество наполнено и пронизано потоками информации, которые нуждаются в обработке. Поэтому без информационных технологий, равно как без энергетических, транспортных и химических технологий, оно нормально функционировать не может. Социально-экономическое планирование и управление, производство и транспорт, банки и биржи, средства массовой информации и издательства, оборонные системы, социальные и правоохранительные базы данных, сервис и здравоохранение, учебные процессы, офисы для переработки научной и деловой информации, наконец, Интернет - всюду ИТ. Информационная насыщенность не только изменила мир, но и создала новые проблемы, которые не были предусмотрены.

Размышляя над этим, хочется верить в прогресс науки и техники, в высоко - компьютеризированное будущее человечества, как единственного существа на планете, пусть и не использующего один, определенный разговорный язык, но способного так быстро прогрессировать и развивать свой интеллект, что и перехода от многоязыковой системы к всеобщему пониманию долго ждать не придется.

В Узбекистане идет очень бурный рост и развитие информационно-коммуникационных технологий, электронной коммерции и сферы услуг.

Под руководством Президента Ислама Каримова был разработан ряд законов, указов и постановлений:

- Указ президента республики узбекистан "о дальнейшем развитии компьютеризации и внедрении информационно-коммуникационных технологий" от 30.05.2002г.
- Закон республики узбекистан от 11.12.2003 г., № 562-ii об электронной

цифровой подписи.

- Закон республики узбекистан от 29.04.2004 г. N 613-ii об электронной коммерции.
- Постановление президента республики узбекистан "о дополнительных мерах по дальнейшему развитию икт" от 08.07.2005г.
- Закон республики узбекистан от 16.04.2006 года № зру – 13, об электронных платежах.
- Постановление кабинета министров республики узбекистан от 01.02.2012 г. N 24 - о мерах по созданию условий для дальнейшего развития компьютеризации и информационно-коммуникационных технологий на местах.
- Постановление президента республики узбекистан от 21.03.2012 г. N пп-1730 - о мерах по дальнейшему внедрению и развитию современных информационно- коммуникационных технологий.
- Постановление президента республики узбекистан от 10.05.2012 г. N пп-1754 - о программе развития сферы услуг в республике узбекистан на 2012-2016 годы.

Список литературы

1. Роберт В Себеста «Основные концепции языков программирования» 2001 г. 672 с.
2. C++, TurboPasckal, QBasik:Эволюция языков программирования
3. Курносов А.П.,Улезько А.В. и др.; Под ред. А.П. Курносова.-М.: КолосС,2005.-207 с
4. Малышев Р.А. Локальные вычислительные сети: Учебное пособие/ РГАТА.- Рыбинск,2005.-83 с.
5. Островский В.А. Информатика: учеб.для вузов. М: Высшая школа,2000.-511 с: ил.
6. Семакин И.А., Информатика: Базовый курс/ Семакин И.А., ЗалоговаЛ.,Русаков С., Шестакова Л.- Москва : БИНОМ., 2005.-105с.
7. Симинович С.В. Информатика .Базовый курс/ Симонович С.В. и др.- СПБ. Издательство « Питер», 2000.-640 с.ил.
8. Nancy Stern,RobertA.Stern,JamesP.Ley - COBOL for the 21st Century
9. Городняя Л.В. Основы функционального программирования. /– М.: Изд-во "Интернет-университет информационных технологий – ИНТУИТ.ру", 2004. – 280 с.: ил. ISBN 5–9556–0008–6
10. Анатолий А. Андрей К. Логическое программирование и Visual Prolog (с CD).. – СПб.: «БХВ–Петербург», 2003. – С. 990. ISBN 5–94157–156–9.
11. Иан Грэхем. Объектно–ориентированные методы. Принципы и практика = Object–Oriented Methods: Principles & Practice. – 3–е изд./ – М.: «Вильямс», 2004. – С. 880. ISBN 5–8459–0438–2
12. Андрей Александреску. Современное проектирование на C++ /Вильямс, 2004 г. 336 стр. Тираж: 3500 экз. ISBN 5–8459–0351–3
13. Легалов А. И. – SoftCraft: разработка трансляторов: конспект лекций
14. А.Ю.Гаевский. «Информатика». Киев: «А.С.К.», 2006 г.
15. А.Г.Гейн. «Основы информатики и вычислительной техники». Москва: «Просвещение», 1999 г.

16. В.Ф.Ляхович. «Основы информатики». Ростов-на-Дону: «Феникс», 1996 г.
17. А.С.Ваулин. «Языки программирования», кн.5, 1993 г.
18. В.Б.Попов. «Паскаль и Дельфи. Учебный курс», Спб.: Питер, 2005 г.
19. С.Н.Лукин. «Турбо-Паскаль 7.0. Самоучитель для начинающих», М.: «Диалог-МИФИ», 2004 г.
20. О.П.Зеленяк. «Практикум программирования на Turbo Paskal. Задачи, алгоритмы и решения», СПб.: ООО «ДиаСофтЮП», 2002 г.
21. Н.Б.Культин. «Основы программирования в Turbo Delphi»,СПб.: БХВ-Петербург, 2007 г.
22. Дейкстра Э. Дисциплина программирования — 1-е изд. — М.: Мир, 1978.— ISBN;
23. Антони Синтес Освой самостоятельно объектно-ориентированное программирование за 21 день = Sams Teach Yourself Object-Oriented Programming in 21 Days. — М.:«Вильямс», 2002.— ISBN 0-672-32109-2;
24. Н. А. Роганова Функциональное программирование: Учебное пособие для студентов высших учебных заведений — М.: ГИНФО, 2002.;
25. Еженедельник "Computerworld", №29, 2000 год // Издательство "Открытые Системы";
26. *Иан Грэхем.* Объектно-ориентированные методы. Принципы и практика = Object-Oriented Methods: Principles & Practice. — 3-е изд. — М.: «Вильямс», 2004. — С. 880. — ISBN 0-201-61913-X
27. *Антони Синтес.* Освой самостоятельно объектно-ориентированное программирование за 21 день = Sams Teach Yourself Object-Oriented Programming in 21 Days. — М.: «Вильямс», 2002. — С. 672. — ISBN 0-672-32109-2
28. *Matt Weisfeld* The Object-Oriented Thought Process. — Fourth Edition. — Addison-Wesley Professional, 2013. — 336 с. — ISBN 978-0-321-86127-6
29. Указ президента республики узбекистан "о дальнейшем развитии компьютеризации и внедрении информационно-коммуникационных технологий" от 30.05.2002г. - http://eduportal.uz/rus/info/information/zakonodat/ukaz3080/

30. Закон республики узбекистан от 11.12.2003 г., № 562-ii об электронной цифровой подписи. - http://cbu.uz/ru/banking_legislation/ruz_laws/electronic_signature.htm
31. Закон республики узбекистан от 29.04.2004 г. n 613-ii об электронной коммерции. - http://cbu.uz/ru/banking_legislation/ruz_laws/commerce.htm
32. постановление президента республики узбекистан "о дополнительных мерах по дальнейшему развитию информационно- коммуникационных технологий" от 08.07.2005г. - http://ict.gov.uz/rus/normativno_pravovaya_baza/ukazi_i_postanovleniya_prezidenta/pp_117.mgr
33. Закон республики узбекистан от 16.04.2006 года № зру – 13, об электронных платежах. - http://cbu.uz/ru/banking_legislation/ruz_laws/electron_outgoing_law.htm
34. Постановление кабинета министров республики узбекистан от 01.02.2012 г. n 24 - о мерах по созданию условий для дальнейшего развития компьютеризации и иктна местах - http://norma.uz/publish/doc/text79389_o_merah_po_sozdaniyu_usloviy_dlya_dalneyshego_razvitiya_kompyuterizacii_i_informacionno-kommuni-_kacionnyh_tehnologiy_na_mestah
35. Постановление президента республики узбекистан от 21.03.2012 г. n пп-1730 - о мерах по дальнейшему внедрению и развитию современных информационно-коммуникационных технологий- http://norma.uz/publish/doc/text80400_o_merah_po_dalneyshemu_vnedreniyu_i_razvitiyu_sovremennyh_informacionno-_kommunikacionnyh_tehnologiy
36. http://ulugov.uz
37. http://gov.uz/uz/
38. http://ziyonet.uz/

Приложение

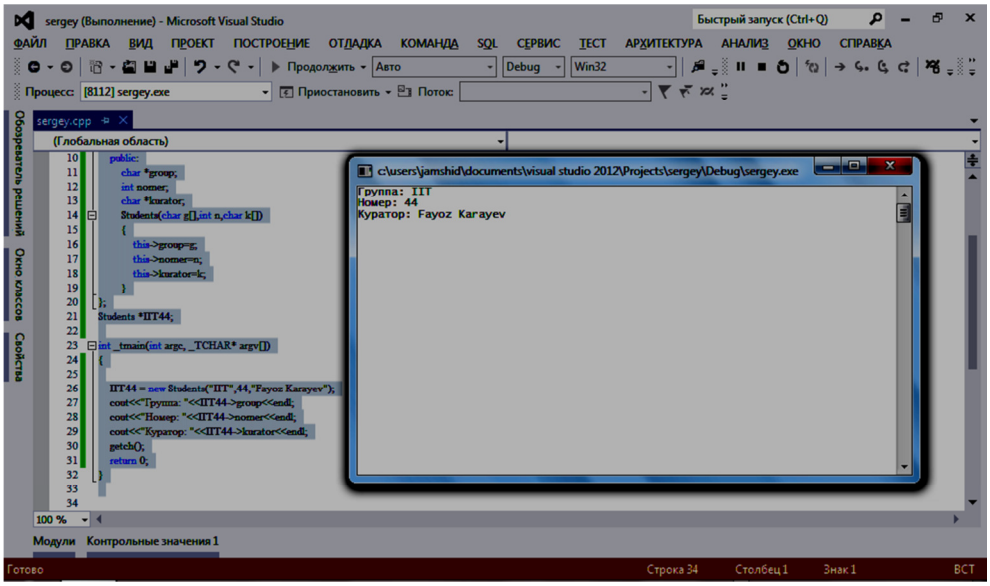
1 Реализация класса

```
// sergey.cpp: определяет точку входа для консольного приложения.
//

#include "stdafx.h"
#include <iostream>
#include  <conio.h>
using namespace std;
class Students
{
	public:
		char *group;
		int nomer;
		char *kurator;
		Students(char g[],int n,char k[])
		{
			this->group=g;
			this->nomer=n;
			this->kurator=k;
		}
};
Students *IIT44;

int _tmain(int argc, _TCHAR* argv[])
{

	IIT44 = new Students("IIT",44,"Fayoz Karayev");
	cout<<"Группа: "<<IIT44->group<<endl;
	cout<<"Номер: "<<IIT44->nomer<<endl;
	cout<<"Куратор: "<<IIT44->kurator<<endl;
	getch();
	return 0;
}
```



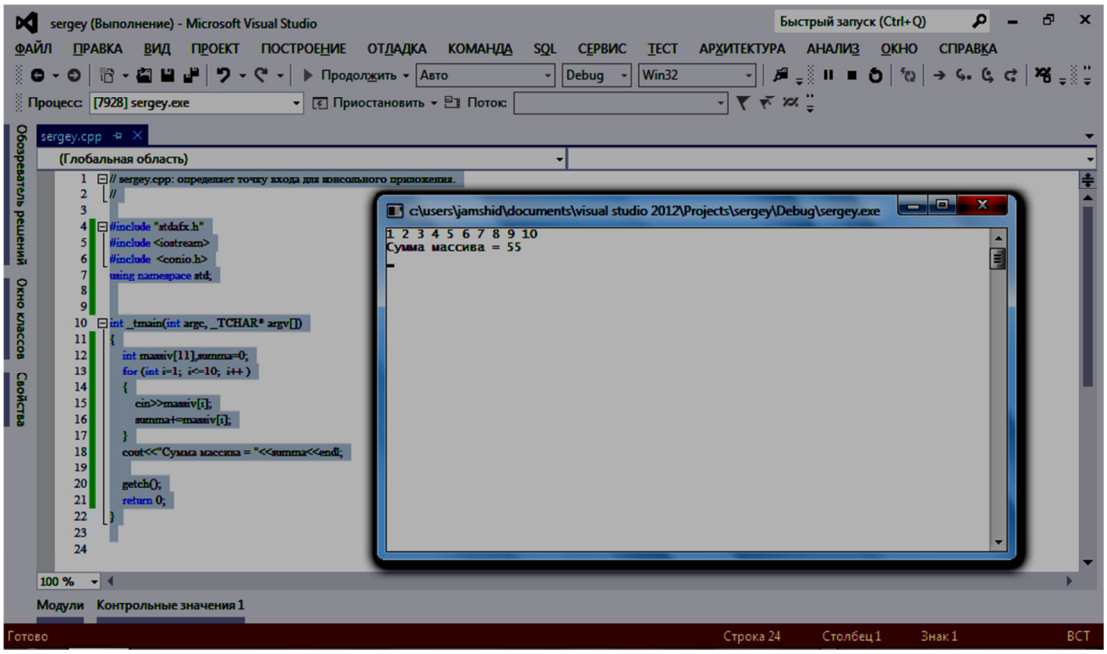
2 Реализация массива

```
// sergey.cpp: определяет точку входа для консольного приложения.
//

#include "stdafx.h"
#include <iostream>
#include  <conio.h>
using namespace std;


int _tmain(int argc, _TCHAR* argv[])
{
	int massiv[11],summa=0;
	for (int i=1;  i<=10;  i++ )
	{
		cin>>massiv[i];
		summa+=massiv[i];
	}
	cout<<"Сумма массива = "<<summa<<endl;

	getch();
	return 0;
}
```



3 Реализация объединения

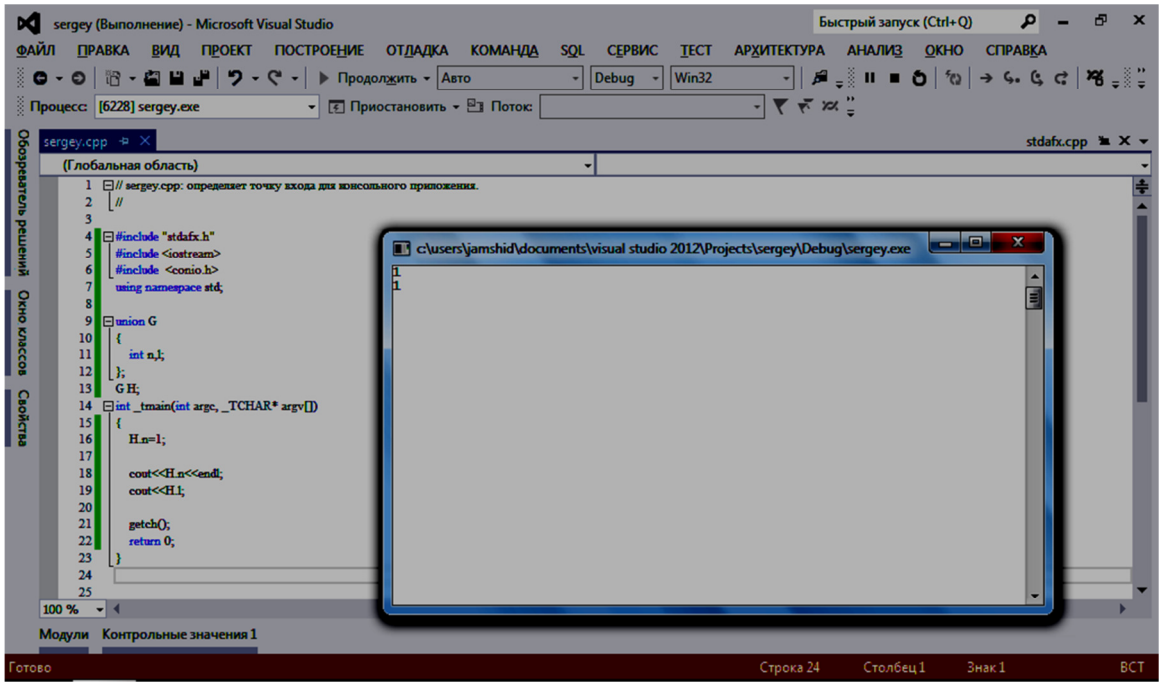
```
// sergey.cpp: определяет точку входа для консольного приложения.
//

#include "stdafx.h"
#include <iostream>
#include  <conio.h>
using namespace std;

union G
{
	int n,l;
};
G H;
int _tmain(int argc, _TCHAR* argv[])
{
	H.n=1;

	cout<<H.n<<endl;
	cout<<H.l;

	getch();
	return 0;
}
```



4 Реализация структуры

```
// sergey.cpp: определяет точку входа для консольного приложения.
//

#include "stdafx.h"
#include <iostream>
#include <string>
#include  <conio.h>
using namespace std;

struct Sergey
{
	string nationality;
	bool chistokrovniy;
};

Sergey Koreets;

int _tmain(int argc, _TCHAR* argv[])
{

	Koreets.nationality = "Кореец";
	Koreets.chistokrovniy=1;

	cout<<"Национальность: "<<Koreets.nationality<<endl;
	cout<<"Чистокровный: "<<((Koreets.chistokrovniy)?"Да":"Нет")<<endl;

	getch();
	return 0;
}
```