Министерство образования Республики Башкортостан
ГАОУ СПО Стерлитамакский колледж строительства, экономики и права

Учебно – методический комплекс

по разделу 1 «Разработка программного модуля» МДК 01.01 «Системное программирование» профессионального модуля «Разработка программных модулей программного обеспечения для компьютерных систем»

для групп специальности 09.02.03 «Программирование в компьютерных системах»

Разработала преподаватель: ***Хасанова А. Х.***

2020

СОДЕРЖАНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Учебно-методический комплекс по разделу 1 «Разработка программного модуля» МДК 01.01 «Системное программирование» профессионального модуля «Разработка программных модулей программного обеспечения для компьютерных систем» представляет собой систему нормативной и учебно-методической документации, средств обучения и контроля, необходимых и достаточных для качественной организации освоения образовательной программы профессионального модуля, согласно учебного плана.

Целью создания УМК является разработка методического обеспечения раздела МДК, а также предоставление студенту полного комплекта учебно-методических материалов для самостоятельного изучения раздела.

При разработке УМК раздела учитывалось, что его компоненты должны:

- соответствовать общей идеологии федеральной и региональной политики, содействовать развитию региональной системы среднего профессионального образования;
- предусматривать логически последовательное изложение учебного материала;
- предполагать использование современных методов и технических средств интенсификации учебного процесса, позволяющих студентам глубоко осваивать учебный материал и получать навыки по его использованию на практике;
- соответствовать современным научным представлениям в предметной области;
- обеспечивать межпредметные связи;
- обеспечивать простоту использования для преподавателей и студентов.

Представленный учебно-методический комплекс был разработан в соответствии с ФГОС и апробирован в учебном процессе. Так же был проведен анализ результатов текущего контроля студентов, на основе которого внесены коррективы в содержание учебного материала. В настоящее время разработка УМК завершена. Он содержит:

1.рабочую программу раздела междисциплинарного курса профессионального модуля;

2.календарно – тематический план;

3. курс лекций;

4.задания и методические указания по выполнению практических работ;

5.задания и методические указания по выполнению лабораторных работ;

6.задания для текущего и итогового контроля знаний студентов;

7.методические рекомендации по выполнению самостоятельной работы;

8.методические рекомендации по организации и методическому сопровождению самостоятельной работы студентов;

9.комплект контрольно – измерительных материалов для проведения дифференцированного зачета

Учебно-методический комплекс по разделу 1 «Разработка программного модуля» МДК 01.01 «Системное программирование» профессионального модуля «Разработка программных модулей программного обеспечения для компьютерных систем» может быть использован в профессиональной подготовке студентов специальностей СПО 230000 Информатика и вычислительная техника и в дополнительном профессиональном образовании повышения квалификации и переподготовки кадров в области разработки программного обеспечения.

РАЗДЕЛ 1. ПАСПОРТ РАБОЧЕЙ ПРОГРАММЫ

Раздела 1 «Разработка программного модуля» МДК 01.01 «Системное программирование» профессионального модуля «Разработка программных модулей программного обеспечения для компьютерных систем»

Область применения программы

Рабочая программа раздела МДК является частью рабочей основной профессиональной образовательной программы в соответствии с ФГОС по специальности СПО **230115 Программирование в компьютерных системах** в части освоения основного вида профессиональной деятельности (ВПД):

Разработка программных модулей программного обеспечения для компьютерных систем и соответствующих профессиональных компетенций (ПК):

ПК 1.1. Выполнять разработку спецификаций отдельных компонент.

ПК 1.6. Разрабатывать компоненты проектной и технической документации с использованием графических языков спецификаций.

Рабочая программа раздела МДК может быть использована в профессиональной подготовке специальностей СПО **230000 Информатика и вычислительная техника** и в дополнительном профессиональном образовании повышения квалификации и переподготовки кадров в области разработки программного обеспечения.

1.2. Цели и задачи модуля – требования к результатам освоения модуля

С целью овладения указанным видом профессиональной деятельности и соответствующими профессиональными компетенциями обучающийся в ходе освоения профессионального модуля должен:

уметь:

- оформлять документацию на программные средства;
- использовать инструментальные средства для автоматизации оформления документации;

знать:

- основные этапы разработки программного обеспечения;
- основные принципы технологии структурного и объектно-ориентированного программирования;
- основные принципы отладки и тестирования программных продуктов;
- методы и средства разработки технической документации

1.3. Рекомендуемое количество часов на освоение программы раздела МДК:

всего – **105** часов, в том числе:

максимальной учебной нагрузки обучающегося – **105** часов, включая:

обязательную аудиторную учебную нагрузку обучающегося – **70** часов;

самостоятельную работу обучающегося – **35** часов.

Результаты освоения раздела 1 «Разработка программного модуля»
МДК 01.01 «Системное программирование»
профессионального модуля «Разработка программных модулей программного обеспечения для компьютерных систем»

Результатом освоения программы раздела МДК является овладение профессиональными (ПК) и общими (ОК) компетенциями:	
Код	**Наименование результата обучения**
ПК 1.1.	Выполнять разработку спецификаций отдельных компонент.
ПК 1.6.	Разрабатывать компоненты проектной и технической документации с использованием графических языков спецификаций.
ОК 1.	Понимать сущность и социальную значимость своей будущей профессии, проявлять к ней устойчивый интерес.
ОК 2.	Организовывать собственную деятельность, выбирать типовые методы и способы выполнения профессиональных задач, оценивать их эффективность и качество.
ОК 3.	Принимать решения в стандартных и нестандартных ситуациях и нести за них ответственность.
ОК 4.	Осуществлять поиск и использование информации, необходимой для эффективного выполнения профессиональных задач, профессионального и личностного развития.
ОК 5.	Использовать информационно-коммуникационные технологии в профессиональной деятельности.
ОК 6.	Работать в коллективе и в команде, эффективно общаться с коллегами, руководством, потребителями.
ОК 7.	Брать на себя ответственность за работу членов команды (подчиненных), за результат выполнения заданий.
ОК 8.	Самостоятельно определять задачи профессионального и личностного развития, заниматься самообразованием, осознанно планировать повышение квалификации.
ОК 9.	Ориентироваться в условиях частой смены технологий в профессиональной деятельности.
ОК 10.	Исполнять воинскую обязанность, в том числе с применением полученных профессиональных знаний (для юношей).

Содержание обучения по профессиональному модулю (ПМ)

Наименование разделов и тем	Содержание учебного материала, лабораторные работы и практические занятия, самостоятельная работа обучающихся, курсовая работа		Объем часов	Уровень освоения
МДК 01.01 Системное программирование				
Раздел 1. Разработка программного модуля			**70**	
Тема 1.1. Программные продукты и их основные характеристики	**Содержание**		6	
	1	Основные понятия программного обеспечения		1
	2	Понятие жизненного цикла программы		2
	3	Основные этапы разработки программного обеспечения		2
Тема 1.2. Стадии разработки программ и программной документации	**Содержание**		4	
	1	Методы и средства разработки технической документации. Понятие о ЕСПД (ГОСТ 19.XXX-XX, ГОСТ 34.XXX-XX).		1
	2	Графические языки спецификаций		2
Тема 1.3. Методы проектирования программных продуктов	**Содержание**		4	
	1	Принципы системного проектирования.		2
	2	Объектно-ориентированное проектирование программных продуктов.		2
Тема 1.4. Проектирование интерфейса пользователя	**Содержание**		4	
	1	Интерфейс пользователя программного продукта ОК 6.1, 6.2, 6.3		1
	2	Основные принципы разработки пользовательского интерфейса		2
Тема 1.5. Методы разработки программных модулей	**Содержание**		4	
	1	Сущность модульного программирования		2
	2	Основные принципы технологии структурного программирования ОК6.1		2
	Практические занятия		4	
	1	Проведение анализа программ с точки зрения стиля. Составление удобочитаемой программы. ОК 2.1, 2.2, 2.3.		
	2	Создание структурного алгоритма ОК 2.1, 2.2		
Тема 1.6. Объектно - ориентированное программирование	**Содержание**		4	
	1	Основные принципы технологии объектно-ориентированного программирования.		2
	2	Этапы объектно- ориентированного проектирования. ОК 6.1, 6.2		2
Тема 1.7. Эффективность и оптимизация программ	**Содержание**		4	
	1	Основные критерии эффективности программного продукта		2
	2	Принципы и приемы оптимизации. ОК 6.1, 6.2, 6.3		2
	Лабораторные занятия		2	
	1	Применение оптимизирующих компиляторов ОК 2.1, 2.2		

Тема 1.8. Отладка программ	**Содержание**		4	
	1	Понятие об ошибке программного обеспечения.		2
	2	Основные принципы отладки программных продуктов.		2
	Лабораторные занятия		2	
	1	Использование средств отладки программ ОК 2.1, 2.2, 2.3		
Тема 1.9. Методы тестирования программ	**Содержание**		4	
	1	Основные принципы тестирования программных продуктов		2
	2	Методы структурного и функционального тестирования программного обеспечения ОК 6.1, 6.2.		2
	Лабораторные занятия		4	
	1	Тестирование программ методами «белого ящика»: покрытия операторов, покрытия решений, покрытия условий		
	2	Тестирование программ методами «белого ящика»: покрытия решений/условий, комбинаторного покрытия условий		
Тема 1.10. Сопровождение программ	**Содержание**		2	
	1	Виды программных документов. Методы разработки программной документации. Технологии разработки документации		2
	Лабораторные занятия		16	
	1	Оформление программной документации. Стадии «Техническое задание» ОК 2.1, 2.2, 2.3.		
	2	Оформление программной документации. Стадии «Эскизный проект» ОК 2.1, 2.2, 2.3.		
	3	Оформление программной документации. Стадии «Технический проект» ОК 2.1, 2.2, 2.3.		
	4	Оформление программной документации. Стадии «Реализация» ОК 2.1, 2.2, 2.3.		
	Практические занятия		2	
	1	Дифференцированный зачет		
Самостоятельная работа при изучении раздела 1 ПМ 01. 1. Реферирование темы «Вспомогательные процессы жизненного цикла программного продукта» 2. Аналитический обзор литературы по теме «Каскадная модель жизненного цикла разработки программного продукта» 3. Подготовка доклада по теме «Модель прототипирования жизненного цикла разработки программного продукта» 4. Составление сводной таблицы «Основные ГОСТы, использующиеся для описания жизненного цикла программного продукта» 5. Составление сводной таблицы «Типы технических решений и документация на них». 6. Аналитический обзор литературы по теме «Методы проектирования программных продуктов» 7. Составление кроссворда по теме «Методы проектирования программных продуктов») 8. Реферирование темы «Этапы проектирования пользовательского интерфейса» 9. Составление диаграммы Насси-Шнейдермана решения задачи 10. Разработка мультимедийной презентации по теме «Принципы объектно-ориентированного проектирования» 11. Аналитический обзор литературы по теме «Количественные характеристики надежности программ» 12. Выполнение упражнения на применение простых приемов оптимизации программ в среде Turbo Pascal 13. Подготовка доклада «Возможности встроенного отладчика интегрированной среды Turbo Pascal» 14. Составление опорного конспекта по теме «Методы тестирования программ» 15. Составление опорного конспекта по теме «Технологии разработки документации» 16. Составление словаря терминов по разделу «Разработка программного модуля»			**35**	

Условия реализации раздела МДК

Требования к минимальному материально-техническому обеспечению

Реализация программы раздела МДК предполагает: наличие лаборатории: прикладного программирования;

Оборудование учебного кабинета и рабочих мест кабинета:

- рабочие места с персональным компьютером по количеству обучающихся;
- рабочее место преподавателя;
- комплект плакатов по МДК;
- комплект учебно-методической документации;
- макеты и наглядные пособия по МДК

Технические средства обучения: - лицензионное программное обеспечение, выход в глобальную сеть Internet на каждом ПК, точки электропитания, сетевое оборудование, обеспечивающее работу локальной сети, мультимедийное оборудование, источники бесперебойного питания, интерактивная доска.

Информационное обеспечение обучения

Перечень рекомендуемых учебных изданий, Интернет-ресурсов, дополнительной литературы

Основные источники:

1.Рудаков А. В. Технология разработки программных продуктов.- М.: Издательский центр «Академия», 2006. - 208 с

2.Культин, Н.Б. Turbo Pascal в задачах и примерах: учебное пособие.–БХВ., 2007 – 256 с.

3.Павловская, Т.А. Паскаль. Программирование на языке высокого уровня. - СПб: Питер, 2007 – 393 с.

Интернет-ресурсы

1. Технология разработки программных продуктов: http://chemisk.narod.ru/html/trpp01.html

2. Введение в технологию разработки программных продуктов: http://www.intuit.ru/department/se/introprogteach/1/

Кадровое обеспечение образовательного процесса

Требования к квалификации педагогических кадров, обеспечивающих обучение по междисциплинарному курсу, руководство практикой: наличие высшего образования, соответствующего профилю модуля.

Инженерно–педагогический состав: преподаватели с высшим образованием, соответствующим профилю модуля, имеющие опыт деятельности в профильных организациях, обязательное прохождение стажировки в профильных организациях не реже 1 раза в 3 года.

Контроль и оценка результатов освоения профессионального модуля (вида профессиональной деятельности)

Результаты (освоенные профессиональные компетенции)	Основные показатели результатов подготовки	Формы и методы контроля и оценки результатов обучения
ПК 1.1. Разработка спецификаций отдельных компонент.	Разработка спецификаций отдельных компонент выполнена в соответствии с требованиями: 1) составлена на основе ЕСПД (ГОСТ 19.XXX-XX, ГОСТ 34.XXX-XX); 2) отражает заданную функциональность и качество компонента	Оценка продукта учебной деятельности (спецификация) по критериям на квалификационном экзамене
ПК 1.6. Разрабатывать компоненты проектной и технической документации с использованием графических языков спецификаций	Разработка компонента проектной и технической документации выполнена в соответствии с требованиями: 1) использованы графические язык спецификаций; 2) составлена на основе ЕСПД (ГОСТ 19.XXX-XX, ГОСТ 34.XXX-XX); 3) отражает заданную функциональность и качество	Оценка продукта учебной деятельности (компоненты проектной и технической документации) по критериям на производственной практике

Формы и методы контроля и оценки результатов обучения должны позволять проверять у обучающихся не только сформированность профессиональных компетенций, но и развитие общих компетенций и обеспечивающих их умений.

Результаты обучения (освоенные общие компетенции)		**Основные показатели оценки результата**	**Формы и методы контроля и оценки результатов обучения**
Код	**Формулировка**		
ОК 1	Понимать сущность и социальную значимость своей профессии, проявлять к ней устойчивый интерес	Приведены произвольные примеры социальной значимости своей профессии Сформулированы сущностные характеристики профессии «программист-техник»	Оценка результатов стандартизованного тестирования на итоговом испытании (экзамене/диф.зачете по УД или МДК, сертификационном экзамене по модулю)
ОК 2	Организовывать собственную деятельность, выбирать типовые методы и способы выполнения профессиональных задач, оценивать их эффективность и качество.	Поставленная цель деятельности соответствует условиям профессиональной задачи Набор ресурсов определен в соответствии с поставленной целью Разметка времени выполнения профессиональной задачи проведена в соответствии с поставленной целью и имеющимися ресурсами Выбранный метод и способ решения профессиональной задачи соответствует типовому (известному) алгоритму решения и проведенной разметки времени Оценка эффективности и качества метода и способа решения задачи соответствует заданной методике оценивания	Оценка продукта деятельности обучающегося (решение практико-ориентированного задания) по критериям (соответствие условиям задачи, типовому алгоритму решения, методики оценивания) на итоговом испытании (экзамене/диф.зачете по УД или МДК, сертификационном экзамене по модулю)
ОК 6	Работать в коллективе и в команде, эффективно общаться с коллегами, руководством, потребителями.	Обсуждение технического задания с потребителем проведено в соответствии с пунктами «Опросного листа клиента»	Оценка результатов формализованного наблюдения за учебной/ профессиональной деятельностью обучающегося на учебной/ производственной практике
		План выполнения работ составлен в соответствии с техническим заданием и одобрен руководством.	Оценка продукта деятельности обучающегося (план работ) по критериям (соответствие тех. заданию) на учебной/ производственной практике
		Распределение работ среди членов команды выполнено в соответствии с планом выполнения работ. Техническое задание выполнено согласно выработанному плану работ	Оценка продукта деятельности обучающегося (техническое задание) по критериям (соответствие пунктам плана) на учебной/производственной практике

РАЗДЕЛ 2. КАЛЕНДАРНО-ТЕМАТИЧЕСКИЙ ПЛАН

Раздела 1 «Разработка программного модуля» МДК 01.01 «Системное программирование» профессионального модуля «Разработка программных модулей программного обеспечения для компьютерных систем»

Наименование разделов, тем	Количество часов	Дата проведения	№ занятия	Вид занятия	Оборудование занятия	Самостоятельная работа студентов	Домашнее задание
Тема 1.1. Программные продукты и их основные характеристики	**6**						
1.1.1. Ознакомление с формами промежуточного и итогового контроля. Основные понятия программного обеспечения	2		1	Комбинированный урок	Мультимедийный проектор	Составление опорного конспекта по теме	Реферирование темы «Вспомогательные процессы жизненного цикла программного продукта»
1.1.2. Понятие жизненного цикла программы	2		2	Комбинированный урок	Раздаточный материал	Составление опорного конспекта по теме	Аналитический обзор литературы по теме «Каскадная модель жизненного цикла разработки программного продукта»
1.1.3. Основные этапы разработки программного обеспечения	2		3	Комбинированный урок	Рабочие места с персональным компьютером	Работа по контрольным вопросам с электронным учебником	Подготовка доклада по теме «Модель прототипирования жизненного цикла разработки программного продукта»
Тема 1.2. Архитектурные особенности операционных систем	**4**						
1.2.1 Методы и средства разработки технической документации. Понятие о ЕСПД (ГОСТ 19.XXX-XX, ГОСТ 34.XXX-XX).	2		4	Комбинированный урок	Мультимедийный проектор	Составление опорного конспекта по теме	Составление сводной таблицы «Основные ГОСТы, использующиеся для описания жизненного цикла ПП»
1.2.2. Графические языки спецификаций	2		5	Комбинированный урок	Раздаточный материал	Работа по контрольным вопросам с электронным учебником	Составление сводной таблицы «Типы технических решений и документация на них»
Тема 1.3. Методы проектирования программных продуктов	**4**						

Наименование разделов, тем	Количество часов	Дата проведения	№ занятия	Вид занятия	Оборудование занятия	Самостоятельная работа студентов	Домашнее задание
1.3.1. Принципы системного проектирования	2		6	Комбинированный урок	Мультимедийный проектор	Составление опорного конспекта по теме	Аналитический обзор литературы по теме «Методы проектирования программных продуктов»
1.3.2. Объектно-ориентированное проектирование программных продуктов	2		7	Комбинированный урок	Раздаточный материал	Работа по контрольным вопросам с электронным учебником	
Тема 1.4. Проектирование интерфейса пользователя	**4**						
1.4.1. Интерфейс пользователя программного продукта	2		8	Комбинированный урок	Раздаточный материал	Составление опорного конспекта по теме	Составление кроссворда по теме «Методы проектирования программных продуктов»
1.4.2. Основные принципы разработки пользовательского интерфейса	2		9	Комбинированный урок	Рабочие места с персональным компьютером	Работа по контрольным вопросам с электронным учебником	Реферирование темы «Этапы проектирования пользовательского интерфейса»
Тема 1.5. Методы разработки программных модулей	**8**						
1.5.1.Сущность модульного программирования	2		10	Комбинированный урок	Раздаточный материал	Составление опорного конспекта по теме	Составление диаграммы Насси-Шнейдермана решения задачи
1.5.2. Основные принципы технологии структурного программирования	2		11	Комбинированный урок	Рабочие места с персональным компьютером	Работа по контрольным вопросам с электронным учебником	
Проведение анализа программ с точки зрения стиля. Составление удобочитаемой программы	2		12	Практическая работа	Раздаточный материал	Выполнение практической работы	
Создание структурного алгоритма	2		13	Практическая работа	Раздаточный материал	Выполнение практической работы	
Тема 1.6. Объектно - ориентированное программирование	**4**						
1.6.1. Основные принципы технологии объектно-ориентированного программирования.	2		14	Комбинированный урок	Мультимедийный проектор	Составление опорного конспекта по теме	Разработка мультимедийной презентации по теме «Принципы объектно-ориентированного проектирования»
1.6.2. Этапы объектно- ориентированного проектирования.	2		15	Комбинированный урок	Мультимедийный проектор	Работа по контрольным вопросам с электронным учебником	

Наименование разделов, тем	Количество часов	Дата проведения	№ занятия	Вид занятия	Оборудование занятия	Самостоятельная работа студентов	Домашнее задание
Тема 1.7. Эффективность и оптимизация программ	**6**						
1.7.1. Основные критерии эффективности программного продукта	2		16	Комбинированный урок	Раздаточный материал	Составление опорного конспекта по теме	Аналитический обзор литературы по теме «Количественные характеристики надежности программ»
1.7.2. Принципы и приемы оптимизации.	2		17	Комбинированный урок	Рабочие места с персональным компьютером	Работа по контрольным вопросам с электронным учебником	Выполнение упражнения на применение простых приемов оптимизации программ в среде Turbo Pascal
Применение оптимизирующих компиляторов	2		18	Лабораторная работа	Рабочие места с персональным компьютером по количеству обучающихся	Выполнение лабораторной работы	
Тема 1.8. Отладка программ	**6**						
1.8.1. Понятие об ошибке программного обеспечения	2		19	Комбинированный урок	Мультимедийный проектор	Составление опорного конспекта по теме	Подготовка доклада «Возможности встроенного отладчика интегрированной среды Turbo Pascal»
1.8.2. Основные принципы отладки программных продуктов	2		20	Комбинированный урок	Раздаточный материал	Работа по контрольным вопросам с электронным учебником	
Использование средств отладки программ	2		21	Лабораторная работа	Рабочие места с персональным компьютером по количеству обучающихся	Выполнение лабораторной работы	
Тема 1.9. Методы тестирования программ	**8**						
1.9.1. Основные принципы тестирования программных продуктов	2		22	Комбинированный урок	Рабочие места с персональным компьютером	Работа по контрольным вопросам с электронным учебником	Составление опорного конспекта по теме «Методы тестирования программ»
1.9.2. Методы структурного и функционального тестирования программного	2		23	Комбинированный урок	Рабочие места с персональным	Работа по контрольным вопросам с электронным	

Наименование разделов, тем	Количество часов	Дата проведения	№ занятия	Вид занятия	Оборудование занятия	Самостоятельная работа студентов	Домашнее задание
обеспечения					компьютером	учебником	
Тестирование программ методами «белого ящика»: покрытия операторов, покрытия решений, покрытия условий	2		24	Лабораторная работа	Рабочие места с персональным компьютером по количеству обучающихся	Выполнение лабораторной работы	
Тестирование программ методами «белого ящика»: покрытия решений/условий, комбинаторного покрытия условий	2		25	Лабораторная работа	Рабочие места с персональным компьютером по количеству обучающихся	Выполнение лабораторной работы	
Тема 1.10. Сопровождение программ	**18**						
1.10.1. Виды программных документов. Методы разработки программной документации. Технологии разработки документации	2		26	Комбинированный урок	Рабочие места с персональным компьютером	Работа по контрольным вопросам с электронным учебником	Составление опорного конспекта по теме «Технологии разработки документации»
Оформление программной документации. Стадии «Техническое задание»	4		27, 28	Лабораторная работа	Раздаточный материал	Выполнение лабораторной работы	Составление словаря терминов по разделу «Разработка программного модуля»
Оформление программной документации. Стадии «Эскизный проект»	4		29, 30	Лабораторная работа	Раздаточный материал	Выполнение лабораторной работы	
Оформление программной документации. Стадии «Технический проект»	4		31, 32	Лабораторная работа	Раздаточный материал	Выполнение лабораторной работы	
Оформление программной документации. Стадии «Реализация»	4		33, 34	Лабораторная работа	Раздаточный материал	Выполнение лабораторной работы	
Дифференцированный зачет	**2**		35	Практическая работа	Раздаточный материал	Выполнение практической работы	

РАЗДЕЛ 3. КУРС ЛЕКЦИЙ
по разделу 1 «Разработка программного модуля» МДК 01.01 «Системное программирование» профессионального модуля «Разработка программных модулей программного обеспечения для компьютерных систем»

ВВЕДЕНИЕ

Технологии программирования играли разную роль на разных этапах развития программирования. По мере повышения мощности компьютеров и развития средств и методологии программирования росла и сложность решаемых на компьютерах задач, что привело к повышенному вниманию к технологии программирования. Резкое удешевление стоимости компьютеров и, в особенности, стоимости хранения информации на компьютерных носителях привело к широкому внедрению компьютеров практически во все сферы человеческой деятельности, что существенно изменило направленность технологии программирования. Человеческий фактор стал играть в ней решающую роль. Сформировалось достаточно глубокое понятие качества ПС, в котором акценты стали ставится не столько на его эффективности, сколько на удобстве работы с ним для пользователей (не говоря уже о его надежности). Широкое использование компьютерных сетей привело к интенсивному развитию распределенных вычислений, дистанционного доступа к информации и электронного способа обмена сообщениями между людьми. Компьютерная техника из средства решения отдельных задач все более превращается в средство информационного моделирования реального и мыслимого мира, способное просто отвечать людям на интересующие их вопросы. Начинается этап глубокой и полной информатизации (компьютеризации) человеческого общества. Все это ставит перед технологией программирования новые и достаточно трудные проблемы.

Сделаем краткую характеристику развития программирования по десятилетиям.

В 50-е годы мощность компьютеров была невелика (компьютеры первого поколения), а программирование для них велось, в основном, в машинном коде. Решались в главным образом научно-технические задачи (счет по формулам), задание на программирование уже содержало, как правило, достаточно точную постановку задачи. Использовалась интуитивная технология программирования: почти сразу приступали к составлению программы по заданию, при этом часто задание несколько раз изменялось (что сильно увеличивало время и без того итерационного процесса составления программы), минимальная документация оформлялась уже после того, как программа начинала работать. Тем не менее именно в этот период родилась фундаментальная для технологии программирования концепция модульного программирования (для преодоления трудностей программирования в машинном коде). Появились первые языки программирования высокого уровня, из которых только ФОРТРАН пробился для использования в следующие десятилетия.

В 60-е годы можно было наблюдать бурное развитие и широкое использование языков программирования высокого уровня (АЛГОЛ 60, ФОРТРАН, КОБОЛ и др.), роль которых в технологии программирования явно преувеличивалась. Надежда на то, что эти языки решат все проблемы при разработки больших программ, не оправдалась. В результате повышения мощности компьютеров и накопления опыта программирования на языках высокого уровня быстро росла сложность решаемых на компьютерах задач, в результате чего обнаружилась ограниченность языков, проигнорировавших модульную организацию программ. И только ФОРТРАН, бережно сохранивший возможность модульного программирования, гордо прошествовал в следующие десятилетия (все его ругали, но его пользователи отказаться от его услуг не могли из-за грандиозного накопления фонда программных модулей, которые с успехом использовались в новых программах). Кроме того, было понято, что важно не только то, на каком языке мы программируем, но и то, как мы программируем. Это было уже началом серьезных размышлений над методологией и тех-

нологией программирования. Появление в компьютерах 2-го поколения прерываний привело к развитию мультипрограммирования и созданию больших программных систем. Последнее стало возможным с использования коллективной разработки, которая поставила ряд серьезных технологических проблем.

В 70-е годы получили широкое распространение информационные системы и базы данных. Этому способствовало очень важное событие, происшедшее в середине 70-ых годов: стоимость хранения одного бита информации на компьютерных носителях стала меньше, чем на традиционных. Интенсивно развивалась технология программирования: обоснование и широкое внедрение нисходящей разработки и структурного программирования, развитие абстрактных типов данных и модульного программирования (в частности, возникновение идеи разделения спецификации и реализации модулей и использование модулей, скрывающих структуры данных), исследование проблем обеспечения надежности и мобильности ПС, создание методики управления коллективной разработкой ПС, появление инструментальных программных средств (программных инструментов) поддержки технологии программирования.

80-е годы характеризуются широким внедрением персональных компьютеров во все сферы человеческой деятельности и тем самым созданием обширного и разнообразного контингента пользователей ПС. Это привело к бурному развитию пользовательских интерфейсов и созданию четкой концепции качества ПС. Появляются языки программирования (например, Ада), учитывающие требования технологии программирования. Развиваются методы и языки спецификации ПС. Выходит на передовые позиции объектный подход к разработке ПС. Создаются различные инструментальные среды разработки и сопровождения ПС. Развивается концепция компьютерных сетей.

90-е годы знаменательны широким охватом всего человеческого общества международной компьютерной сетью, персональные компьютеры стали подключаться к ней как терминалы. Это поставило ряд проблем регулирования доступа к компьютерно-сетевой информации (как технологического, так и юридического и этического характера). Остро встала проблема защиты компьютерной информации и передаваемых по сети сообщений. Стали бурно развиваться компьютерная технология (CASE-технология) разработки ПС и связанные с ней формальные методы спецификации программ. Начался решающий этап полной информатизации и компьютеризации общества.

Тема 1.1. Программные продукты и их основные характеристики

1.1.1. Основные понятия программного обеспечения

Возможности компьютера как технической основы системы обработки данных связаны с используемым программным обеспечением.

Программа (program, routine) - упорядоченная последовательность команд (инструкций) компьютера для решения задачи.

Программное обеспечение (sowtware) - совокупность программ обработки данных и необходимых для их эксплуатации документов.

Программы предназначены для машинной реализации задач. Термины задачи и приложение имеют очень широкое употребление в контексте информатики и программного обеспечения.

Задача (problem, task) - проблема, подлежащая решению. Приложение (application) - программная реализация на компьютере решения задачи.

Таким образом, задача означает проблему, подлежащую реализации с использованием средств информационных технологий, а приложение - реализованное на компьютере решение по задаче. Приложение, являясь синонимом слова "программа", считается более удачным термином и широко используется в информатике.

Термин задача употребляется также в сфере программирования, особенно в режиме мультипрограммирования и мультипроцессорной обработки, как единица работы вычислительной системы, требующая выделения вычислительных ресурсов (процессорного

времени, основной памяти и т.п.). В данной главе этот термин употребляется в смысле первого определения.

Существует большое число разнообразных классификаций задач. С позиций специфики разработки и вида программного обеспечения будем различать два класса задач - технологические и функциональные.

Технологические задачи ставятся и решаются при организации технологического процесса обработки информации на компьютере. Технологические задачи являются основой для разработки сервисных средств программного обеспечения в виде утилит, сервисных программ, библиотек процедур и др., применяемых для обеспечения работоспособности компьютера, разработки других программ или обработки данных функциональных задач.

Функциональные задачи требуют решения при реализации функций управления в рамках информационных систем предметных областей. Например, управление деятельностью торгового предприятия, планирование выпуска продукции, управление перевозкой грузов и т.п. Функциональные задачи в совокупности образуют предметную область и полностью определяют ее специфику.

Предметная (прикладная) область (application domain) - совокупность связанных между собой функций, задач управления, с помощью которых достигается выполнение поставленных целей.

Процесс создания программ можно представить как последовательность действий, представленных на рис. 1.

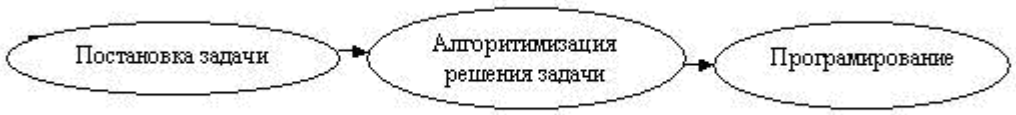


Рис. 1. Схема процесса создания программ

Постановка задачи (problem definition) - это точная формулировка решения задачи на компьютере с описанием входной и выходной информации.

К основным характеристикам функциональных задач, уточняемым в процессе ее формализованной постановки, относятся:

-цель или назначение задачи, ее место и связи с другими задачами;

-условия решения задачи с использованием средств вычислительной техники;

-содержание функций обработки входной информации при решении задачи;

-требования к периодичности решения задачи;

-ограничения по срокам и точности выходной информации;

-состав и форма представления выходной информации;

-источники входной информации для решения задачи;

-пользователи задачи (кто осуществляет ее решение и пользуется результатами решение и пользуется результатами решения).

Выходная информация по задаче может быть представлена в виде документ типа листинга или машинограммы), сформированных кадров - видеограммы на экране монитора файла базы данных, выходного сигнала устройству управления (рис. 2).

Входная информация по задаче определяется как данные, поступающие на код задачи и используемые для ее решения. Входной информацией служат первичные данные документов ручного заполнения, информация, хранимая в файлах базы данных (результаты решения других задач, нормативно-справочная информация - классификаторы, кодификаторы, справочники), входные сигналы отдатчиков (см. рис. 2).

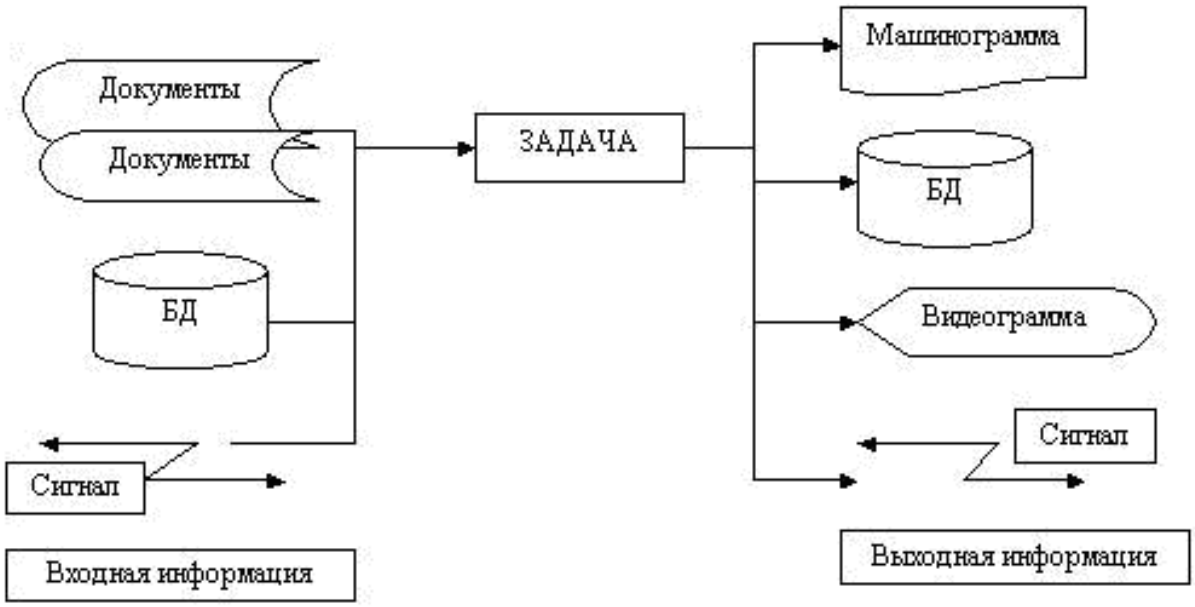


Рис. 2. Схема входной и выходной информации

Программирование (programming) - теоретическая и практическая деятельность, связанная с созданием программ. Программирование базируется на комплексе научных дисциплин, направленных на исследование, разработку и применение методов и средств разработки программ (специализированного инструментария создания программ). При разработке программ используются ресурсоемкие и наукоемкие технологии, высококвалифицированный интеллектуальный труд.

Технология программирования – это совершенствование профессиональной культуры программирования, организация и упорядочение труда самого программиста, независимо от конкретного языка программирования, решаемой задачи и ЭВМ

Программотехника (software engineering) - технология разработки, отладки, верификации и внедрения программного обеспечения.

Все программы по характеру использования и категориям пользователей можно разделить на два класса (рис.3) - утилитарные программы и программные продукты (изделия).

Утилитарные программы ("программы для себя") предназначены для удовлетворения нужд их разработчиков. Чаще всего утилитарные программы выполняют роль сервиса в технологии обработки данных либо являются программами решения функциональных задач, не предназначенных для широкого распространения.

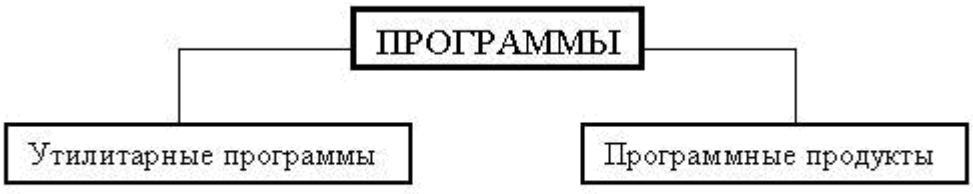


Рис. 3. Классификация программ по категориям пользователей.

Программный продукт - комплекс взаимосвязанных программ для решения определенной проблемы (задачи) массового спроса, подготовленный к реализации как любой вид промышленной продукции. Программные продукты предназначены для удовлетворения потребностей пользователей, широкого распространения и продажи.

При разработке для массового распространения фирма-разработчик, с одной стороны, должна обеспечить универсальность выполняемых функций обработки данных, с другой стороны, гибкость и настраиваемость программного продукта на условия конкретного применения.

Программный продукт разрабатывается на основе промышленной технологии выполнения проектных работ с применением современных инструментальных средств программирования. Специфика заключается в уникальности процесса разработки алгоритмов и программ, зависящего от характера обработки информации и используемых инструментальных средств. На создание программных продуктов затрачиваются значительные ресурсы - трудовые, материальные, финансовые; требуется высокая квалификация разработчиков.

Как правило, программные продукты требуют сопровождения, которое осуществляется специализированными фирмами - распространителями программ (дистрибьюторами), реже - фирмами-разработчиками.

Сопровождение программного продукта - поддержка работоспособности программного продукта, переход на его новые версии, внесение изменений, исправление обнаруженных ошибок и т.п.

Время и затраты на разработку программных продуктов не могут быть определены с большой степенью точности заранее. Основными характеристиками программ являются:

- алгоритмическая сложность (логика алгоритмов обработки информации);
- состав и глубина проработки реализованных функций обработки;
- полнота и системность функций обработки;
- объем файлов программ;
- требования к операционной системе и техническим средствам обработки со стороны программного средства;
- объем дисковой памяти;
- размер оперативной памяти для запуска программ;
- тип процессора;
- версия операционной системы;
- наличие вычислительной сети и др.

Спецификой программных продуктов (в отличие от большинства промышленных изделий) является также и то, что их эксплуатация должна выполняться на правовой основе - лицензионные соглашения между разработчиком и пользователями с соблюдением авторских прав разработчиков программных продуктов.

В настоящее время бурно развивается направление, связанное с технологией создания программных продуктов. Это обусловлено переходом на промышленную технологию производства программ, стремлением к сокращению сроков, трудовых и материальных затрат на производство и эксплуатацию программ, обеспечению гарантированного уровня их качества. Это направление часто называют программотехникой.

Программотехника (software engineering) - технология разработки, отладки, верификации и внедрения программного обеспечения.

Инструментарии технологии программирования - программные продукты поддержки (обеспечения) технологии программирования.

В рамках этих направлений сформировались следующие группы программных продуктов:

- средства для создания приложений, включающие:

локальные средства, обеспечивающие выполнение отдельных работ по созданию программ;

интегрированные среды разработчиков программ, обеспечивающие выполнение комплекса взаимосвязанных работ по созданию программ;

- CASE-технология (Computer-Aided System Engineering), представляющая методы анализа, проектирования и создания программных систем и предназначенная дли автоматизации процессов разработки и реализации информационных систем.

Средства для создания приложений на рынке программных продуктов наиболее представительны и включают языки и системы программирования, а также инструментальную среду пользователя.

Язык программирования - формализованный язык для описания алгоритма решения задачи на компьютере.

Средства для создания приложений - совокупность языков и систем программирования, а также различные программные комплексы для отладки и поддержки создаваемых программ.

Интегрированные среды разработки программ. Дальнейшим развитием локальных средств разработки программ, которые объединяют набор средств для комплексного применения на всех технологических этапах создания программ, являются интегрированные программные среды разработчиков. Основное назначение инструментария данного вида - повышение производительности труда программистов, автоматизация создания кодов программ, обеспечивающих интерфейс пользователя графического типа, разработка приложений для архитектуры клиент-сервер, запросов и отчетов.

CASE-технология - программный комплекс, автоматизирующий весь технологический процесс анализа, проектирования, разработки и сопровождения сложных программных систем.

Средства CASE-технологии - относительно новое, сформировавшееся на рубеже 80-х г направление. Массовое применение затруднено крайне высокой стоимостью и предъявляемыми требованиями к оборудованию рабочего места разработчика.

Средства CASE-технологий делятся на две группы:

- встроенные в систему реализации - все решения по проектированию и реализации привязаны к выбранной системе управления базами данных (СУБД);

- независимые от системы реализации - все решения по проектированию ориентированы на унификацию начальных этапов жизненного цикла и средств их документирования, обеспечивают большую гибкость в выборе средств реализации.

Основное достоинство CASE-технологии - поддержка коллективной работы над проектом за счет возможности работы в локальной сети разработчиков, экспорта/импорта любых фрагментов проекта, организационного управления проектом.

Другой класс CASE-технологий поддерживает только разработку программ, включая:

- автоматическую генерацию кодов программ на основании их спецификаций;

- проверку корректности описания моделей данных и схем потоков данных;

- документирование программ согласно принятым стандартам и актуальному состоянию проекта;

- тестирование и отладку программ.

Кодогенерация программ выполняется двумя способами; создание каркаса программ и создание полного продукта. Каркас программы служит для последующего ручного варианта редактирования исходных текстов, обеспечивая возможность вмешательства программиста; полный продукт не редактируется вручную.

В рамках CASE-технологий проект сопровождается целиком, а не только его программные коды. Проектные материалы, подготовленные в CASE-технологии, служат заданием программистам, а само программирование скорее сводится к кодированию - переводу на определенный язык структур данных и методов их обработки, если не предусмотрена автоматическая кодогенерация.

Большинство CASE-технологий использует также метод "прототипов" для быстрого создания программ на ранних этапах разработки. Кодогенерация программ осуществляет-

ся автоматически - до 85 - 90% объектных кодов и текстов на языках высокого уровня, а в качестве языков наиболее часто используются Ада, Си, Кобол.

*Основные характеристики программных продуктов**

Алгоритм - система точно сформулированных правил, определяющая процесс преобразования допустимых исходных данных (входной информации) в желаемый результат (выходную информацию) за конечное число шагов.

Алгоритм решения задачи имеет ряд обязательных свойств:

-дискретность - разбиение процесса обработки информации на более простые этапы (шаги выполнения), выполнение которых компьютером или человеком не вызывая затруднений;

-определенность алгоритма - однозначность выполнения каждого отдельного шага преобразования информации;

-выполнимость - конечность действий алгоритма решения задач, позволяющая получить желаемый результат при допустимых исходных данных за конечное число шагов;

-массовость - пригодность алгоритма для решения определенного класса задач.

В алгоритме отражаются логика и способ формирования результатов решения с указанием необходимых расчетных формул, логических условий, соотношений для контроля достоверности выходных результатов. В алгоритме обязательно должны быть предусмотрены все ситуации, которые могут возникнуть в процессе решения комплекса задач.

Алгоритм решения комплекса задач и его программная реализация тесно взаимосвязаны. Специфика применяемых методов проектирования алгоритмов и используемых при этом инструментальных средств разработки программ может повлиять на форму представления и содержание алгоритма обработки данных.

Программные продукты имеют многообразие показателей качества, которые отражают следующие аспекты:

- насколько хорошо (просто, надежно, эффективно) можно использовать программный продукт;

- насколько легко эксплуатировать программный продукт;

- можно ли использовать программный продукт при изменении условия его применения и др.

Дерево характеристик качества программных продуктов представлено на рис. 4.

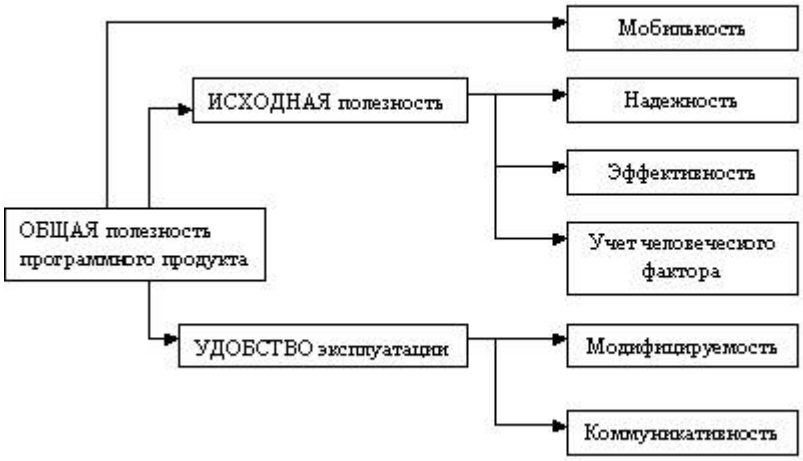


Рис. Дерево характеристик качества программных продуктов.

Мобильность программных продуктов означает их независимость от технического комплекса системы обработки данных, операционной среды, сетевой технологии обработки данных, специфики предметной области и т.п. Мобильный (многоплатформный) про-

граммный продукт может быть установлен на различных моделях компьютеров и операционных систем, без ограничений на его эксплуатацию в условиях вычислительной сети. Функции обработки такого программного продукта пригодны для массового использования без каких-либо изменений.

Надежность работы программного продукта определяется бессбойностью и устойчивостью в работе программ, точностью выполнения предписанных функций обработки, возможностью диагностики возникающих в процессе работы программ ошибок.

Эффективность программного продукта оценивается как с позиций прямого его назначения - требований пользователя, так и с точки зрения расхода вычислительных ресурсов, необходимых для его эксплуатации.

Расход вычислительных ресурсов оценивается через объем внешней памяти для размещения программ и объем оперативной памяти для запуска программ.

Учет человеческого фактора означает обеспечение дружественного интерфейса для работы конечного пользователя, наличие контекстно-зависимой подсказки или обучающей системы в составе программного средства, хорошей документации для освоения и использования заложенных в программном средстве функциональных возможностей, анализ и диагностику возникших ошибок и др.

Модифицируемость программных продуктов означает способность к внесению изменений, например расширение функций обработки, переход на другую техническую базу обработки и т.п.

Коммуникативность программных продуктов основана на максимально возможной их интеграции с другими программами, обеспечении обмена данными в общих форматах представления (экспорт/импорт баз данных, внедрение или связывание объектов обработки и др.).

В условиях существования рынка программных продуктов важными характеристиками являются:

- стоимость,
- количество продаж;
- время нахождения на рынке (длительность продаж);
- известность фирмы-разработчика и программы;
- наличие программных продуктов аналогичного назначения.

Программные продукты массового распространения продаются по ценам, которые учитывают спрос и конъюнктуру рынка (наличие и цены программ-конкурентов). Большое значение имеет проводимый фирмой маркетинг, который включает:

- формирование политики цен для завоевания рынка;
- широкую рекламную кампанию программного продукта;
- создание торговой сети для реализации программного продукта (так называемые дилерские и дистрибьютерные центры);
- обеспечение сопровождения и гарантийного обслуживания пользователей программного продукта, создание горячей линии (оперативный ответ на возникающие в процессе эксплуатации программных продуктов вопросы);
- обучение пользователей программного продукта.

1.1.2. Понятие жизненного цикла программы

Одним из базовых понятий методологии проектирования ИС является понятие жизненного цикла ее программного обеспечения (ЖЦ ПО). ЖЦ ПО - это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.

Основным нормативным документом, регламентирующим ЖЦ ПО, является международный стандарт ISO/IEC 12207. Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

Структура ЖЦ ПО по стандарту ISO/IEC 12207 базируется на трех группах процессов:

- основные процессы ЖЦ ПО (приобретение, поставка, разработка, эксплуатация, сопровождение);
- вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем);
- организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого ЖЦ, обучение).

Разработка включает в себя все работы по созданию ПО и его компонент в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала и т.д. Разработка ПО включает в себя, как правило, анализ, проектирование и реализацию - программирование.

Эксплуатация включает в себя работы по внедрению компонентов ПО в эксплуатацию, в том числе конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала и т.д., и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения, модификацию ПО в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО, обучение персонала и т.п. Обеспечение качества проекта связано с проблемами проверки соответствию требованиям на данном этапе проектирования и тестирования ПО. Проверка позволяет оценить соответствие параметров разработки исходным требованиям. Она частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла ПО, прежде всего процессы разработки и сопровождения ПО. При создании проектов сложных ИС, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникает проблема учета их связей и функций, создания унифицированной структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях жизненного цикла. Общие принципы и рекомендации конфигурационного учета, планирования и управления конфигурациями ПО отражены в проекте стандарта ISO 12207-2 .

К настоящему времени наибольшее распространение получили следующие две основные модели ЖЦ:

- каскадная модель (70-85 г.г.);
- спиральная модель (86-90 г.г.).

Основной характеристикой каскадной модели является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем (рис. 4). Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

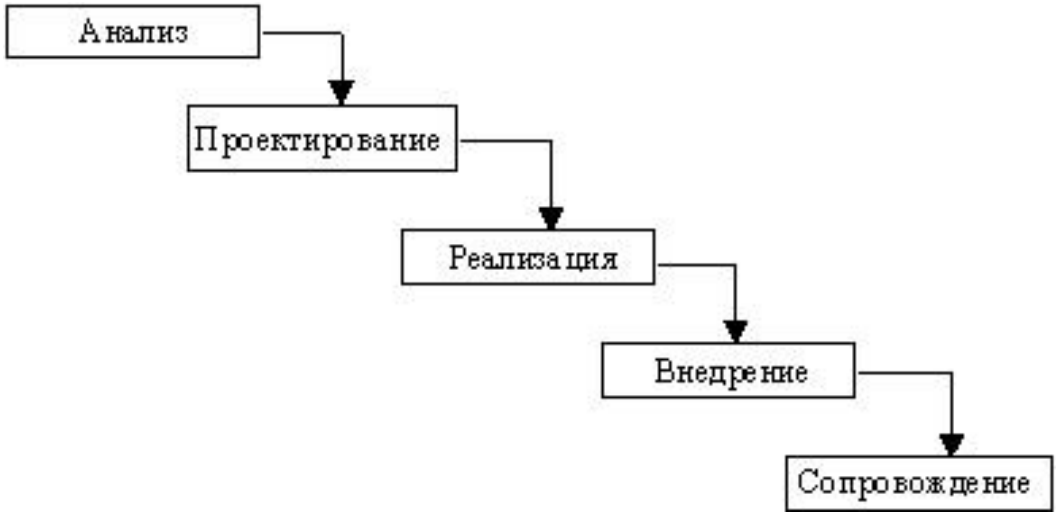


Рис. 4. Каскадная схема разработки ПО

Каскадный подход хорошо зарекомендовал себя при построении ИС, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования. В эту категорию попадают расчетные системы, системы реального времени и другие задачи создания автоматизированных систем. Реальный процесс создания ПО для автоматизированных систем никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений (рис. 5):

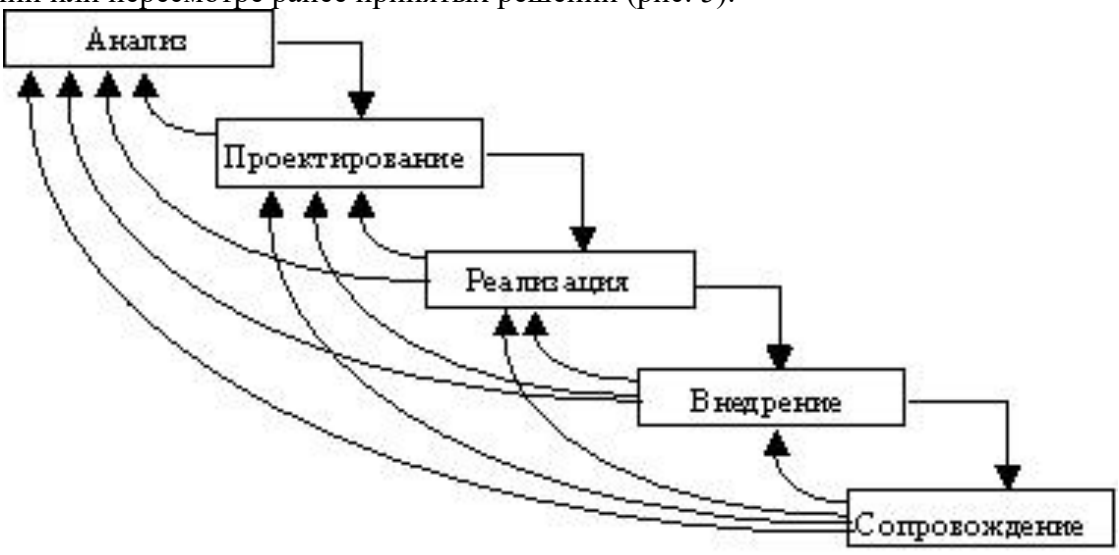


Рис. 5. Реальный процесс разработки ПО по каскадной схеме

Основным недостатком каскадного подхода является существенное затягивание сроков выполнения проекта и других работ в периоде ЖЦ. Согласование результатов с поль-

зователями производится только в точках, планируемых после завершения каждого этапа работ. Требования к ИС представлены в виде технического задания и неизменны на все время ее создания. Таким образом, пользователи могут внести свои замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО, пользователи получают систему, не удовлетворяющую их потребностям. Модели (как функциональные, так и информационные) автоматизируемого объекта могут устареть одновременно с их утверждением.

Для преодоления перечисленных проблем была предложена *спиральная модель* ЖЦ (рис. 6), делающая упор на начальные этапы ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации. Разработка итерациями отражает объективно существующий спиральный цикл создания системы. Неполное завершение работ на начальных этапах позволяет переходить на следующий, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная задача - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований. Основная проблема спирального метода – не нарушать временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе опыта, полученного в предыдущих проектах.

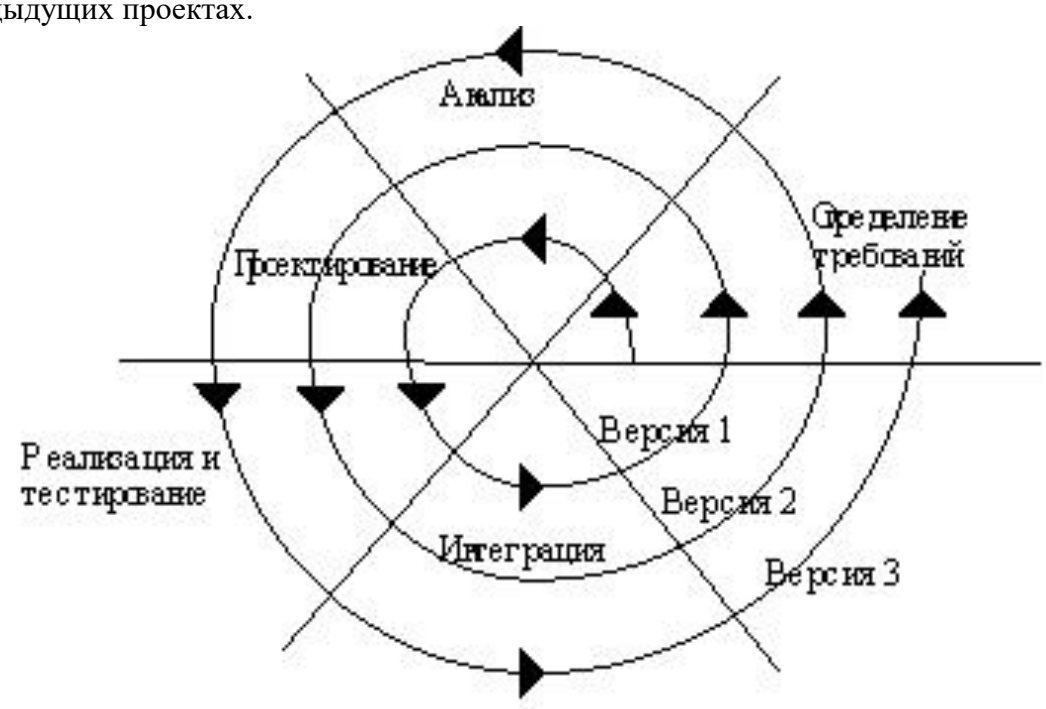


Рис. 6. Спиральная модель ЖЦ

1.1.3. Основные этапы разработки программного обеспечения

Процесс производства программного обеспечения можно разбить на несколько отдельных действий. Способ организации этих действий в виде этапов некоего процесса

может варьироваться в зависимости от выбранной модели. Впрочем, эти действия должны выполняться при реализации любого проекта независимо от того, как они организованы в процессе. Этапы ориентировочно можно представить как анализ, проектирование и реализацию.

Анализ осуществимости

Данный этап часто выполняется фактически до начала процесса производства, в поддержку решения о том, действительно ли нужна новая разработка. Целью его является составление отчета по анализу осуществимости, в котором, наряду с обсуждением компромиссов между затратами и экономическим эффектом, представляются различные сценарии и альтернативные решения. Анализ осуществимости часто используется для принятия организацией решения "создать или приобрести": стоит ли разрабатывать продукт самим или экономически выгоднее купить похожий?

Для выполнения анализа осуществимости специалист по программному обеспечению сначала должен проанализировать проблему, по меньшей мере, на глобальном уровне. Поскольку разработчики ПО не могут быть уверены в том, что их предложение будет принято, они имеют весьма ограниченный стимул для инвестирования средств в анализ проблемы. С другой стороны, если изучение проблемы даст неточные результаты, то ресурсы, необходимые на разработку программного приложения, могут быть недооценены, что выльется в появление серьезных проблем с бюджетом.

На основании описания проблемы во время предварительного анализа, разработчики определяют альтернативные решения. Для каждого предложенного решения оцениваются затраты и даты поставки.

Итак, анализ осуществимости пытается предположить будущие сценарии разработки программного обеспечения. Результатом является документ, в котором должны содержаться, по крайней мере, следующие пункты:

1. Определение проблемы.
2. Альтернативные решения и ожидаемые от них преимущества.
3. Необходимые ресурсы, затраты и сроки поставки для каждого предложенного альтернативного решения.

Выявление, понимание и спецификация требований

Между разработчиками и заказчиками существует соглашение о том, что процедуры выявления, понимания и спецификации требований являются наиболее критичными аспектами процесса программной инженерии. В самом деле, дисциплина *выработки требований* направлена на создание стандартных и систематических методов для выявления, документирования, классификации и анализа требований.

Спецификация ПО – формализованное представление сервисов, которыми будет обладать создаваемое ПО, а также ограничений, налагаемых на функциональные возможности и разработку ПО.

В спецификации требований специалист должен описать, какие качества должно демонстрировать приложение, а не *способ получения* этих качеств в процессе проектирования и реализации. Например, необходимо определить выполняемые приложением функции без указаний конкретной распределенной архитектуры, модульной структуры или алгоритмов, которые должны применяться в решении.

Как уже отмечалось, разрабатываемое программное приложение очень часто является частью более общей системы. Критичной операцией в этом смысле является выделение требований к программному обеспечению из требований всей системы. Требования к программе — это то, чему должно удовлетворять программное решение. Они определяют обязанности программных компонентов в рамках всего системного решения.

Основная цель деятельности по определению требований — точное понимание взаимодействия между разрабатываемым приложением и его внешним окружением. Таким окружением может быть, скажем, физический завод, работу которого программное приложение призвано автоматизировать и контролировать, либо это может быть библиотека,

где библиотекари используют систему для регистрации в каталогах новых поступлений, выдачи книг читателям и где читатели могут просматривать каталоги в поиске нужных книг.

Результатом деятельности по составлению требований является *документ спецификации требований,* описывающий результаты анализа. Цель этого документа двоякая: с одной стороны, он должен быть проанализирован и согласован разными участниками на предмет того, что учтены пожелания всех заказчиков, а с другой — он используется разработчиками для создания решения, удовлетворяющего требованиям.

Еще одной возможной составляющей формирования требований является определение *плана испытаний системы.* Во время тестирования системы фактически проверяется выполнение ею заданных требований. Поэтому способ, которым можно этого в конечном итоге добиться — согласование с заказчиком на стадии системного тестирования и оформление вместе с документом спецификации требований.

Ниже приведен возможный перечень пунктов документа спецификации требований, который может быть руководством специалиста по программному обеспечению:

1.Предметная область. Краткое описание предметной области приложения и целей, которых необходимо достичь при разработке конечного продукта.

2.Функциональные требования. Описывают действия программного продукта, используя неформальные, полуформальные, формальные представления либо их комбинацию.

3.Нефункциональные требования. Их можно классифицировать по следующим категориям: надежность (работоспособность, целостность, безопасность, защищенность и т. д.); точность результатов; производительность; вопросы взаимодействия человека с компьютером; эксплуатационные ограничения; физические ограничения; переносимость и др.

4.Требования к процессу разработки и сопровождения. Сюда входят процедуры управления качеством (в частности процедуры тестирования системы), приоритеты необходимых функций, возможные изменения процедур обслуживания системы и прочие требования.

Определение архитектуры программного обеспечения и рабочий проект

Проектирование — это вид деятельности, при котором разработчики структурируют программное приложение на разных уровнях его детализации. Результатом является *документ технических требований на проектирование,* содержащий описание архитектуры программного продукта.

Кодирование и тестирование модулей

Написание кода и тестирование модулей — операции, посредством которых пишутся программы на каком-либо языке программирования. Кодирование и тестирование модулей составляли единственную общепризнанную фазу процесса разработки в прежние времена, хотя это всего лишь один из нескольких этапов любого процесса структурного проектирования. Результатом этой деятельности является реализованная и протестированная коллекция модулей.

Сборка и системное тестирование

Интегрирование (сборка) заключается в компоновке программного приложения из набора отдельно разработанных и протестированных компонентов. Сборка не всегда рассматривается как операция, отдельная от кодирования. Фактически пошаговые разработки могут постепенно интегрировать и тестировать компоненты по мере их разработки. Несмотря на то, что два этих этапа можно объединить, они принципиально различаются по масштабу проблем, которые призваны решать: первая относится к локальному программированию, тогда как вторая — к программированию системы в целом.

Комплексное тестирование включает в себя тестирование наборов модулей по мере их объединения, при условии предварительного тестирования каждого модуля в отдельности.

Поставка, развертывание и сопровождение ПО

По завершении разработки программного приложения остается выполнить еще определенное количество операций. Во-первых, программный продукт необходимо доставить заказчику. Чаще всего это осуществляется в два этапа. На первом этапе, предваряя официальный выпуск, приложение поставляется членам отобранной группы заказчиков. Целью этой процедуры является проведение своего рода управляемого эксперимента для определения, на основании отзывов пользователей, необходимости внесения изменений в программный продукт до его официального выпуска. Такой вид системного тестирования, выполняемого выбранными заказчиками, называется *бета-тестированием***.**

Техническое обслуживание заключается в исправлении ошибок, оставшихся в системе *(корректирующее сопровождение),* в адаптации приложения к изменениям внешней среды **(***настраивающее сопровождение),* а также в совершенствовании, изменении или добавлении в программу новых функций и качеств *(усовершенствующее сопровождение).* Не стоит забывать, что цена сопровождения часто превышает 60 % от общей цены продукта и что до 20 % затрат на сопровождение составляет доля корректирующего и настраивающего сопровождения, а 50 % приходится на долю усовершенствующего сопровождения. На основании этой статистики можно сделать вывод о том, что *развитие* здесь, возможно, — более уместный термин, нежели сопровождение (хотя последний используется чаще).

Другой тип классификации затрат на сопровождение был описан в работе Линца (Lienz) и Свонсона (Swanson) в 1980 г. Их анализ показал, что порядка 42 % затрат относятся на внесение изменений в требования пользователей, 17 % — на изменение формата данных, 12 % — на устранение аварийных неполадок, 9 % — на отладку процедур, 6 % — на модификацию аппаратных средств, 5 % — на исправление документации, 4 % — на повышение производительности и остальное — на прочие причины.

В общем случае, в отношении технического сопровождения можно сделать следующие выводы.

– Как уже рассматривалось раньше, требования являются основным источником проблем сопровождения как по причине сложности их описания, так и по причине их постоянного изменения.

– Довольно много ошибок не исправляется до поставки системы заказчику. Это — серьезная проблема, поскольку, чем позже обнаружена ошибка, тем дороже обходится ее исправление. Понятно, что предпочтительнее и дешевле исправлять ошибки требований во время анализа, нежели после развертывания системы, потому что эту же ошибку придется исправлять во всех инсталляциях данной системы.

– Подверженность изменениям — это характерное свойство любого программного продукта, однако поддерживать изменения в программных продуктах довольно сложно.

Вопросы для самопроверки

1. Приведите классификацию задач с позиций специфики разработки и вида программного обеспечения и характеристику каждого класса.
2. Сформулируйте определения основных понятий программного обеспечения: технологии программирования, программного продукта, сопровождения программного продукта.
3. Что такое жизненный цикл ПО?
4. Какой нормативный документ регламентирует ЖЦ ПО?
5. На каких трех группах процессов базируется структура ЖЦ ПО?
6. Опишите процесс разработки ЖЦ ПО
7. Опишите процесс эксплуатации ЖЦ ПО
8. Опишите процесс управления проектом ЖЦ ПО
9. Опишите процесс управления конфигурацией ЖЦ ПО
10. Опишите этапы процесса проектирования ЖЦ ПО
11. Каким требованиям должна удовлетворять функциональная спецификация?

12. Опишите основные характеристики и структуру каскадной модели ЖЦ

13. Назовите недостатки каскадного подхода

14. Изобразите схему реального процесса создания ПО

15. Опишите основные характеристики и структуру спиральной модели ЖЦ

16. Охарактеризуйте этап разработки программного обеспечения, на котором выполняется анализ осуществимости.

17. Охарактеризуйте этап разработки программного обеспечения, на котором выполняется проектирование ПО.

18. Охарактеризуйте этап разработки программного обеспечения, на котором выполняется реализация ПО.

19. Дайте определение спецификации ПО. Из каких пунктов может состоять этот документ?

20. Перечислите типы программных продуктов, относящихся к инструментарию технологии программирования.

Самостоятельная работа

1. Реферирование темы «Вспомогательные процессы жизненного цикла программного продукта» (3 ч.)
2. Аналитический обзор литературы по теме «Каскадная модель жизненного цикла разработки программного продукта» (2ч.)
3. Подготовка доклада по теме «Модель прототипирования жизненного цикла разработки программного продукта» (2 ч.)

Тема 1.2. Стадии разработки программ и программной документации

1.2.1. Методы и средства разработки технической документации. Понятие о ЕСПД (ГОСТ 19.XXX-XX, ГОСТ 34.XXX-XX)

Единая система программной документации (ЕСПД) — комплекс государственных стандартов Российской Федерации, устанавливающих взаимосвязанные правила разработки, оформления и обращения программ и программной документации.

В стандартах ЕСПД устанавливают требования, регламентирующие разработку, сопровождение, изготовление и эксплуатацию программ, что обеспечивает возможность:

- унификации программных изделий для взаимного обмена программами и применения ранее разработанных программ в новых разработках;
- снижения трудоемкости и повышения эффективности разработки, сопровождения, изготовления и эксплуатации программных изделий;
- автоматизации изготовления и хранения программной документации.

В нашей стране жизненный цикл разработки ПО установлен стандартом ГОСТ 19.102-77 Стадии разработки программ и программной документации и содержит следующие стадии и этапы

1. Техническое задание (ТЗ).
2. Эскизный проект (ЭП).
3. Технический проект (ТП).
4. Рабочий проект (РП).
5. Внедрение.

В табл.1 показаны стадии разработки и этапы, их составляющие.

Неверно предполагать, что жизненный цикл разработки ПО согласно ГОСТ 19.102-77 есть последовательное выполнение стадий и этапов, определенных в таблице 1. В реальном жизненном цикле трудно провести четкую и определенную границу между этапами, а сам процесс создания ПО является итеративным: после завершения некоторого этапа почти всегда есть необходимость в коррекции уже выполненных этапов и стадий с целью

внесения уточнений. При разработке принципиально нового ПО иногда бывает необходимо осуществить пробную реализацию с целью получения информации, требующейся для принятия решения на некоторой стадии.

Таблица 1.

Стадии разработки	Этапы работ
Техническое задание	1. Обоснование необходимости разработки программ. 2. Выполнение научно-исследовательских работ (НИР). 3. Разработка и утверждение технического задания.
Эскизный проект	1. Разработка эскизного проекта. 2. Утверждение эскизного проекта.
Технический проект	1. Разработка технического проекта. 2. Утверждение технического проекта.
Рабочий проект	1. Разработка программы. 2. Разработка программной документации. 3. Испытание программы.
Внедрение	1. Подготовка и передача программы.

Специалистам в области разработки ПО известно, что наиболее важными стадиями в жизненном цикле разработки являются начальные, так как ошибки, допущенные на них, требуют значительных затрат на исправление на конечных стадиях.

1. Техническое задание

На стадии Техническое задание выполняются следующие работы, входящие в состав соответствующих этапов.

1.1. Обоснование необходимости разработки программ:

постановка задачи;

сбор исходных материалов;

выбор и обоснование критериев эффективности и качества;

обоснование необходимости проведения НИР.

1.2. Выполнение научно-исследовательских работ:

определение структуры входных и выходных данных;

предварительный выбор методов решения задач;

обоснование целесообразности применения ранее разработанных программ;

определение требований к техническим средствам;

обоснование принципиальной возможности решения поставленных задач.

1.3. Разработка и утверждение технического задания:

определение требований к программе;

разработка технико-экономического обоснования разработки программы;

определение стадий, этапов и сроков разработки программы и документации на нее;

выбор языков программирования;

определение необходимости проведения НИР на последующих стадиях;

согласование и утверждение ТЗ.

Результатом выполнения данной стадии является **техническое задание**, оформленное в соответствии с ГОСТ 19.105-78 (изм. 09.1981.) Общие требования к программным документам и ГОСТ 19.106-78 Общие требования к программным документам, выполненным печатным способом на листах формата 11 и 12 (по ГОСТ 2.301-68).

2. Эскизный проект

Основные этапы и содержание работ на стадии Эскизный проект приведены в таблице 2.

Таблица 2

Этапы работ	Содержание

Разработка ЭП	1. Предварительная разработка структуры входных и выходных данных. 2. Уточнение методов решения задач. 3. Разработка общего описания алгоритма решения задачи. 4. Разработка технико-экономического обоснования.
Утверждение ЭП	1. Разработка пояснительной записки. 2. Согласование и утверждение эскизного проекта.

Конкретное содержание работ стадии эскизного проекта и их объем определяет степень сложности разрабатываемого ПО. Результатом выполнения данной стадии является полное описание архитектуры ПО. Как правило, это описание делается на нескольких уровнях иерархии. На верхнем уровне детализации выделяются основные подсистемы, которым присваиваются имена, устанавливаются связи между подсистемами, их функции, получаемые путем декомпозиции предполагаемых функций ПО. Затем процедура декомпозиции выполняется для каждой подсистемы, выделяются модули, составляющие данную подсистему. В конечном итоге, получается иерархически организованная система, состоящая из уровней, каждый из которых представляет собой совокупность взаимосвязанных модулей.

Единицы, выделяемые на различных иерархических уровнях функциональной архитектуры системы, определяются по усмотрению разработчика. Стандарты ЕСПД различают программные единицы только с точки зрения их документирования.

Результаты эскизного проекта отображаются в документе Пояснительная записка к эскизному проекту, оформленному в соответствии с ГОСТ 19.105-78 и ГОСТ 19.404-79.

После утверждения пояснительной записки она становится программным документам, правила дублирования, учета, хранения которого определяется ГОСТ 19.601-78 Общие правила дублирования, обращения, учета и хранения и ГОСТ 19.602-78 Правила дублирования, учета и хранения программных документов, выполненных печатным способом. Последующие стадии и этапы разработки ПО могут выявить необходимость внесения изменений в ЭП. Эти изменения должны быть отражены в пояснительной записке в соответствии с ГОСТ 19.603-78 Общие правила внесения изменений в программные документы и ГОСТ 19.602-78 Правила внесения изменений в программные документы, выполненные печатным способом.

3. Технический проект

Основные этапы и содержание работ на стадии Технический проект приведены в таблице 3.

Содержанием работ на этой стадии является проектирование структуры ПО. Результатом - реализующий заданный и утвержденный в техническом задании комплекс программ как иерархическая структура программных модулей, заданных своими функциональными спецификациями. Форма представления результата - Пояснительная записка к техническому проекту согласно ГОСТ 19.105-78, ГОСТ 19.404-79.

Разработка структуры ПО заключается в выделении всех программных компонентов по функциональным признакам, определение функциональных спецификаций модулей и уточнение внешних функциональных спецификаций, структуры входных и выходных данных, определении операционной среды, языковых средств и конфигурации аппаратных средств.

Спецификации модулей являются внешними характеристиками и содержат все сведения, необходимые вызывающим модулям. На последующих стадиях разработки спецификации оформляются в виде комментариев в начале текста исходной программы модуля. На данной стадии спецификации оформляются в виде комментария на принятом в организации, занимающейся разработкой ПО, языке спецификаций

Таблица 3

Этапы работ	**Содержание**

Разработка ТП	1. Уточнение структуры входных и выходных данных. 2. Разработка алгоритмов решения задач. 3. Определение формы представления входных и выходных данных. 4. Определение синтаксиса и семантики языка. 5. Разработка структуры программы. 6. Окончательное определение конфигурации технических средств.
Утверждение ТП	1. Разработка плана мероприятий по разработке и внедрению программ. 2. Разработка пояснительной записки. 3. Согласование и утверждение ТП.

4. Рабочий проект.

Основные этапы и содержание работ на стадии Рабочий проект приведены в табл. 4.

Таблица 4

Этапы работ	Содержание
Разработка ПО	1. Программирование и отладка программ.
Разработка программной документации	1. Разработка программных документов в соответствии с требованиями ГОСТ 19.101-77.
Испытание ПО	1. Разработка, согласование и утверждение программ и методики испытаний 2. Проведение предварительных государственных, межведомственных приемо-сдаточных и других видов испытаний. 3. Корректировка ПО и программной документации по результатам испытаний.

Содержанием работ на этой стадии является описание ПО на выбранном проблемно-ориентированном языке (кодирование), отладка, разработка, согласование и утверждение порядка и методики испытаний, разработка программных документов, проведение тестирования, корректировка программ и программной документации по результатам тестирования, проведение приемо-сдаточных испытаний. Результат - ПО в форме программной документации, в форме документации на ПО или в форме программного изделия.

5. На стадии **Внедрения** осуществляется подготовка и передача ПО и программной документации для сопровождения и/или изготовления, оформление и утверждение акта о передаче ПО на сопровождение или изготовление, передача ПО в фонд алгоритмов и программ.

Кроме рассмотренного выше жизненного цикла программ, установленного 19 - ой системой стандартов, необходимо иметь в виду, что существует жизненный цикл автоматизированных систем, установленный ГОСТ 34.601-90. Имеет смысл рассмотреть стадии и этапы разработки АС, так как 19-я система стандартов в настоящее время морально устарела и более современным представляется жизненный цикл АС. Анализ жизненных циклов разработки ПО, установленных ГОСТ 19.102 и ГОСТ 34.601, а также международных стандартов, регламентирующих данный процесс, будет приведен ниже.

Стандарт ГОСТ 34.601 распространяется на автоматизированные системы (АС), представляющие собой организационно-технические системы, обеспечивающую выработку решений на основе автоматизации информационных потоков в различных видах деятельности (исследование, проектирование, управление и т.п.), включая их сочетания, создаваемые в организациях, объединениях и на предприятиях. Данный стандарт устанавливает стадии и этапы создания АС.

В процессе разработки АС создают, в общем случае, следующие виды обеспечения: техническое, программное, информационное математическое и др. Проектные решения по программному, техническому и информационному обеспечению реализуют как изделия в виде взаимоувязанной совокупности компонент и комплексов, входящей в состав АС с необходимой документацией. Справочное приложение к ГОСТ 34.601 устанавливает, что

программное обеспечение АС - совокупность программ на носителях информации с программной документацией по ГОСТ 19.101. Таким образом, при разработке программного обеспечения можно пользоваться как стандартом ГОСТ 19.102, так и ГОСТ 34.601.

Процесс создания АС представляет собой совокупность упорядоченных во времени, взаимосвязанных, объединенных в стадии и этапы работ, выполнение которых необходимо и достаточно для создания АС, соответствующей заданным требованиям. Стадии и этапы создания АС выделяются как части процесса создания по соображениям рационального планирования и организации работ, заканчивающих заданным результатом.

Как и в стандарте ГОСТ 19.102, работы по развитию АС осуществляются по стадиям и этапам, применяемым для создания АС. Состав и правила выполнения работ на установленных настоящим стандартом стадиях и этапах определяют в соответствующей документации организаций, участвующих в создании конкретных видов АС.

Стадии и этапы создания АС в общем случае приведены в таблице 5.

Стандартом ГОСТ 34.601 допускается исключать стадию Эскизный проект и отдельные этапы работ на всех стадиях, объединять стадии Технический проект и Рабочая документация в одну стадию Технорабочий проект. В зависимости от специфики создаваемых АС и условий их создания допускается выполнять отдельные этапы работ до завершения предшествующих стадий, параллельное во времени выполнения этапов работ, включение новых этапов работ.

Рассмотрим этапы, касающиеся разработки программного обеспечения.

При Обследование объекта и обоснование необходимости создания АС проводят: сбор данных об объекте автоматизации и осуществляемых видах деятельности; оценку качества функционирования объекта и осуществляемых видов деятельности, выявления проблем, решение которых возможно средствами автоматизации; оценку (технико-экономической, социальной и т.п.) целесообразности создания АС.

На этапе Формирование требований пользователя к АС проводят:

подготовку исходных данных для формирования требований к АС (характеристика объекта автоматизации, описание требований к системе, ограничения допустимых затрат на разработку, ввод в действие и эксплуатацию, эффект, ожидаемый от системы, условия создания и функционирования);

формулировку и оформление требований пользователя к АС.

Таблица 5

Стадии	Этапы работ
1. Формирование требований к АС	1.1. Обследование объекта и обоснование необходимости создания АС 1.2. Формирование требований пользователя к АС 1.3. Оформление отчета о выполненной работе и заявки на разработку АС (тактико-технического задания)
2. Разработка концепции АС	2.1. Изучение объекта 2.2. Проведение необходимых научно - исследовательских работ 2.3. Разработка вариантов концепции АС и выбор варианта концепции АС, удовлетворяющего требованиям пользователя 2.4. Оформление отчета о выполненной работе
3. Техническое задание	3.1. Разработка и утверждение технического задания на создание АС
4. Эскизный проект	4.1. Разработка предварительных проектных решений по системе и ее частям 4.2. Разработка документации на АС и ее части

5. Технический проект	5.1. Разработка проектных решений по системе и ее частям 5.2. Разработка документации на АС и ее части 5.3. Разработка и оформление документации на поставку изделий для комплектования АС и/или технических требований (технических заданий) на их разработку 5.4. Разработка заданий на проектирование в смежных частях проекта объекта автоматизации
6. Рабочая документация	6.1. Разработка рабочей документации на систему и ее части 6.2. Разработка или адаптация программ
7. Ввод в действие	7.1. Подготовка объекта автоматизации к вводу АС в действие 7.2. Подготовка персонала 7.3. Комплектация АС поставляемыми изделиями (программными и техническими средствами, программно-техническими комплексами, информационными изделиями) 7.4. Строительно-монтажные работы 7.5. Пусконаладочные работы 7.6. Проведение предварительных испытаний 7.7. Проведение опытной эксплуатации 7.8. Проведение приемочных испытаний
8. Сопровождение АС	8.1. Выполнение работ в соответствии с гарантийными обязательствами 8.2. Послегарантийное обслуживание

На этапе Оформление отчета о выполненной работе и заявки на разработку АС (тактико-технического задания) проводят оформление отчета о выполненных работах на данной стадии и оформление заявки на разработку АС (тактико-технического задания) или другого заменяющего ее документа с аналогичным содержанием.

При Изучении объекта и Проведении необходимых НИР организация-разработчик проводит детальное изучение объекта автоматизации и необходимые НИР, связанные с поиском путей и оценкой возможности реализации требований пользователя, оформляют и утверждают отчеты о НИР.

Этап Разработка вариантов концепции АС и выбор варианта концепции АС, удовлетворяющего требованиям пользователя направлен на разработку альтернативных вариантов концепции создаваемой АС и планов реализации; оценку необходимых ресурсов на их реализацию и обеспечение функционирования; оценку преимуществ и недостатков каждого варианта; сопоставление требований пользователя и характеристик предлагаемой системы и выбор оптимального варианта; определение порядка оценки качества и условий приемки системы; оценку эффектов, получаемых от системы.

На этапе Оформление отчета о выполненной работе подготавливают и оформляют отчет, содержащий описание выполненных работ на стадии, описание и обоснование предлагаемого варианта концепции системы.

На этапе Разработка и утверждение технического задания на создание АС проводят разработку, оформление, согласование и утверждение технического задания на АС и, при необходимости, технических заданий на части АС.

Результатом выполнения этапа Разработка предварительных проектных решений по системе и ее частям является: функции АС; функции подсистем, их цели и эффекты; состав комплекса задач и отдельных задач; концепции информационной базы, ее укрупненная структура; функции СУБД; состав вычислительной системы; функции и параметры основных программных средств.

На этапе Разработка проектных решений по системе и ее частям обеспечивают разработку общих решений: по системе и ее частям, функционально-алгоритмической структуре системы; по функциям персонала и организационной структуре; по структуре техни-

ческих средств; по алгоритмам решения задач и применяемым языкам; по организации и ведению информационной базы, системе классификации и кодирования информации; по программному обеспечению.

На этапах 4.2 и 5.2 Разработка документации на АС и ее части проводят разработку, оформление, согласование и утверждение документации в объеме, необходимом для описания полной совокупности принятых проектных решений и достаточном для дальнейшего выполнения работ по созданию АС.

На этапе Разработка и оформление документации на поставку изделий для комплектации АС и (или) технических требований (технических заданий) на их разработку проводят: подготовку и оформление документации на поставку изделий для комплектования АС; определение технических требований и составление ТЗ на разработку изделий, не изготавливаемых серийно.

На этапе Разработка заданий на проектирование в смежных частях проекта объекта автоматизации осуществляют разработку, оформление, согласование и утверждение заданий на проектирование в смежных частях проекта объекта автоматизации для проведения строительных, электротехнических, санитарно-технических и других подготовительных работ, связанных с созданием АС.

При выполнении этапа Разработка рабочей документации на систему и ее части осуществляют разработку рабочей документации, содержащей все необходимые и достаточные сведения для обеспечения выполнения работ по вводу АС в действие и ее эксплуатации, а также для поддерживания уровня эксплуатационных характеристик (качества) системы в соответствии с принятыми проектными решениями, ее оформление, согласование и утверждение. Виды документов - по ГОСТ 34.201.

На этапе Разработка и адаптация программ проводят разработку программ и программных средств системы, выбор, адаптацию и (или) привязку приобретенных программных средств, разработку программной документации в соответствии с ГОСТ 19.101.

На этапе Подготовка объекта автоматизации к вводу АС в действие проводят работы по организационной подготовке объекта автоматизации к вводу АС в действие, в том числе: реализацию проектных решений по организационной структуре АС; обеспечение подразделений объекта управления инструктивно-методическими материалами; внедрение классификаторов информации.

Этап Подготовка персонала направлен на обучение персонала и проверку его способности обеспечить функционирование АС.

На этапе Комплектация АС поставляемыми изделиями обеспечивают получение комплектующих изделий серийного и единичного производства, материалов и монтажных изделий. Проводят входной контроль их качества.

На этапе Пусконаладочные работы проводят автономную наладку технических и программных средств, загрузку информации в БД и проверку системы ее ведения; комплексную наладку всех средств системы.

На этапе Проведение предварительных испытаний осуществляют: испытание АС на работоспособность и соответствие ТЗ в соответствии с программой и методикой предварительных испытаний; устранение неисправностей и внесение изменений в документацию на АС, в том числе эксплуатационную в соответствии с протоколом испытаний; оформление акта о приемке АС в опытную эксплуатацию.

На этапе Проведение опытной эксплуатации проводят: опытную эксплуатацию АС; анализ результатов опытной эксплуатации АС; доработку (при необходимости) программного обеспечения АС; дополнительную наладку (при необходимости) технических средств АС; оформление акта о завершении опытной эксплуатации.

На этапе Проведение приемочных испытаний проводят испытания на соответствие ТЗ в соответствии с программой и методикой приемочных испытаний, анализ результатов испытаний АС и устранение недостатков, выявленных при испытаниях, а также оформление акта о приемке АС в постоянную эксплуатацию.

На этапе Выполнение работ в соответствии с гарантийными обязательствами осуществляются работы по устранению недостатков, выявленных при эксплуатации АС в течение установленных гарантийных сроков, внесению необходимых изменений в документацию на АС.

На этапе Послегарантийное обслуживание осуществляются работы по анализу функционирования системы, выявлению отклонений фактических эксплуатационных характеристик АС от проектных значений, установлению причин этих отклонений, устранению выявленных недостатков и обеспечение стабильности эксплуатационных характеристик АС, внесению необходимых изменений в документацию на АС.

Требования к содержанию документов, разрабатываемых на каждой стадии и этапе, устанавливает руководящий документ РД-50-34.698-90. Кроме того, необходимо использовать для разработки соответствующие стандарты Единой системы программной документации, Единой системы конструкторской документации и ГОСТ 34.602. Виды и комплектность документов регламентированы в ГОСТ 34.201.

1.2.2. Графические языки спецификаций

Методы спецификации программ в CASE-системах

Методологической основой верхних CASE-систем являются методы спецификации программ, то есть, описания задачи, которую должна решать программа.

Для того, чтобы спецификация могла быть проанализирована системой, она должна иметь определенный язык, описание должно придерживаться определенных правил.

Средства спецификации, применяемые в CASE-системах могут применяться не только для специфицирования программ, но и для описания и анализа таких отраслей деятельности, как бизнес (примером CASE-системы, ориентированной на анализ и оптимизацию бизнес-процессов, является BPWin).

Для удобства и простоты работы, в CASE-системах используются графические языки спецификаций и описания процессов и структур данных. Наиболее распространенными являются:

- диаграммы потоков данных (data flow diagram);
- диаграммы объектов-связей, называемые еще диаграммами «сущность-связь» (ER-диаграммы, entity relation diagram);
- диаграммы переходов-состояний (state transition diagram).

Для описания иерархических структур широко используются разнообразные деревья. Диаграмма потоков данных применяется, как правило, для описания систем обработки данных в бизнесе, ER-диаграммы используются при описании информационных структур баз данных, диаграммы переходов-состояний служат для описания систем реального времени.

Диаграмма потоков данных состоит из трех типов узлов: узлов обработки данных, узлов хранения данных и внешних узлов, представляющих внешние, по отношению к используемой диаграмме, источники или потребители данных. Дуги в диаграмме соответствуют потокам данных, передаваемых от узла к узлу. Они помечены именами соответствующих данных. Описание процесса, функции или системы обработки данных, соответствующих узлу диаграммы, может быть представлено диаграммой следующего уровня детализации, если процесс достаточно сложен.

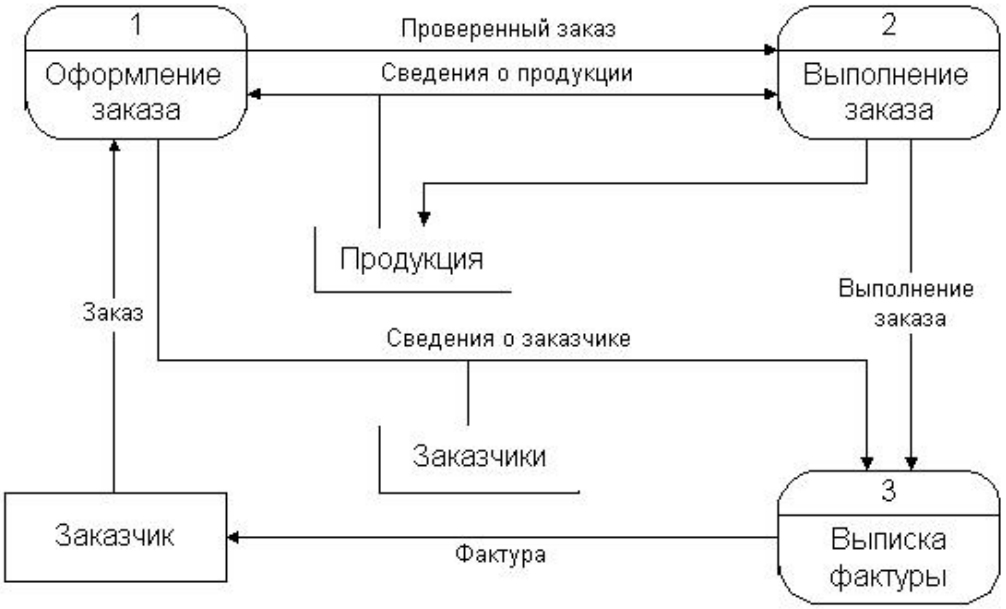


Рис. 7. Диаграмма потоков данных системы обработки заказов

Развертка диаграмм одного уровня детализации представляется в виде дерева разверток, напоминающего диаграмму Джексона. Другим аналогом дерева развертки является HIPO-технология.

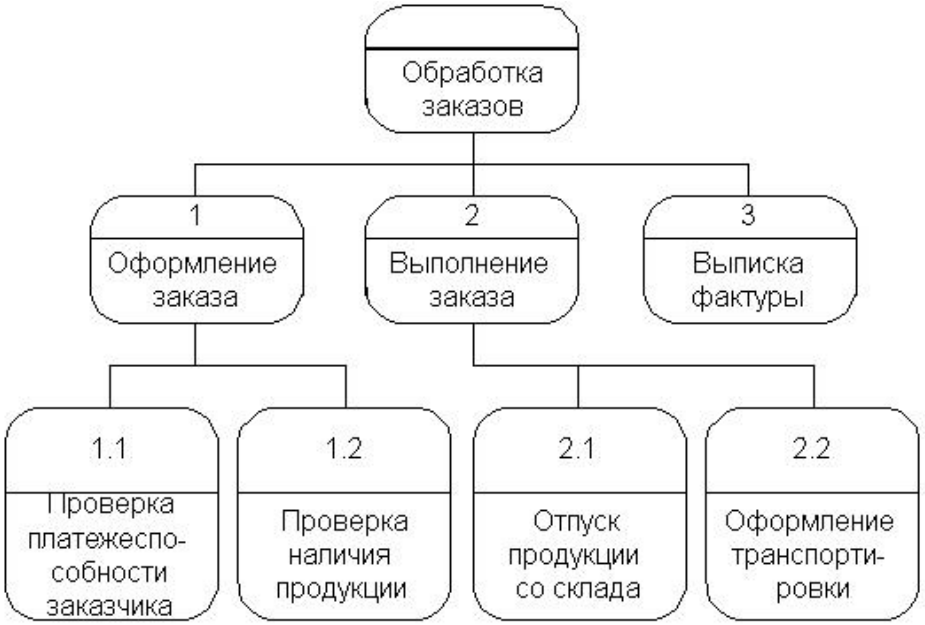


Рис. 8. Дерево разверток системы обработки заказов

Диаграмма переходов-состояний

В диаграммах такого вида узлы соответствуют состояниям динамической системы, а дуги – переходу системы из одного состояния в другое. Узел, из которого выходит дуга, является начальным состоянием, узел, в который дуга входит – следующим. Дуга помечается именем входного сигнала или события, вызывающего переход, а так же сигналом или действием, сопровождающим переход.

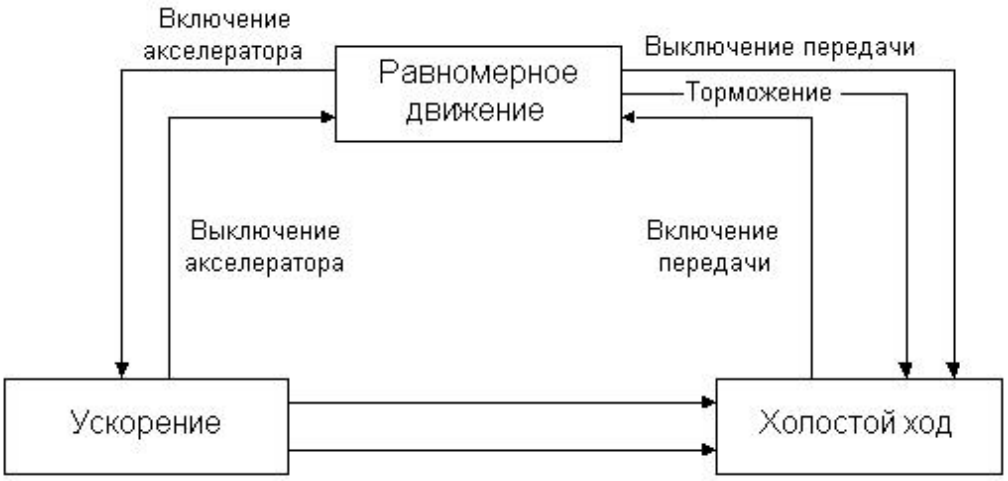


Рис. 9. Диаграмм переходов-состояний двигателя автомобиля

На рисунке приведен возможный пример состояний двигателя автомобиля в зависимости от состояния управления.

Диаграмма сущностей-связей

Такая диаграмма содержит узлы двух типов: для объектов (сущностей) и для связей. Дуги соединяют узел отношение с его аргументами. Классическим примером использования диаграмм сущностей-связей является описание концептуальных моделей в базах данных. Продемонстрируем потенциальные возможности использования ER-диаграмм в этом разрезе:

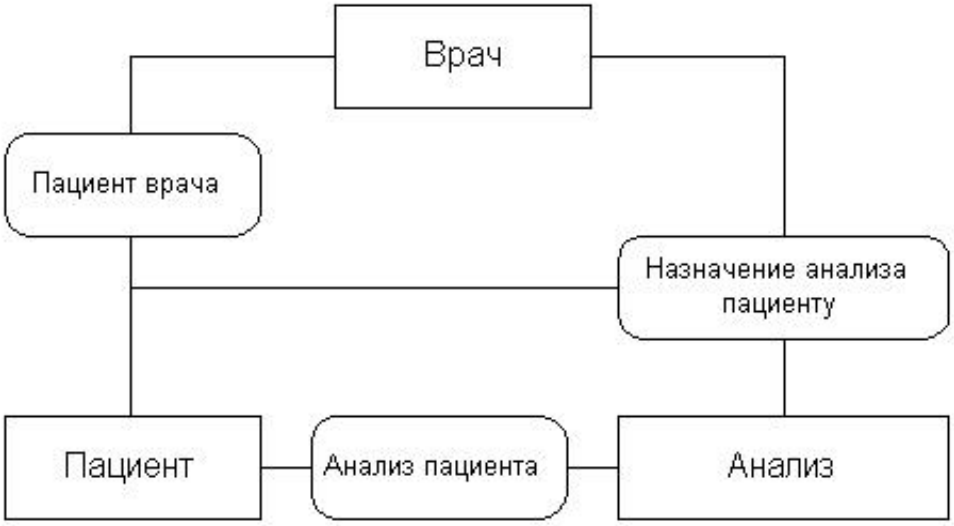


Рис. 10. Диаграмма объектов связей больничных баз данных

Здесь схематично изображена модель базы данных больницы. Узлы-прямоугольники являются сущностями, а узлы со скругленными углами – связями.

Обобщение методов спецификации программ для CASE-систем.

Как правило, CASE-система поддерживает тот из указанных методов специфицирования, для разработок в какой отрасли она предназначена. Некоторые системы поддерживают одновременно несколько систем специфицирования. Но все эти системы имеют общие черты. Все они удобны и интуитивно понятны, и, как правило, имеют графический интерфейс, направленный на упрощение разработки. Все они поддерживают работу с иерархическими структурами, автоматически обновляя зависимые объекты при изменении влияющих. Все системы включают средства анализа, направленные на поиск и детальную проработку «узких» мест модели. Сюда входят анализ так называемого «крити-

ческого пути», ценовая оценка (в смысле требований к ресурсам, времени и т.п.), оптимизация. Это становится возможным благодаря тому, что вся модель разработана на однотипном «языке», специально предназначенном для описания различных моделей.

Вопросы для самопроверки

1. Перечислите стадии разработки программного продукта и этапы, их составляющие
2. Какие работы выполняются на стадии Техническое задание.
3. Опишите основные этапы и содержание работ на стадии Эскизный проект.
4. Опишите основные этапы и содержание работ на стадии Технический проект.
5. Опишите основные этапы и содержание работ на стадии Рабочий проект.
6. Какие работы выполняются на стадии Рабочий проект?
7. Каким стандартом кроме ГОСТ 19.102 можно пользоваться при разработке программного обеспечения. Почему?
8. Опишите стадии и этапы создания АС.
9. Назовите графические языки спецификаций.
10. Охарактеризуйте диаграмму потоков данных. В каких случаях она используется?
11. Охарактеризуйте диаграмму переходов-состояний. В каких случаях она используется?
12. Охарактеризуйте диаграмму сущностей-связей. В каких случаях она используется?
13. На чем основано обобщение методов спецификации программ для CASE-систем

Самостоятельная работа

1. Составление сводной таблицы «Основные ГОСТы, использующиеся для описания жизненного цикла программного продукта» (2 ч.)
2. Составление сводной таблицы «Типы технических решений и документация на них». (2 ч.)

Тема 1.3. Методы проектирования программных продуктов

1.3.1. Принципы системного проектирования

К концу 20 века не только существенно возросла сложность проектируемых объектов, но и их воздействие на общество и окружающую среду, тяжкость последствий аварий из-за ошибок разработки и эксплуатации, высокие требования к качеству и цене, сокращению сроков выпуска новой продукции. Необходимость учета этих обстоятельств заставляла вносить изменения в традиционный характер и методологию проектной деятельности.

При создании объектов их уже необходимо было рассматривать в виде *систем*, то есть комплекса взаимосвязанных внутренних элементов с определенной структурой, широким набором свойств и разнообразными внутренними и внешними связями. Сформировалась новая проектная идеология, получившая название системного проектирования.

Системное проектирование комплексно решает поставленные задачи, принимает во внимание взаимодействие и взаимосвязь отдельных объектов-систем и их частей как между собой, так и с внешней средой, учитывает социально-экономические и экологические последствия их функционирования. Системное проектирование основывается на тщательном совместном рассмотрении объекта проектирования и процесса проектирования, которые в свою очередь включают ещё ряд важных частей

Принципы системного проектирования

Системное проектирование должно базироваться на системном подходе. В настоящее время ещё нельзя утверждать, что известны их полные состав и содержание применительно к проектной деятельности, однако можно сформулировать наиболее важные из них:

- Практическая полезность:

- деятельность должна быть *целенаправленной*, устремленной на удовлетворение действительных потребностей реального потребителя или определенной социальной, возрастной или иной групп людей;
- деятельность должна быть *целесообразной*. Важно вскрыть причины, препятствующие использованию существующих объектов для удовлетворения новых потребностей, выявить вызывающие их ключевые противоречия и сконцентрировать усилия на решении главных задач;
- деятельность должна быть *обоснованной и эффективной*. Разумным будет использование не любого решения задачи, а поиск *оптимального варианта*;

• Единство составных частей:

- целесообразно любой объект, сложный ли он или простой, рассматривать как *систему*, внутри которой можно выделить логически связанные более простые части — *подсистемы*, единство частных свойств которых и образует качественно новые свойства объекта-системы;
- разрабатываемые объекты предназначены для людей, ими создаются и эксплуатируются. Поэтому человек также обязан рассматриваться в качестве одной из взаимодействующих систем. При этом должно приниматься во внимание не только физическое взаимодействие, но и духовно-эстетическое воздействие;
- внешняя, или как её ещё называют — *жизненная среда*, также должна рассматриваться в качестве системы, взаимосвязанной с проектируемым объектом;

• Изменяемость во времени:

- учёт этапов жизненного цикла объекта;
- учёт истории и перспектив развития и применения разрабатываемого объекта, а также областей науки и техники, на достижениях которых базируются соответствующие разработки.

Нисходящее и восходящее проектирование

Ведение разработки объекта последовательно от общих черт к детальным называется *нисходящим проектированием*. Его результатом будут требования к отдельным частям и узлам. Возможен ход разработки от частного к общему, что образует процесс *восходящего проектирования*. Такое проектирование встречается, если одна или несколько частей уже являются готовыми (покупными или уже разработанными) изделиями.

Нисходящее и восходящее проектирование обладают своими достоинствами и недостатками. Так, при нисходящем проектировании возможно появление требований, впоследствии оказывающихся нереализуемыми по технологическим, экологическим или иным соображениям. При восходящем проектировании возможно получение объекта, не соответствующего заданным требованиям. В реальной жизни, вследствие итерационного характера проектирования, оба его вида взаимосвязаны.

Например, разрабатывая при нисходящем проектировании автомобиль (от общей схемы к его частям, например, — к мотору), необходимо увязать общую компоновку с размерами и мощностью уже выпускаемых двигателей. В противном случае придётся разрабатывать применительно к данной компоновке новый двигатель, либо изменять первоначальные варианты его расположения или схему компоновки всего автомобиля.

1.3.2. Объектно-ориентированное проектирование программных продуктов

Деловые компьютерные программы, используемые в бизнесе и научных исследованиях, строятся на основе моделей реального мира. В таких моделях реальным процессам и системам ставится в соответствие совокупность величин, называемых переменными состояния. Изменение состояния исследуемого процесса или системы отображается изменением переменных состояния модели. В общем случае математическая модель описывается набором переменных состояния и отношениями (связями) между этими переменными. Переменные состояния могут быть как числовыми, так и не числовыми, в том числе словами и предложениями естественного языка. Одним из подходов, обеспечивающих структурирование математической модели и упрощение ее программирования, является объектный подход, в котором реальный процесс или система представляются совокупностью объектов, взаимодействующих друг с другом.

Понятию ***"объект"*** сопоставляют ряд дополняющих друг друга определений. Ниже приведены некоторые из них.

Объект - это осязаемая реальность, характеризующаяся четко определяемым поведением.

Объект - особый опознаваемый предмет, блок или сущность (реальная или абстрактная), имеющая важное функциональное назначение в данной предметной области.

Объект может быть охарактеризован структурой, состоянием объекта, его поведением и индивидуальностью.

Состояние объекта определяется перечнем всех возможных (обычно статических) свойств и текущими значениями (обычно динамическими) каждого из этих свойств. Свойства объекта характеризуются значениями его параметров.

Поведение объекта описывает, как объект воздействует на другие объекты или как он подвергается воздействию со стороны других объектов с точки зрения изменения его собственного состояния и состояния других объектов. Говорят также, что поведение объекта определяется его действиями.

Определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию называют *операцией*. В объектно-ориентированных языках программирования операции называют *методами*. Можно выделить пять типов операций:

- конструктор, создание и инициализация объекта;
- деструктор, разрушающий объект;
- модификатор, изменяющий состояние объекта;
- селектор для доступа к переменным объекта без их изменения;
- итератор для доступа к содержанию объекта по частям в определенной последовательности.

Известна и другая классификация методов объекта, когда выделяют функции управления, реализации, доступа и вспомогательные функции.

Под индивидуальностью объекта понимают свойство объекта, позволяющее отличать этот объект от всех других объектов.

Объекты могут находиться в определенных отношениях друг к другу. Эти отношения могут быть иерархическими. Основные иерархические отношения - это отношения использования и включения.

Отношение использования реализуется посылкой сообщений от объекта А к объекту В. При этом объект А может выступать в роли:

- активного или воздействующего объекта, когда он воздействует на другие объекты, но сам воздействию не подвергается;
- пассивного или исполняющего, когда объект подвергается воздействию, но сам на другие объекты не воздействует;
- посредника, если объект и воздействует и сам подвергается воздействию.

Отношение включения имеет место, когда составной объект содержит другие объекты.

Структура и поведение сходных объектов определяют *класс* объектов.

Между классами также могут быть установлены отношения:

- отношение разновидности (кошка - вид определенного биологического семейства или кошка - домашнее животное);
- включения или составной части (лапа - часть кошки);
- ассоциативности, когда между классами есть чисто смысловая связь (кошки и собаки - домашние животные).

Объектно-ориентированный подход к проектированию программных изделий предполагает:

- проведение объектно-ориентированного анализа предметной области;
- объектно-ориентированное проектирование;

- разработку программного изделия с использованием объектно-ориентированного языка программирования.

Вопросы для самопроверки

1. Опишите назначение системного проектирования.
2. Охарактеризуйте принцип практической полезности системного проектирования.
3. Охарактеризуйте принцип единства составных частей системного проектирования.
4. Охарактеризуйте принцип изменяемости во времени системного проектирования.
5. Перечислите достоинства и недостатками нисходящего и восходящего проектирования.
6. Объясните сущность понятия «объект».
7. Что такое метод?
8. Назовите типы методов (операций) объектно-ориентированного проектирования.
9. В каких отношениях могут находиться объекты?
10. Что определяет класс объектов?
11. В каких отношениях могут находиться классы объектов?
12. В чем заключается объектно-ориентированный подход к проектированию программных изделий?

Самостоятельная работа

Аналитический обзор литературы по теме «Методы проектирования программных продуктов» (2 ч.)

Тема 1.4. Проектирование интерфейса пользователя

1.4.1. Интерфейс пользователя программного продукта

Пользовательский интерфейс — это совокупность информационной модели проблемной области, средств и способов взаимодействия пользователя с информационной моделью, а также компонентов, обеспечивающих формирование информационной модели в процессе работы программной системы.

Под информационной моделью понимается условное представление проблемной области, формируемое с помощью компьютерных (визуальных и звуковых) объектов, отражающих состав и взаимодействие реальных компонентов проблемной области.

Средства и способы взаимодействия с информационной моделью определяются составом аппаратного и программного обеспечения, имеющегося в распоряжении пользователя, и от характера решаемой задачи. Например, для пользователя, который хочет переписать файл с диска на жесткий диск, такими средствами являются устройства ввода-вывода (клавиатура, мышь и экран монитора) и два дисковода с дисками. А вот для пользователя, который пытается установить собственные значения параметров BIOS, перечень доступных средств существенно шире. Совсем другое дело — та часть интерфейса, которая относится к программным средствам.

Во-первых, для программы значительно сложнее сформулировать **объективные** требования по составу и компоновке органов управления; зачастую не только пользователи, но и сами разработчики не могут объяснить, почему программа имеет именно такие «рычаги» и «педали». Во-вторых, их перечень значительно шире, а состав изменяется во много раз **динамичнее**, чем состав аппаратных средств компьютера.

Распространенной является ситуация, когда программы, равноценные по назначению и функциональным возможностям, оказываются совсем разными по организации взаимодействия с пользователем. При этом совсем не обязательно интерфейс какой-то из программ будет хуже, он просто будет другим**.** И если по какой-то причине знакомая программа окажется недоступной, освоение новой придется начинать практически с нуля.

Значительно большие потери может понести пользователь, которому предстоит либо выбрать одну из незнакомых программ, либо перейти на новую версию уже используемой программы.

В первом случае выбор может быть сделан в пользу менее функциональной и менее надежной программы, но обладающей более привлекательным (с субъективной точки зрения) интерфейсом.

Во втором же случае незнакомый интерфейс повой версии может оказаться психологическим барьером, не преодолев который пользователь так и не сможет воспользоваться преимуществами новой версии. Яркий пример такой ситуации — неожиданно медленный (для Microsoft) переход пользователей от Windows3.* к Windows 9*.

Таким образом, эффективность работы пользователя определяется не только функциональными возможностями имеющихся в его распоряжении аппаратных и программных средств, но и доступностью для пользователя этих возможностей. В свою очередь, полнота использования потенциальных возможностей имеющихся ресурсов зависит от качества пользовательского интерфейса.

Если надо напечатать с помощью компьютера пригласительные билеты на юбилей, придется воспользоваться текстовым или графическим редактором. Все редакторы «общего пользования» позволяют выполнять примерно один и тот же перечень операций, но весь вопрос в том, как они это делают и каким представлял себе разработчик потенциального пользователя своего продукта. Редактор с неудачным интерфейсом может потребовать от пользователя знакомства с совершенно новыми для него терминами, такими как «лигатура» и «кегль», а после каждого неудачного действия заставит возобновить работу с самого начала. Работа с таким редактором может закончиться тем, что юбиляр предпочтет купить пригласительные билеты в магазине.

Главный вывод заключается в том, что *качество пользовательского интерфейса* является самостоятельной характеристикой программного продукта, сопоставимо по значимости с такими его показателями, как надежность и эффективность использования вычислительных ресурсов.

Важное следствие: разработчик приложения должен знать, что такое хороший интерфейс, и как его построить.

Диалоговый режим

Большинство программных продуктов, особенно прикладного характера, ориентированных на конечного пользователя, работают в диалоговом режиме взаимодействия с пользователем таким образом, что ведется обмен сообщениями, влияющими на обработку данных.

В диалоговом режиме под воздействием пользователя осуществляются запуск функций (методов) обработки, изменение свойств объектов, производится настройка параметров выдачи информации на печать и т.п.

Системы, поддерживающие диалоговые процессы, классифицируются на:

- системы с жестким сценарием диалога - стандартизированное представление информации обмена;
- дескрипторные системы - формат ключевых слов сообщений;
- тезаурусные системы - семантическая сеть дескрипторов, образующих словарь системы (аналог - гипертекстовые системы);
- системы с языком деловой прозы - представление сообщений на языке, естественном для профессионального пользования.

Наиболее просты для реализации и распространены диалоговые системы с жестким сценарием диалога, которые предоставлены в виде:

- меню- диалог инициируется программой; пользователю предлагается выбор альтернативы функций обработки из фиксированного перечня; предоставляемое меню может быть иерархическим и содержать вложенные подменю следующего уровня;
- действия запрос-ответ - фиксирован перечень возможных значений, выбираемых из списка, или ответы типа Да/Нет;
- запрос по формату - с помощью ключевых слов, фраз или путем заполнения экранной формы с регламентированным по составу и структуре набором реквизитов осуществляется подготовка сообщений.

Диалоговый процесс управляется согласно созданному сценарию, для которого определяются:

- точки (момент, условие) начала диалога;
- инициатор диалога - человек или программный продукт;
- параметры и содержание диалога - сообщения, состав и структура меню, экранные формы и т.п.;
- реакция программного продукта на завершение диалога.

Описание сценария диалога выполняют:

- блок-схема, в которой предусмотрены блоки выдачи сообщений и обработки полученных ответов;

- ориентированный граф, вершины которого - сообщения и выполняемые действия, дуги - связь сообщений; словесное описание;

- специализированные объектно-ориентированные языки построения сценариев.

Для создания диалоговых процессов и интерфейса конечного пользователя наиболее подходят объектно-ориентированные инструментальные средства разработки программ.

В составе инструментальных средств СУБД содержатся построители меню, с помощью которых создается ориентированная на конечного пользователя совокупность режимом и команд в виде главного меню и вложенных подменю. Конструктор экранных форм СУБД используется для разработки форматов экранного ввода и редактирования данных базы данных и входной информации, управляющей работой программного продукта .

В ряде СУБД и электронных таблиц, текстовых редакторов существуют различные типы диалоговых окон содержащих разнообразные объекты управления:

- тексты сообщения;
- поля ввода информации пользователя;
- списки возможных альтернатив для выбора;
- кнопки и т.п.

В среде электронных таблиц и текстовых редакторов имеются возможности настройки главных меню (удаление ненужных, добавление новых режимов и команд), создания системы подсказок с помощью встроенных средств и языков программирования.

Графический интерфейс пользователя

Графический интерфейс пользователя (Graphics User Interface - GUI) - ГИЛ является обязательным компонентом большинства современных программных продуктов, ориентированных на работу конечного пользователя. К графическому интерфейсу пользователя предъявляются высокие требования как с чисто инженерной, так и с художественной стороны разработки, при его разработке ориентируются на возможности человека.

Наиболее часто графический интерфейс реализуется в интерактивном режиме работы пользователя для программных продуктов, функционирующих в среде Windows, и строится в виде системы спускающихся меню с использованием в качестве средства манипуляции мыши и клавиатуры. Работа пользователя осуществляется с экранными формами, содержащими объекты управления, панели инструментов с пиктограммами режимов и команд обработки.

Стандартный графический интерфейс пользователя должен отвечать ряду требований:

- поддерживать информационную технологию работы пользователя с программным продуктом - содержать привычные и понятные пользователю пункты меню, соответствующие функциям обработки, расположенные в естественной последовательности использования;

- ориентироваться на конечного пользователя, который общается с программой на внешнем уровне взаимодействия;

- удовлетворять правилу "шести" - в одну линейку меню включать не более 6 понятий, каждое из которых содержит не более 6 опций;

- графические объекты сохраняют свое стандартизованное назначение и по возможности местоположение на экране.

1.4.2. Основные принципы разработки пользовательского интерфейса

Создание качественного интерфейса требует значительно большего, чем просто соблюдение некоторых инструкций. Оно предполагает реализацию принципа «интересы пользователя превыше всего» и соответствующую методологию разработки всего программного продукта. В англоязычной литературе для описания такого подхода используется термин User-centered Design -**UCD** («Разработка, ориентированная на пользователя»). Эта технология, кроме всего прочего, предполагает как можно более раннее Проектирование интерфейса с последующим его развитием в процессе разработки самого программного продукта.

Основное достоинство хорошего интерфейса пользователя заключается в том, что *пользователь всегда чувствует, что он управляет программным обеспечением, а не программное обеспечение управляет им.*

Для создания у пользователя такого ощущения «внутренней свободы» интерфейс должен обладать целым рядом свойств, рассмотренных ниже.

Естественность интерфейса

Естественный интерфейс — такой, который не вынуждает пользователя существенно изменять привычные для него способы решения задачи. Это, в частности, означает, что сообщения и результаты, выдаваемые приложением, не должны требовать дополнительных пояснений. Целесообразно также сохранить систему обозначений и терминологию, используемые в данной предметной области.

Использование знакомых пользователю понятий и образов (метафор) обеспечивает интуитивно понятный интерфейс при выполнении его заданий. Вместе с тем, используя метафоры, не надо ограничивать их машинную реализацию полной анналогией с одноименными объектами реального мира. Например, папка на Рабочем столе Windows может использоваться для хранения целого ряда других объектов. Метафоры являются своего рода «мостиком», связывающим образы реального мира с теми действиями и объектами, которыми приходится манипулировать пользователю при его работе на компьютере; они обеспечивают «узнавание», а не «вспоминание». Пользователи запоминают действие, связанное со знакомым объектом, более легко, чем они запомнили бы имя команды, связанной с этим действием.

Согласованность интерфейса

Согласованность позволяет пользователям переносить имеющиеся знания на новые задания, осваивать новые аспекты быстрее, и благодаря этому фокусировать внимание на решаемой задаче, а не тратить время на уяснение различий в использовании тех или иных элементов управления, команд и т.д. Обеспечивая преемственность полученных ранее знаний и навыков, согласованность делает интерфейс узнаваемым и предсказуемым.

Согласованность важна для всех аспектов интерфейса, включая имена команд, визуальное представление информации и поведение интерактивных элементов. Для реализации свойства согласованности в создаваемом программном обеспечении, необходимо учитывать его различные аспекты.

Согласованность в пределах продукта

Одна и та же команда должна выполнять одни и те же функции, где бы она ни встретилась, причем одним и тем же образом. Например, если в одном диалоговом окне команда *Копировать* означает немедленное выполнение соответствующих действий, то в другом окне она не должна требовать от пользователя дополнительно указать расположение копируемой информации. Надо использовать одну и ту же команду, чтобы выполнить функции, которые кажутся подобными пользователю.

Согласованность в пределах рабочей среды

Поддерживая согласованность с интерфейсом, предоставляемым операционной системой, приложение может «опираться» на те знания и навыки пользователя, которые он получил ранее при работе с другими приложениями.

Согласованность в использовании метафор

Если поведение некоторого программного объекта выходит за рамки того, что обычно подразумевается под соответствующей ему метафорой, у пользователя могут возникнуть трудности при работе с таким объектом. Например, если для программного объекта *Корзина* определить операцию *Запуск,* то для уяснения ее смысла пользователю, скорее всего, потребуется посторонняя помощь.

Дружественность интерфейса (принцип «прощения» пользователя)

Пользователи обычно изучают особенности работы с новым программным продуктом методом проб и ошибок. Эффективный интерфейс должен принимать во внимание такой подход. На каждом этапе работы он должен разрешать только соответствующий набор действий и предупреждать пользователей о тех ситуациях, где они могут повредить системе или данным; еще лучше, если у пользователя существует возможность отменить или исправить выполненные действия.

Даже при наличии хорошо спроектированного интерфейса пользователи могут делать те или иные ошибки. Эти ошибки могут быть как «физического» типа (случайный выбор неправильной команды или данных) так и «логического» (принятие неправильного решения на выбор команды или данных). Эффективный интерфейс должен позволять предотвращать ситуации, которые, вероятно закончатся ошибками. Он также должен уметь **адаптироваться** к потенциальным ошибкам пользователя и облегчать ему процесс устранения последствий таких ошибок.

Принцип «обратной связи»

Всегда обеспечивайте обратную связь для действий пользователя. Каждое действие пользователя должно получать визуальное, а иногда и звуковое подтверждение того, что программное обеспечение восприняло введенную команду; при этом вид реакции, по возможности, должен учитывать природу выполненного действия.

Обратная связь эффективна в том случае, если она реализуется своевременно, т.е. как можно ближе к точке последнего взаимодействия пользователя с системой. Когда компьютер обрабатывает поступившее задание, полезно предоставить пользователю информацию относительно состояния процесса, а также возможность прервать этот процесс в случае необходимости. Ничто так не смущает не очень опытного пользователя, как заблокированный экран, который никак не реагирует на его действия. Типичный пользователь способен вытерпеть только несколько секунд ожидания ответной реакции от своего электронного «собеседника».

Простота интерфейса

Интерфейс должен быть простым. При этом имеется в виду не упрощенчество, а обеспечение **легкости** в его изучении и в использовании. Кроме того, он должен предоставлять **доступ** ко всему перечню функциональных возможностей, предусмотренных данным приложением. Реализация доступа к широким функциональным возможностям и обеспечение простоты работы противоречат друг другу. Разработка эффективного интерфейса призвана сбалансировать эти цели.

Один из возможных путей поддержания простоты — представление на экране информации, минимально необходимой для выполнения пользователем очередного шага задания. Многословные командные имена или сообщения, непродуманные или избыточные фразы затрудняют пользователю извлечение существенной информации.

Другой путь к созданию простого, но эффективного интерфейса — размещение и представление элементов на экране с учетом их смыслового значения и логической взаимосвязи. Это позволяет использовать в процессе работы ассоциативное мышление пользователя.

Можно помочь пользователям управлять сложностью отображаемой информации, используя *последовательное раскрытие* (диалоговых окон, разделов меню и т.д.). Последовательное раскрытие предполагает такую организацию информации, при которой в каждый момент времени на экране находится только та ее часть, которая необходима для выполнения очередного шага. Сокращая объем информации, представленной пользователю, уменьшают объем информации, подлежащей обработке. Примером такой организации является иерархическое (каскадное) меню, каждый уровень которого отображает только те пункты, которые соответствуют одному, выбранному пользователем, пункту более высокого уровня.

Гибкость интерфейса

Гибкость интерфейса — это его способность учитывать уровень подготовки и производительность труда пользователя. Гибкость предполагает возможность изменения структуры диалога и/или входных данных. Концепция гибкого ***(адаптивного)*** интерфейса в настоящее время является одной из основных областей исследования взаимодействия человека и ЭВМ. Основная проблема состоит не в том, как организовать изменения в диалоге, а в том, какие признаки нужно использовать для определения необходимости внесения изменений и их сути.

Эстетическая привлекательность

Проектирование визуальных компонентов является важнейшей составной частью разработки программного интерфейса. Корректное визуальное представление используемых объектов обеспечивает передачу весьма важной дополнительной информации о поведении и взаимодействии различных объектов. В то же время следует помнить, что каждый визуальный элемент, который появляется на экране, потенциально требует внимания пользователя, которое, как известно, не безгранично. Следует формировать на экране среду, которая не только содействовала бы пониманию пользователем представленной информации, но и позволяла бы сосредоточиться на наиболее важных ее аспектах.

Наибольших успехов в проектировании пользовательского интерфейса, обладающего перечисленными свойствами, к настоящему времени добились разработчики компьютерных игр.

Качество интерфейса сложно оценить количественными характеристиками, однако более или менее объективную его оценку можно получить на основе приведенных ниже частных показателей.

1. **Время**, необходимое определенному пользователю для достижения заданного уровня знаний и навыков по работе с приложением (например, непрофессиональный пользователь должен освоить команды работы с файлами не более чем за 4 часа).

2. **Сохранение** полученных рабочих **навыков** по истечении некоторого времени (например, после недельного перерыва пользователь должен выполнить определенную последовательность операций за заданное время).

3. **Скорость решения задачи** с помощью данного приложения; при этом должно оцениваться не быстродействие системы и не скорость ввода данных с клавиатуры, а время, необходимое для достижения цели решаемой задачи. Исходя из этого, критерий оценки по данному показателю может быть сформулирован, например, так: пользователь должен обработать за час не менее 20 документов с ошибкой не более 1%.

4. **Субъективная удовлетворенность** пользователя при работе с системой (которая количественно может быть выражена в процентах или оценкой по n-бальной шкале).

Основные правила, которые обеспечивают эффективность пользовательского интерфейса.

• Интерфейс пользователя необходимо проектировать и разрабатывать как отдельный компонент создаваемого приложения.

• Необходимо учитывать возможности и особенности аппаратно-программных средств, на базе которых реализуется интерфейс.

• Целесообразно учитывать особенности и традиции той предметной области, к которой относится создаваемое приложение.

• Процесс разработки интерфейса должен носить итерационный характер, его обязательным элементом должно быть согласование полученных результатов с потенциальным пользователем.

• Средства и методы реализации интерфейса должны обеспечивать возможность его адаптации к потребностям и характеристикам пользователя.

Вопросы для самопроверки

1. Сформулируйте понятие пользовательского интерфейса.
2. Охарактеризуйте значение пользовательского интерфейса для эффективности работы пользователя.
3. Приведите классификацию систем, поддерживающих диалоговые процессы.
4. В каком виде могут быть представлены диалоговые системы с жестким сценарием диалога?
5. Каким требованиям должен отвечать стандартный графический интерфейс пользователя?
6. Что означает согласованность интерфейса?
7. В чем заключается принцип дружественности интерфейса?
8. Что означает гибкость интерфейса?
9. На каких принципах основано создание простого, но эффективного интерфейса?
10. На основе каких показателей можно оценить качество интерфейса?

Самостоятельная работа

1. Составление кроссворда по теме «Методы проектирования программных продуктов» (2 ч.)
2. Реферирование темы «Этапы проектирования пользовательского интерфейса» (3ч.)

Тема 1.5. Методы разработки программных модулей

1.5.1. Сущность модульного программирования

Приступая к разработке программы, следует иметь в виду, что она, как правило, является большой системой, поэтому необходимо принять меры для ее упрощения. Для этого программу разрабатывают по частям, которые называются *программными модулями.* Такой метод создания программ называют модульным программированием.

Модульное программирование – это процесс разделения программы на логические части, называемые модулями, и последовательное программирование каждой части. Когда большая единая задача делиться на подзадачи, то значительно проще прочесть и понять программу.

Оптимальный размер модуля 60 строк.

Модульное программирование основано на понятии модуля — программы или функционально завершенного фрагмента программы.

Модуль характеризуют:

– один вход и один выход. На входе программный модуль получает определенный набор исходных данных, выполняет их обработку и возвращает один набор выходных данных;

– функциональная завершенность. Модуль выполняет набор определенных операций для реализации каждой отдельной функции, достаточных для завершения начатой обработки данных;

– логическая независимость. Результат работы данного фрагмента программы не зависит от работы других модулей;

– слабые информационные связи с другими программными модулями. Обмен информацией между отдельными модулями должен быть минимален;

– размер и сложность программного элемента в разумных рамках.

Таким образом, модули содержат описание исходных данных, операции обработки данных и структуры взаимосвязи с другими модулями.

Программный модуль является самостоятельным программным продуктом. Это означает, что каждый программный модуль разрабатывается, компилируется и отлаживается отдельно от других модулей программы. Более того, каждый разработанный программный модуль может включаться в состав разных программных систем при условии выполнения требований, предъявляемых к его использованию в документации к этому модулю. Таким образом, программный модуль может рассматриваться и как средство упрощения сложных программ, и как средство накопления и многократного использования программистских знаний.

Следует стремиться к независимости между модулями или программами. Каждый модуль должен иметь свое назначение, отличающееся от назначения других модулей. Это должен быть замкнутый блок, вход и выход которого четко определены. Стремление к независимости хорошо тем, что менее вероятно, что изменения в одной подпрограмме влияет на оставшуюся часть программы. Воздействие изменения в одном модуле на другую часть программы называется *волновым эффектом*. Этот эффект можно уменьшить сведя к минимуму связь между модулями. Простой путь уменьшения волнового эффекта – избегать использование глобальных перемен и делать модуль небольшим. Минимизация взаимосвязи между модулями – это модульное сцепление, которое происходит за счет усиления связей между элементами одного модуля (модульная прочность). Таким образом, тесно связанные элементы надо стремиться поместить в один модуль. Использование модулей приводит к уменьшению сложностей, факторы сложности при этом включают три составляющие:

1) функциональная сложность – обусловлено тем, что один модуль выполняет слишком много функций;

2) распределенная сложность – это сложность идентификации общей функции распределенной между несколькими модулями за счет чего утрачивается возможность уменьшения сложности всей программы при модульном программировании;

3) сложность связи – определяется сложностью взаимодействия модулей, при использовании общих данных.

Модульная структура программы представляет собой древовидную структуру, в узлах которой размещаются программные модули, а направленные дуги показывают статическую подчиненность модулей. Если в тексте модуля имеется ссылка на другой модуль, то их на структурной схеме соединяет дуга, которая исходит из первого и входит во второй модуль. Другими словами, каждый модуль может обращаться к подчиненным ему модулям. При этом модульная структура программной системы, кроме структурной схемы, должна включать в себя еще и совокупность спецификаций модулей, образующих эту систему.

Функция верхнего уровня обеспечивается главным модулем; он управляет выполнением нижестоящих функций, которым соответствуют подчиненные модули.

При определении набора модулей, реализующих функции конкретного алгоритма, необходимо учитывать следующее:

1. модуль вызывается на выполнение вышестоящим по иерархии модулем и, закончив работу, возвращает ему управление;
2. принятие основных решений в алгоритме выносится на максимально высокий по иерархии уровень;
3. если в разных местах алгоритма используется одна и та же функция, то она оформляется в отдельный модуль, который будет вызываться по мере необходимости.

Состав, назначение и характер использования программных модулей в значительной степени определяются инструментальными средствами.

Например, при разработке СУБД используются следующие программные модули:

1. экранные формы ввода и/или редактирования информации базы данных;
2. отчеты;
3. макросы;
4. стандартные средства для обработки информации;
5. меню для выбора функции обработки и др.

Методы разработки при модульном программировании

Спецификация программного модуля состоит из функциональной спецификации модуля, описывающей семантику функций, выполняемых этим модулем по каждому из его входов, и синтаксической спецификации его входов, позволяющей построить на используемом языке программирования синтаксичеки правильное обращение к нему. Функциональная спецификация модуля определяется теми же принципами, что и функциональная спецификация программной системы.

Существуют разные методы разработки модульной структуры программы, в зависимости от которых определяется порядок программирования и отладки модулей, указанных в этой структуре. Обычно в литературе обсуждаются два метода метод восходящей разработки и метод нисходящей разработки.

Метод восходящей разработки

Сначала строится древовидная модульная структура программы. Затем поочередно проектируются и разрабатываются модули программы, начиная с модулей самого нижнего уровня, затем предыдущего уровня и т. д. То есть модули реализуются в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же восходящем порядке. Достоинство метода заключается в том, что каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули. Недостатки метода восходящей разработки заключаются в следующем:

- на нижних уровнях модульной структуры спецификации могут быть еще определены не полностью, что может привести к полной переработке этих модулей после уточнения спецификаций на верхнем уровне;
- для восходящего тестирования всех модулей, кроме головного, который является модулем самого верхнего уровня, приходится создавать вызывающие программы, что приводит к созданию большого количества отладочного материала, но не гарантирует, что результаты тестирования верны;
- головной модуль проектируется и реализуется в последнюю очередь, что не дает продемонстрировать его заказчику для уточнения спецификаций.

Метод нисходящей разработки

Как и в предыдущем методе, сначала строится модульная структура программы в виде дерева. Затем проектируются и реализуются модули программы, начиная с модуля самого верхнего уровня — головного, далее разрабатываются модули уровнем ниже и т. д. При этом переход к программированию какого-либо модуля осуществляется только в том случае, если уже запрограммирован модуль, который к нему обращается. Затем производится их поочередное тестирование и отладка в таком же нисходящем порядке. Первым тестируется головной модуль программы, который представляет всю тестируемую программу, при этом все модули, к которым может обращаться головной, заменяются их имитаторами (так называемыми «заглушками»). Каждый *ими-*

татор модуля является простым программным фрагментом, реализующим сам факт обращения к данному модулю с необходимой для правильной работы программы обработкой значений его входных параметров и с выдачей, если это необходимо, подходящего результата. Далее производится тестирование следующих по уровню модулей. Для этого имитатор выбранного для тестирования модуля заменяется самим модулем, и добавляются имитаторы модулей, к которым может обращаться тестируемый модуль.

Недостатком нисходящего подхода к программированию является необходимость абстрагироваться от реальных возможностей выбранного языка программирования и придумывать абстрактные операции, которые позже будут реализованы с помощью модулей. Однако способность к таким абстракциям является необходимым условием разработки больших программных средств.

1.5.2. Основные принципы технологии структурного программирования

Один из методов улучшения программы является структурное программирование (СП). Оно предназначено для организации проектирования программ и процесса кодирования. Таким образом, чтобы предотвратить большинство логических ошибок и обнаружить те, которые допущены.

Перечислим некоторые достоинства структурного программирования:

1. Значительно сокращает число вариантов построения программы по одной и той же спецификации, что значительно снижает сложность программы.
2. Логически связанные операторы находятся визуально ближе, а слабо связанные дальше, что позволяет обходиться без блок-схем.
3. Сильно упрощается процесс тестирования и отладки структурированных программ.
4. Обеспечивает более высокую производительность работы за счет того, что действие каждой управляющей структуры хорошо известно и нет необходимости его обдумывать.
5. Обеспечивает ясность и читаемость программ;
6. Обеспечивает более высокую эффективность за счет глобальной оптимизации программы.

Структурное кодирование – это метод написания хорошо структурированных программ, который позволяет получать программы более удобные для тестирования, модификации и использования. Программы произвольных размеров и сложности могут быть написаны на основе ограниченного множества базисных структур. Этот принцип положен в основу проектирования схем, где любая логическая структура может быть создана из элементарных структур:

- *структура последовательности* – это формализация того, что операторы программы выполняются в порядке их появления в программе, пока что-то не изменит их последовательность.
- *структура выбора* – это выбор одного из двух действий исходя из выполненного некоторого условия.
- *структура повторения* – используется для повторного выполнения группы команд до тех пор, пока не выполниться некоторое условие. Получение правильной программы путем замены операторов программы на управляющие логические структуры называется *вложением структур*.

В основу структурирования положены следующие простые правила:

1.программа должна составляться мелкими шагами, размер шага определяется количеством решений, применяемых программистом на этом шаге.

2.сложная задача разбивается на простые легко воспринимаемые части, каждая из которых имеет только один вход и один выход.

3.логика программы должна опираться на минимальное число простых базовых управляющих структур.

Фундаментом структурного программирования является теорема о структуризации. Эта теорема устанавливает, что как бы не была сложна задача, блок-схема программы может быть представлена с использованием весьма ограниченного числа элементарных управляющих структур. Эти элементарные структуры могут соединяться между собой, образуя более сложные структуры, при этом они могут представлять собой довольно

сложные блок-схемы с одним входом и с одним выходом.

Чтобы обойтись без произвольных передач управления в программе, там, где это возможно, лучше не использовать операторы GOTO, RETURN. Характерной особенностью структурированной программы является не столько отсутствие этих операторов, а сколько наличие жесткой структуры ее организации. Если текст программы будет занимать несколько десятков страниц, то восприятие такой программы будет затрудненно, поэтому рекомендуется вести структурирование текста программы (например, программа состоит из 80 страниц исходного текста, необходимо этот исходный текст заменить на текст, в котором отражаются наиболее важные, в функциональном смысле, фразы в виде сегментов). Если расписать весь программный комплекс крупными сегментами, то описание всего комплекса может занять всего одну страницу, а это наглядно и удобно.

Различают три вида вычислительного процесса, реализуемого программами: линейный, разветвленный и циклический.

Линейная структура процесса вычислений предполагает, что для получения результата необходимо выполнить некоторые операции в определенной последовательности.

Разветвленная структура процесса вычислений предполагает, что конкретная последовательность операций зависит от значений одной или нескольких переменных.

Циклическая структура процесса вычислений предполагает, что для получения результата некоторые действия необходимо выполнить несколько раз.

Для реализации указанных вычислительных процессов в программах используют соответствующие управляющие операторы. Первые процедурные языки программирования высокого уровня, такие, как FORTRAN, понятием «тип вычислительного процесса» не оперировали. Для изменения линейной последовательности операторов в них, как в языках низкого уровня, использовались команды условной (при выполнении некоторого условия) и безусловной передач управления. Потому и программы, написанные на этих языках, имели запутанную структуру, присущую в настоящее время только низкоуровневым (машинным) языкам.

Именно для изображения схем алгоритмов таких программ в свое время был разработан ГОСТ 19.701-90, согласно которому каждой группе действий ставится в соответствие специальный блок (табл. 6). Хотя этот стандарт предусматривает блоки для обозначения циклов, он не запрещает и произвольной передачи управления, т. е. допускает использование команд условной и безусловной передачи управления при реализации алгоритма.

В качестве примера, демонстрирующего особенности использования команд передачи управления для организации требуемого типа вычислительного процесса, рассмотрим программу на языке Ассемблера.

Пример 1 Реализовать на языке Ассемблера подпрограмму поиска минимального элемента массива

На рис. 11 приведен пример неудачной реализации этой подпрограммы. Стрелками показаны передачи управления. Даже с комментариями и стрелками понять хорошо известный алгоритм достаточно сложно.

```
MinEl   proc   near
        lea    BX, a        ; загрузили адрес массива
        mov    CX, n        ; загрузили счетчик - количество элементов
less:   mov    AX, 0(BX)    ; приняли элемент в качестве максимума
cycle:  sub    CX           ; уменьшили счетчик
        jcxz   exit         ; если все числа, то выход
        add    BX,2         ; адрес следующего элемента
        cmp    AX, 0(BX)    ; сравниваем текущий элемент
        jl     less         ; если меньше, то меняем максимум
        jmp    cycle        ; иначе переходим к следующему элементу
exit:   ret
MinEl   endp
```

Рис. 11. Подпрограмма поиска минимального элемента (неудачная реализация на Ассемблере)

Таблица 6

Название блока	Обозначение	Назначение блока
Терминатор	Действие	Начало, завершение программы или подпрограммы
Процесс	Действие	Обработка данных (вычисления, пересылки и т. п.)
Данные	Данные	Операции ввода-вывода
Решение	Условие	Ветвления, выбор, итерационные и поисковые циклы
Подготовка	Действия	Счетные циклы
Граница цикла	Начало Конец	Любые циклы
Предопределенный процесс	Имя	Вызов процедур
Соединитель	Имя	Маркировка разрывов линий
Комментарий	--- Комментарий	Пояснения к операциям

После того, как в 60-х годах XX в. было доказано, что любой сколь угодно сложный алгоритм можно представить с использованием трех основных управляющих конструкций, в языках программирования высокого уровня появились управляющие операторы для реализации соответствующих конструкций. Эти три конструкции принято считать базовыми. К ним относят конструкции:

- *следование* - обозначает последовательное выполнение действий (рис. 12, а);
- *ветвление* - соответствует выбору одного из двух вариантов действий (рис. 12, б);

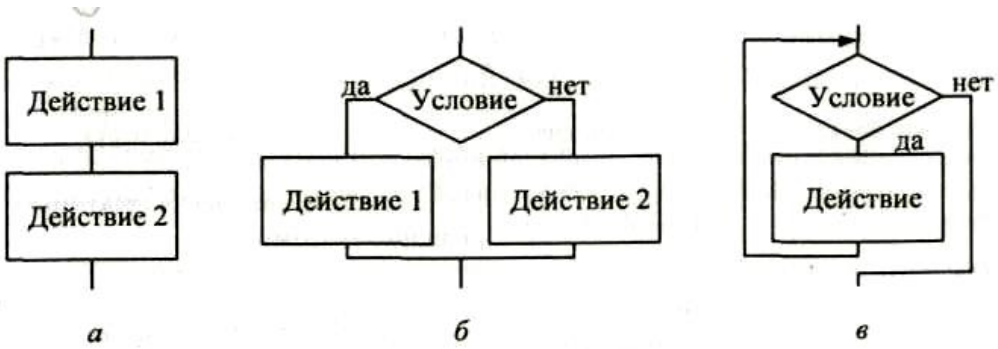


Рис. 12. – Базовые алгоритмические структуры:

• *цикл-пока* - определяет повторение действий, пока не будет нарушено некоторое условие, выполнение которого проверяется в начале цикла (рис. 12, в).

Помимо базовых, процедурные языки программирования высокого уровня обычно используют еще три конструкции, которые можно составить из базовых:

• *выбор* - обозначает выбор одного варианта из нескольких в зависимости от значения некоторой величины (рис. 13, а);

• *цикл-до* - обозначает повторение некоторых действий до выполнения заданного условия, проверка которого осуществляется после выполнения действий в цикле (рис. 13, б);

• *цикл с заданным числом повторений* (счетный цикл) - обозначает повторение некоторых действий указанное количество раз (рис. 13, в).

Любая из дополнительных конструкций легко реализуется через базовые. Перечисленные шесть конструкций были положены в основу структурного программирования.

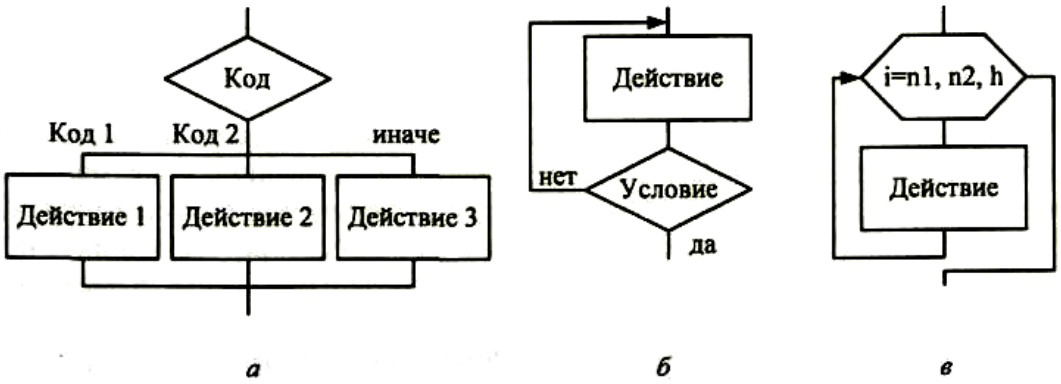


Рис. 13.– Дополнительные структуры алгоритмов
а- выбор; б- цикл-до; в- цикл с заданным числом повторений

Примечание. Слово «структурное» в данном названии подчеркивает тот факт, что при программировании использованы только перечисленные конструкции (структуры). Отсюда и понятие «программирование без go to».

Программы, написанные с использованием только структурных операторов передачи управления, называют *структурными,* чтобы подчеркнуть их отличие от программ, при проектировании или реализации которых использовались низкоуровневые способы передачи управления.

Несмотря на то, что Ассемблер не предусматривает соответствующих конструкций, «структурно» можно программировать и на нем. Вернемся к примеру 1.

Пример 1 (вариант 2). Поскольку реализуемый цикл по типу «счетный» с количеством повторений n-1, используем соответствующую команду Ассемблера. Уберем и усложняющий понимание возврат на метку less, заменив его дубликатом команды сохранения текущего максимального элемента.

Полученный в результате «структурированный» вариант программы поиска максимального элемента массива приведен на рис. 14. Единственный возврат реализует циклический процесс, а передача управления на следующие команды - ветвление.

Представление алгоритма программы в виде схемы с точки зрения структурного программирования имеет два недостатка:

• предполагает слишком низкий уровень детализации, что часто скрывает суть сложных алгоритмов;

• позволяет использовать неструктурные способы передачи управления, причем часто на схеме алгоритма они выглядят проще, чем эквивалентные структурные.

```
MinEl   proc   near
        lea    BX, a        ; загрузили адрес массива
        mov    CX, n        ; загрузили счетчик - количество элементов
        mov    AX, 0(BX)    ; приняли элемент в качестве максимума
        sub    CX           ; уменьшили счетчик
        add    BX,2         ; занесли адрес следующего элемента
cycle:  cmp    AX, 0(BX)    ; сравниваем текущий элемент
        jge    next         ; если не меньше, то пропускаем
        mov    AX, 0(BX)    ; приняли элемент в качестве максимума
next:   add    BX,2         ; адрес следующего элемента
        loop   cycle        ; если не все, то повторяем
        ret
MinEl   endp
```

Рис.14. Подпрограмма поиска максимального элемента массива («структурный» вариант)

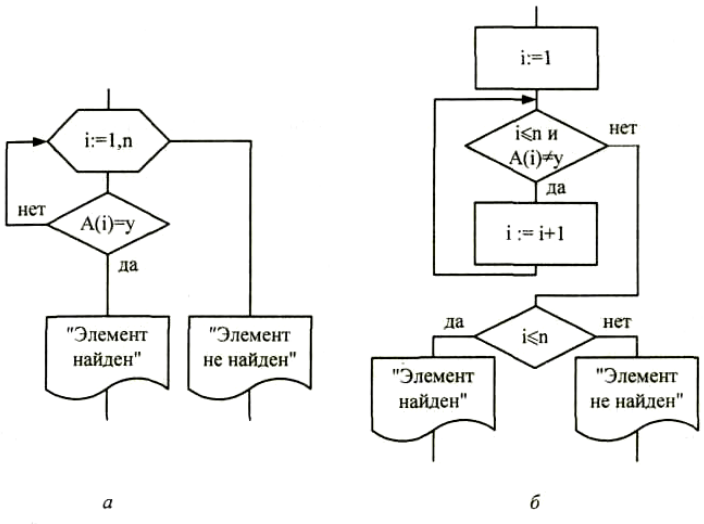


Рис. 15. Схема алгоритма реализации поискового цикла:

Классическим примером последнего является организация поискового цикла с использованием неструктурной передачи управления (рис. 15, а) и эквивалентный структурный вариант (рис. 15, б).

Кроме схем, для описания алгоритмов можно использовать псевдокоды, Flow-формы и диаграммы Насси-Шнейдермана. Все перечисленные нотации с одной стороны базируются на тех же основных структурах, что и структурное программирование, а с другой - допускают разные уровни детализации.

Псевдокоды. *Псевдокод* - формализованное текстовое описание алгоритма (текстовая нотация). В литературе были предложены несколько вариантов псевдокодов. Один из них приведен в табл. 7.

Описать с помощью псевдокодов неструктурный алгоритм невозможно. Использование псевдокодов изначально ориентирует проектировщика только на структурные способы передачи управления, а потому требует более тщательного анализа разрабатываемого алгоритма. В отличие от схем алгоритмов, псевдокоды не ограничивают степень детализации проектируемых операций. Они позволяют соизмерять степень детализации действия с уровнем абстракции, на котором это действие рассматривают, и хорошо согласуются с основным методом структурного программирования - методом пошаговой детализации.

Таблица 7

Структура	Псевдокод	Структура	Псевдокод
Следование	<действие 1> <действие 2>	Выбор	**Выбор** <код> <код 1>: <действие 1> <код 2>: <действие 2> ... **Все-выбор**
Ветвление	**Если** <условие> **то** <действие 1> **иначе** <действие 2> **Все-если**	Цикл с заданным количеством повторений	**Для** <индекс> = <n>,<k>,<h> <действие> **Все-цикл**
Цикл-пока	**Цикл-пока** <условие> <действие> **Все-цикл**	Цикл-до	**Выполнять** <действие> **До** <условие>

В качестве примера посмотрим, как будет выглядеть на псевдокоде описание алгоритма поискового цикла, представленного на рис. 14:

i:=l
Цикл-пока i < n *и A[i] ≠ y*
i:=i+l ***Все-цикл***
Если *i ≤n*
то *Вывести «Элемент найден»*
иначе *Вывести «Элемент не найден»* ***Все-если***

Flow-формы. *Flow-формы* представляют собой графическую нотацию описания структурных алгоритмов, которая иллюстрирует вложенность структур. Каждый символ Flow-формы соответствует управляющей структуре и изображается в виде прямоугольника. Для демонстрации вложенности структур символ Flow-формы может быть вписан в соответствующую область прямоугольника любого другого символа. В прямоугольниках символов содержится текст на естественном языке или в математической нотации. Размер прямоугольника определяется длиной вписанного в него текста и размерами вложенных прямоугольников. Символы Flow-форм, соответствующие основным и дополнительным управляющим конструкциям, приведены на рис. 15.

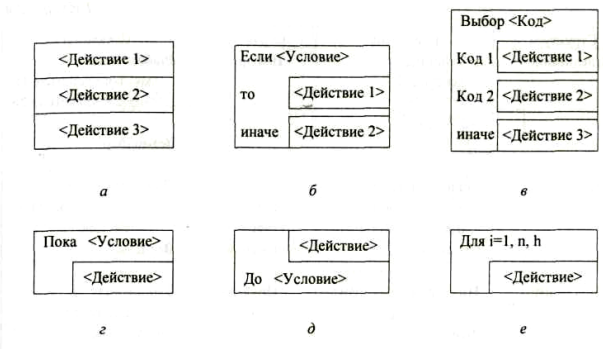


Рис. 15. Условные обозначения Flow-форм для основных конструкций:
а- следование; б- ветвление; в- выбор; г- цикл-до; е- счетный цикл

На рис. 16 представлено описание рассмотренного ранее поискового цикла с использованием Flow-формы. Хорошо видны вложенность и следование конструкций, изображенных прямоугольниками.

Диаграммы Насси-Шнейдермана. *Диаграммы Насси-Шнейдермана* являются развитием Flow-форм. Основное их отличие от Flow-форм заключается в том, что область обозначения условий и вариантов ветвления изображают в виде треугольников (рис. 17). Такое обозначение обеспечивает большую наглядность представления алгоритма.

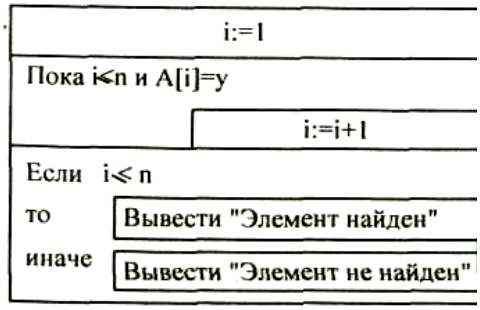


Рис. 16. Алгоритм поискового цикла

Также, как при использовании псевдокодов, описать неструктурный алгоритм, применяя Flow-формы или диаграммы Насси-Шнейдермана, невозможно (для неструктурных передач управления в этих нотациях просто отсутствуют условные обозначения). В то же время, являясь графическими, эти нотации лучше отображают вложенность конструкций, чем псевдокоды.

Общим недостатком Flow-форм и диаграмм Насси-Шнейдермана является сложность построения изображений символов, что усложняет практическое применение этих нотаций для описания больших алгоритмов.

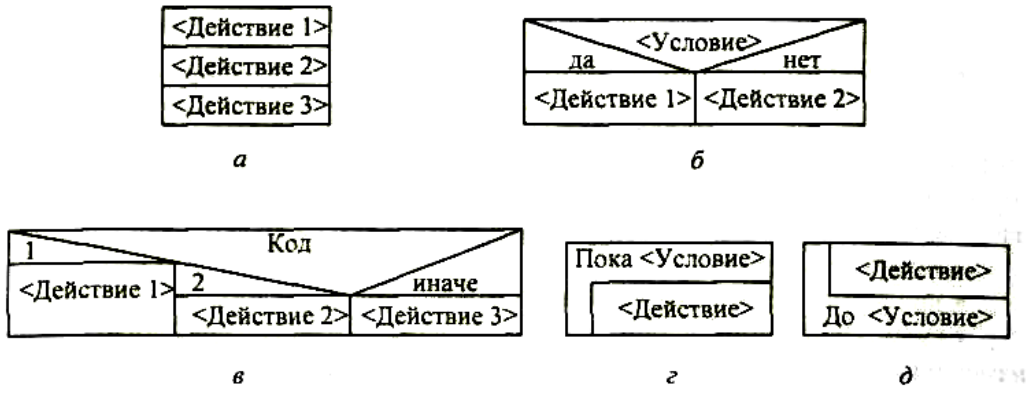


Рис. 17. Условные обозначения диаграмм Насси-Шнейдермана для основных кон-

Вопросы для самопроверки

1. Сформулируйте определение модульного программирования.
2. Приведите характеристики модуля.
3. Что называется волновым эффектом? Как можно его уменьшить?
4. Опишите модульную структуру программы.
5. Опишите метод восходящей разработки модульной структуры программы. Назовите достоинства и недостатки метода.
6. Опишите метод нисходящей разработки модульной структуры программы. Назовите достоинства и недостатки метода.
7. Перечислите достоинства структурного программирования.
8. Что такое структурное кодирование? Перечислите элементарные структуры, на основе которых строится любая логическая структура.
9. Сформулируйте правила, положенные в основу структурирования программного модуля.
10. Что называется псевдокодом? Приведите варианты описания псевдокодов.
11. Для чего предназначены Flow-формы и диаграммы Насси-Шнейдермана?
12. Назовите недостатки Flow-форм и диаграмм Насси-Шнейдермана.

Самостоятельная работа

1. Составить диаграмму Насси-Шнейдермана решения следующей задачи:

2. Дана прямоугольная матрица чисел A = $||a_{ij}||$, состоящая из *m* строк и *n* столбцов. Требуется найти строку *к,* сумма S_k элементов которой максимальна, найти в этой строке минимальный ее элемент a_{kl}, отметить номер соответствующего столбца *l* и вычислить сумму элементов этого столбца S_i (2 ч.)

Тема 1.6. Объектно - ориентированное программирование

1.6.1. Основные принципы технологии объектно-ориентированного программирования

Как говорилось в п. 1.3.2. объектно-ориентированное программирование (ООП) представляет собой способ программирования, который напоминает процесс человеческого мышления. ООП более структурировано, чем другие способы программирования и позволяет создавать модульные программы с представлением данных на определенном уровне абстракции. Основная цель ООП – это повышение эффективности разработки программ.

При структурном подходе программист обычно разделяет (структурирует) описываемый объект на составные части, стараясь описать свойства отдельных частей, не вдаваясь в подробности взаимодействия между ними.

Базовым в ООП является понятие объекта. Объект имеет определенные свойства. Состояние объекта задается значениями его признаков. Объект «знает», как решать определенные задачи, т.е. располагает методами решения. Программа, написанная с использованием ООП состоит из объектов, которые могут взаимодействовать между собой.

Концепция ООП заключается в том, что каждый объект является экземпляром некоторого класса объектов.

Все объекты с одинаковыми наборами атрибутов принадлежат к одному классу. Однако объединение объектов в классы определяется не набором атрибутов, а семантикой (смыслом). Так, например, объекты конюшня и лошадь могут иметь одинаковые атрибуты: цена и возраст. При этом они могут относиться к одному классу, если рассматриваются просто как товар, либо к разным классам, что более естественно.

Каждый класс имеет свои особенности поведения и характеристик, определяющих этот класс. Один класс отличается от других именем и, обычно, набором поддерживаемых интерфейсов. Интерфейсы, в свою очередь, представляют собою набор сообщений, которые можно посылать объекту.

ООП является наилучшим инструментом для построения иерархических деревьев или структур данных.

Характеристики ООП

1. Все является объектом.
2. Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщения - 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.
5. В классе задается 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 определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

Принципы ООП

1. *Абстракция данных* - это выделение существенных характеристик некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы относительно дальнейшего рассмотрения и анализа. Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности его поведения от деталей их реализации. Выбор правильного набора

абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Объекты представляют собою не полную информацию о реальных сущностях предметной области. Их модели, адекватны решаемой задаче, работать с ними намного удобнее, чем с низкоуровневым описанием всех возможных свойств и реакций объекта.

2. Инкапсуляция является важнейшим свойством объекта, на котором строится ООП. Инкапсуляция заключается в том, что объект скрывает в себе детали, которые несущественны для использования. В традиционном подходе к программированию с использованием глобальных переменных программист не был застрахован от ошибок, связанных с использованием процедур, не предназначенных для обработки данных, но связанных с этими переменами. «Жесткое» связывание данных и процедур их обработки в одном объекте позволяет избежать многие неприятности. Инкапсуляция и является средством организации доступа к данным *только через соответствующие методы*.

Инкапсуляция — это принцип, согласно которому любой класс должен рассматриваться как чёрный ящик — пользователь класса должен видеть и использовать только интерфейс (от английского interface — внешнее лицо, т. е. список декларируемых свойств и методов) класса и не вникать в его внутреннюю реализацию. Этот принцип (теоретически) позволяет минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов.

3. Наследование – это еще одно базовое понятие ООП. Наследование позволяет определить новые объекты, используя свойства прежних, дополняя или изменяя их. Объект – наследник получает все поля и методы «родителя», к которым он может добавить свои собственные поля и методы или заменить («перекрыть») их своими методами.

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

4. Полиморфизм заключается в том, что одно и то же имя может соответствовать различным действиям в зависимости от типа объекта. *(*Присваивание действию одного имени, которое затем совместно используется вниз и вверх по иерархии объектов, причем каждый объект иерархии выполняет это действие способом, характерным именно для него.*)*

Полиморфизмом называют явление, при котором классы-потомки могут изменять реализацию метода класса-предка, сохраняя его сигнатуру (таким образом, сохраняя неизменным интерфейс класса-предка). Это позволяет обрабатывать объекты классов-потомков как однотипные объекты, не смотря на то, что реализация методов у них может различаться.

1.6.2. Этапы объектно- ориентированного проектирования

Процесс разработки программного обеспечения с использованием объектно-ориентированного программирования включает четыре этапа:

1.Анализ.

2.Проектирование.

3.Эволюция.

4.Модификация.

Особенностью ООП является то, что объект или группа объектов могут разрабатываться отдельно, поэтому их проектирование может находиться на различных этапах. Например, интерфейсные классы уже реализованы, а структура классов предметной области еще только уточняется. Обычно проектирование начинается, когда какой-либо фрагмент предметной области достаточно полно описан в процессе анализа.

Этап 1. Цель анализа – максимально полное описание задачи. На этом этапе:

– Выполняется анализ предметной области задачи. Анализ предметной области позволяет выделить ее сущности, определить первоначальные требования к функциональности и определить границы проекта. По результатам анализа разрабатывается структурная схема программного продукта, на которой показываются основные объекты и сообщения, передаваемые между ними.

– Строится объектная декомпозиция разрабатываемой системы. При объектной декомпозиции система разбивается на объекты или компоненты, которые взаимодействуют друг с другом, обмениваясь сообщениями. Сообщения описывают или представляют собой некоторые события.

– Определяются важнейшие особенности поведения объектов – описание абстракций.

Например, необходимо разработать программный продукт «Телефонная книга» для мобильного телефона. На первом этапе определяются основные требования к данному приложению: оно должно работать на операционной системе Android, поддерживать хранение до 1 000 записей и их быструю сортировку. Выполняется поиск наиболее подходящего алгоритма сортировки. Выбирается быстрый алгоритм сортировки. Основными компонентами приложения будут запись телефонной книги и менеджер функций.

Этап 2. Проектирование. Различают:

– логическое проектирование, при котором принимаемые решения практически не зависят от условий эксплуатации;

– физическое проектирование, при котором приходится принимать во внимание факторы эксплуатации.

Логическое проектирование заключается в разработке структуры классов. Определяются поля для хранения составляющих состояния объектов и алгоритмы методов, реализующих аспекты поведения объектов. При этом используются такие приемы разработки классов как наследование, композиция, наполнение, полиморфизм. Результатом является иерархия или диаграмма классов, отражающие взаимосвязь классов и их описание.

Физическое проектирование включает объединение описаний классов в модули, выбор схемы их подключения (статическая или динамическая компоновка), определение способов взаимодействия с оборудованием, с операционной системой и/или другим программным обеспечением (например, базами данных, сетевыми программами), обеспечение синхронизации процессов для систем параллельной обработки и т. д.

На втором этапе (продолжение примера с предыдущей сцены) для разрабатываемого приложения «Телефонная книга» строится структура классов: класс «Запись» с полями «Имя», «Фамилия», «Телефонный номер». Класс «Менеджер» с методами «Добавление», «Удаление», «Редактирование», «Сортировка». Определяются функции, предоставляемые операционной системой Android для работы с приложением.

Этап 3. Эволюция системы. Эволюция – это процесс поэтапной реализации и подключения классов к проекту. В первую очередь происходит создание основной программы или проекта будущего программного продукта. Затем реализуются и подключаются классы, так чтобы создать грубый, но, по возможности, работающий прототип будущей системы. Он тестируется и отлаживается. В итоге получается работоспособный прототип продукта, который может быть, например, показан заказчику для уточнения требований. Затем к системе подключается следующая группа классов, например, связанная с реализацией некоторого пункта меню. Полученный вариант также тестируется и отлаживается, и так далее, до реализации всех возможностей системы. Использование поэтапной реализации существенно упрощает тестирование и отладку программного продукта. На третьем этапе выполняется программная реализация разрабатываемого приложения «Телефонная книга».

Этап 4. Модификация. Модификация – это процесс добавления новых функциональных возможностей или изменение существующих свойств системы. Как правило, изменения затрагивают реализацию класса, оставляя без изменения его интерфейс, что при использовании ООП обычно обходится без особых неприятностей, так как процесс изменений затрагивает локальную область. Изменение интерфейса – также не очень сложная задача, но ее решение может повлечь за собой необходимость согласования процессов взаимодействия объектов, что потребует изменений в других классах программы. Однако сокращение количества параметров в интерфейсной части по сравнению с модульным программированием существенно облегчает и этот процесс. Простота модификации позволяет сравнительно легко адаптировать программные системы к изменяющимся условиям эксплуатации, что увеличивает время жизни систем, на разработку которых затрачиваются огромные временные и материальные ресурсы. На четвертом этапе для разрабатываемого приложения «Телефонная книга» добавляется новая функциональная возможность – функция добавления изображения в запись телефонной книги.

Вопросы для самопроверки

1. Назовите характеристики объектно- ориентированного программирования.
2. Сформулируйте принципы объектно-ориентированного программирования.
3. В чем заключаются принципы абстракции и инкапсуляции данных и инкапсуляции?
4. В чем заключаются принципы наследования и полиморфизма?

5. Назовите этапы объектно-ориентированного программиронвания.
6. Охарактеризуйте этап анализа объектно –ориентированного проектирования.
7. Охарактеризуйте этап проектирования.
8. Охарактеризуйте этап эволюции.
9. Охарактеризуйте этап модификации.

Самостоятельная работа

Разработка мультимедийной презентации по теме «Принципы объектно-ориентированного проектирования» (2 ч.)

Тема 1.7. Эффективность и оптимизация программ

1.7.1. Основные критерии эффективности программного продукта

Программные продукты имеют многообразие показателей качества, которые отражают следующие аспекты:

- насколько хорошо (просто, надежно, эффективно) можно использовать программный продукт;
- насколько легко эксплуатировать программный продукт;
- можно ли использовать программный продукт при изменении условия его применения и др.

Дерево характеристик качества программных продуктов представлено на рис. 18.

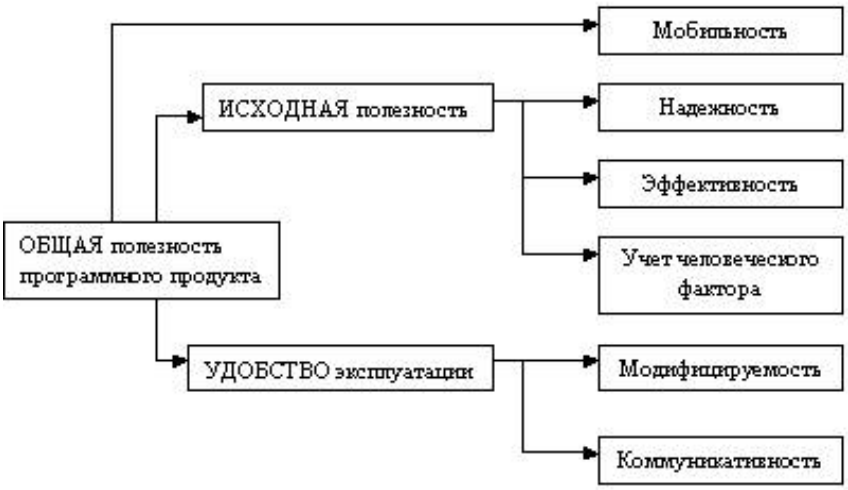


Рис. 18. Дерево характеристик качества программных продуктов.

Мобильность программных продуктов означает их независимость от технического комплекса системы обработки данных, операционной среды, сетевой технологии обработки данных, специфики предметной области и т.п. Мобильный (многоплатформный) программный продукт может быть установлен на различных моделях компьютеров и операционных систем, без ограничений на его эксплуатацию в условиях вычислительной сети. Функции обработки такого программного продукта пригодны для массового использования без каких-либо изменений.

Надежность работы программного продукта определяется бессбойностью и устойчивостью в работе программ, точностью выполнения предписанных функций обработки, возможностью диагностики возникающих в процессе работы программ ошибок.

Расход вычислительных ресурсов оценивается через объем внешней памяти для размещения программ и объем оперативной памяти для запуска программ.

Учет человеческого фактора означает обеспечение дружественного интерфейса для работы конечного пользователя, наличие контекстно-зависимой подсказки или обучающей системы в составе программного средства, хорошей документации для освоения и использования заложенных в программном средстве функциональных возможностей, анализ и диагностику возникших ошибок и др.

Модифицируемость программных продуктов означает способность к внесению изменений, например расширение функций обработки, переход на другую техническую базу обработки и т.п.

Коммуникативность программных продуктов основана на максимально возможной их интеграции с другими программами, обеспечении обмена данными в общих форматах представления (экспорт/импорт баз данных, внедрение или связывание объектов обработки и др.).

В условиях существования рынка программных продуктов важными характеристиками являются:

- стоимость,
- количество продаж;
- время нахождения на рынке (длительность продаж);
- известность фирмы-разработчика и программы;
- наличие программных продуктов аналогичного назначения.

Программные продукты массового распространения продаются по ценам, которые учитывают спрос и конъюнктуру рынка (наличие и цены программ-конкурентов). Большое значение имеет проводимый фирмой маркетинг, который включает:

- формирование политики цен для завоевания рынка;
- широкую рекламную кампанию программного продукта;
- создание торговой сети для реализации программного продукта (так называемые дилерские и дистрибьютерные центры);
- обеспечение сопровождения и гарантийного обслуживания пользователей программного продукта, создание горячей линии (оперативный ответ на возникающие в процессе эксплуатации программных продуктов вопросы);
- обучение пользователей программного продукта.

Эффективность программного продукта оценивается как с позиций прямого его назначения - требований пользователя, так и с точки зрения расхода вычислительных ресурсов, необходимых для его эксплуатации.

Главным критерием эффективности программ является распределение ресурсов вычислительных систем. Неравномерность задач по допустимому времени задержки или допустимой вероятности пропуска решений, а также различия параметров вычислительных систем, позволяют изменить качество решения задач выделением соответствующих ресурсов вычислительных систем. Упорядочивание последовательности решения задач и рациональное использование ресурсов вычислительных систем сокращает запаздывание в решении задач, и в некоторой степени приводит к эквивалентно повышению производительности вычислительных систем. Производительность является одним из важнейших критериев эффективности вычислительных систем в целом и методов распределения ресурсов в частности. Существуют такие понятия, как относительный и абсолютный приоритеты. При распределении буферной памяти для приема и выдачи сообщений применяются буферные накопители, объем которых ограничивает эффективность использования. Ограничение буферных накопителей зависит от их структурного построения и распределения имеющейся памяти на зоны. На эффективность существенно влияют степень заполнения памяти и передача информации на накопители на обработку.

1.7.2. Принципы и приемы оптимизации

Оптимизация программы - это улучшение какой-либо характеристики программы, называемой критерием оптимизации. Оптимизация программ в основном выполняется по двум основным критериям: быстродействие и объему используемых данных.

Производительность приложения определяется самым узким его участком, поэтому в первую очередь нужно определить части программы, на которых будет выполняться оптимизация. Процесс оптимизации следует начать с профилировки программы. ***Профилировкой*** называют измерение производительности как всей программы, так и отдельных ее фрагментов, с целью нахождения «горячих точек» - тех участков программы, на выполнение которых расходуется наибольшее количество времени. При этом важно отметить, что ликвидация не самых «горячих» точек программы, практически не увеличивает ее быстродействия.

Основная цель профилировки – это исследование характера поведения приложения во всех его точках. В зависимости от степени детализации в качестве «точки» рассматривается как отдельная машинная команда, так и целая конструкция высокого языка - функция, цикл, процедура. Сложная программа состоит из большого числа функций. Нет смысла оптимизировать их все – трудоемкость такого подхода будет выше выгод, полученных от оптимизации программы цели-

ком. Для начала необходимо локализовать участки кода с максимальной вычислительной трудоемкостью. Участки программы, которые в наибольшей степени влияют на ее производительность, в силу наиболее частого выполнения или своей ресурсоемкости называются критическим кодом. В поиске критического кода программы используют профайлеры (профилировщики) – специальные программы, которые измеряют временные затраты на выполнение участков кода программы. Профилировщики представляют возможности для оптимизации программ. К таким программам относятся Intel VTune, AMD Code Analyst, profile.exe и множество других. Наиболее мощным из них на сегодняшний день является пакет от Intel. Эта программа позволяет измерить время обработки каждой команды и вывести полную статистику о состоянии процессора при выполнении каждой команды.

Большинство современных профилировщиков поддерживают следующий набор базовых операций:

• определение общего времени исполнения каждой точки программы;
• определение удельного времени исполнения каждой точки программы;
• определение причины и/или источника конфликтов;
• определение количества вызовов той или иной точки программы;
• определение степени покрытия программы.

Основные правила оптимизации:

1. Прежде чем приступать к оптимизации, необходимо иметь надежно работающий неоптимизированный вариант.

2. Основной прирост оптимизации дает не учет особенностей системы, а алгоритмическая оптимизация.

3. Обнаружив профилировщиком узкие места необходимо произвести оптимизацию в рамках языка высокого уровня.

Возможны ситуации, где в неудовлетворительной производительности кода виноваты процессор или подсистема памяти, а не компилятор. Лишь после анализа листинга следует приступать к ассемблерной оптимизации.

Оптимизация начинается с выделения профилировщиком критического кода и анализа его неоптимальности. Причем каждое внесенное изменение необходимо проверять профилировщиком. После завершения оптимизации локального фрагмента программы, необходимо выполнить контрольную профилировку всей программы целиком на предмет обнаружения новых появившихся «горячих точек».

Проводя оптимизацию, не следует забывать о ее цели. Фактически идеал недостижим, поэтому оптимизацию следует завершать когда:

1. Производительность программы признана удовлетворяющей;

2. В программе отсутствуют «горячие точки», то есть количество инструкций равномерно распределено по все программе, и дальнейшая оптимизация потребует переписывания большого количества кода;

3. Сложность алгоритма настолько высока, что не представляется возможным дальнейшая оптимизация без значительных временных затрат;

4. Критическая зависимость от платформы, когда дальнейшая машинно-зависимая оптимизация приведет к потере совместимости с одной из целевых платформ.

Ко всем методам оптимизации алгоритма предъявляются следующие требования:

1. оптимизация должна быть по возможности максимально машинно- независимой и переносимой на другие платформы (операционные системы) без существенных потерь эффективности.

2. оптимизация не должна увеличивать трудоемкость разработки (в том числе тестирования) приложения более чем на 10-15%.

3. оптимизирующий алгоритм должен давать выигрыш не менее чем на 20-25% в скорости выполнения.

4. оптимизация не должна допускать безболезненное внесение изменений.

Алгоритмические приемы оптимизации

Приемы оптимизации программы можно разделить на алгоритмические и машинно-зависимые способы. В случае использования алгоритмических приемов оптимизации используются различные математические и логические методы для улучшения параметров алгоритма. Такой способ оптимизации невозможно автоматизировать, успешность его применения зависит от программиста. Способность программиста к алгоритмической оптимизации программы зависит от его

понимания предметной области: владения им базовых концепций применяемых алгоритмов и особенностей предметной области программы.

В первую очередь это замена алгоритмов на более быстродействующие. Часто бывает, что более простой алгоритм показывает низкую производительность по сравнению с более сложными. Тогда, возможна замена эквивалентных алгоритмов, например, замена пузырьковой сортировки массива на быструю сортировку.

В некоторых случаях возможна оптимизация программы за счет снижение точности. В зависимости от особенностей предметной области возможно уменьшить разрядность представления чисел или перейти от выполнения операций с числами с плавающей запятой к целым числам или числам с фиксированной запятой.

На практике используется весьма широкий набор машинно-независимых оптимизирующих преобразований, что связано с большим разнообразием неоптимальностей. К ним относятся:

разгрузка участков повторяемости - это такой способ оптимизации, который состоит в вынесении вычислений из многократно исполняемых участков программы на участки программы, редко исполняемые. К этому виду преобразования относятся различные чистки зон, тел циклов и тел рекурсивных процедур, когда инвариантные по результату выполнения выражения, исполняемые при каждом прохождении участка повторяемости, выносятся из него. Если размещение осуществляется перед входом в участок повторяемости, то эту ситуацию называют чисткой вверх, если же за выходом из участка повторяемости, то чисткой вниз;

упрощение действий - этот способ оптимизации ориентирован на улучшение программы за счет замены групп (как правило, удаленных друг от друга) вычислений на группу вычислений, дающий тот же результат с точки зрения всей программы, но имеющих меньшую сложность;

чистка программы - данный способ повышает качество программы за счет удаления из нее ненужных объектов и конструкций. Набор преобразований этого типа включает в себя следующие оптимизации: удаление идентичных операторов, удаление из программы операторов, недостижимых по управлению от начального, удаление несущественных операторов, то есть операторов, не влияющих на результат программы, удаление процедур, к которым нет обращений, удаление неиспользуемых переменных и другие;

экономия памяти и оптимизация работы с памятью - улучшения быстродействия возможно за счет уменьшения объема памяти, отводимой под информационные объекты программы в каждом ее исполнении;

реализация действий - это способ повышения быстродействия программы за счет выполнения определенных ее вычислений на этапе трансляции;

сокращение программы и другие методы.

Машинно-зависимые приемы оптимизации

Машинно-зависимые используют особенности устройства и работы конкретной системы. Ярким примером машинно-зависимой оптимизации является векторизация операций, т.е. использование потоковых расширений процессора, таких как MMX (MultiMedia eXtensions), SSE (Streaming SIMD Extensions) и т.п. Машино-зависимую оптимизацию можно выполнять двумя различными способами. Первый способ основан на понимании работы кодогенератора компилятора, его алгоритма и рекомендуется для приложений, в которых компилятор выбирается в начале проекта и в дальнейшем не меняется. При использовании такого способа преобразуется исходный код программы, написанный на языке высокого уровня. Для тех проектов, в которых заранее не известен компилятор (OpenSource проекты, кроссплатформенные приложения) применятся другой способ, основанный на замещении ресурсоемких участков кода ассемблерными вставками. При такой оптимизации ухудшается переносимость кода на другие платформы. Машинно-зависимые способы оптимизации довольно хорошо автоматизируются и большую часть их выполняют оптимизирующие компиляторы. Однако всегда остаются моменты в программе, которые можно оптимизировать вручную.

Вопросы для самопроверки

1. Какие аспекты отражают показатели качества программных продуктов?
2. Перечислите показатели качества программных продуктов.
3. Охарактеризуйте показатели качества программных продуктов: мобильность, надежность, расход вычислительных ресурсов.
4. Охарактеризуйте показатели качества программных продуктов: учет человеческого фактора, модифицируемость, коммуникативность, эффективность.
5. Назовите характеристики программных продуктов.

6. Что такое опитмизация программ?

7. Дайте определение профилировки.

8. Какие участки программы называются «горячими точками»?

9. Какие программы-профилировщики вы знаете? Назовите базовые операции, которые эти программы поддерживают.

10. Сформулируйте основные правила оптимизации.

11. В чем заключаются машинно – зависимые приемы оптимизации?

12. В чем заключаются алгоритмические приемы оптимизации?

Самостоятельная работа

1. Аналитический обзор литературы по теме «Количественные характеристики надежности программ» (2 ч.)

2. Выполнение упражнения на применение простых приемов оптимизации программ в среде Turbo Pascal:

3. Составить программу вычисления значения многочлена $ax^4 + bx^3 + cx^2 + dx = e$ (2 ч.)

Тема 1.8. Отладка программ

1.8.1. Понятие об ошибке программного обеспечения

Программирование сложный интеллектуальный процесс, сущность и основные закономерности которого еще недостаточно изучены. В программировании не может быть незначительных ошибок или несущественных погрешностей. Пропуск запятой или отсутствие пробела между символами делает программу неработоспособной.

Систематические характеры ошибок могут служить ориентиром для разработчиков при распределении усилий при создании комплекса программ. Кроме того характеристики ошибок в процессе проектирования программного комплекса помогают:

- Оценивать реальное состояние проекта и планировать трудоемкость и срок до его завершения;
- Рассчитывать необходимую эффективность средств защиты от невыявленных ошибок;
- Оценивать требующиеся ресурсы ПК по памяти и производительности с учетом затрат на устранение ошибок;
- Проводить исследования и осуществлять адекватный выбор показательной сложности компонент и комплекса в целом, а также некоторые другие показатели качества.

Анализ первичных ошибок в программе производится на двух уровнях детализации:

1. дифференциально – с учетом типовых ошибок, сложности и степени автоматизации их обнаружения, затрат на корректировку
2. обобщенно – по суммарным характеристикам их обнаружения в зависимости от продолжительности разработки, эксплуатации и сопровождения ПС.

Классификация ошибок:

1. *Технологические ошибки*. В документации и фиксировании программ в памяти ПК составляют 5-10% от общего числа ошибок обнаруживаемых при отладке. Большинство технологических ошибок выявляется автоматически формализованными методами. Например, при ручной подготовки машинных носителей (перфокарты, магнитные ленты) исходные данные имеют вероятность искажения около 10^{-3} на символ или 10^{-4} на двоичный разряд, т.е. все зависит от селектирующих свойств человека.

2. *Программные ошибки* по количеству и типам в первую очередь определяются степенью автоматизации программирования и глубиной формализованного контроля текстов программ. Количество программных ошибок зависит от квалификации разработчика и от общего объема программного комплекса, от глубины логического и информационного взаимодействия модулей и от ряда других факторов.

3. *Алгоритмические ошибки* значительно труднее поддаются обнаружению методами формализованного автоматического контроля. К алгоритмическим следует отнести прежде всего ошибки обусловленные некорректной постановкой функциональных задач. Когда в спецификациях не полностью оговорены все условия, необходимые для получения правильного результата.

4. *Системные ошибки* – сложных комплексов программ определяются прежде всего неполной информацией о реальных процессах происходящих в источниках и потребителях информации.

На начальных стадиях проектирования не всегда удается точно сформулировать целевую задачу всей системы, а также целевые задачи основных групп программ и эти задачи уточняются в процессе проектирования. В соответствии с этим уточняются и конкретизируются техническое задание, описание программ, и выявляются отклонения от уточненного задания, которые могут квалифицироваться как системные ошибки.

1.8.2. Основные принципы отладки программных продуктов

Под отладкой понимается процесс, позволяющий получить программу, работающую с требующимися характеристиками в заданной области входных данных. Отладка тесно соприкасается с тестированием, предшествует ему, но в то же время осуществляется на основании тестирования, однако это скорее искусство выявления природы ошибки и ее локализации.

Методологии и стратегия отладки начинается не по завершению написания текста программы, а непосредственно в момент написания. Для легкости последующей отладки рекомендуется писать небольшие, хорошо закомментированные и специфицированные модули, которые выполняют узкие функции.

Во время отладки надо постараться определить природу ошибки, является ли это ошибка ошибкой аппаратуры (что вполне вероятно, если используется какое-то нестандартное оборудование, для которого разрабатывается программа), операционной системы (что маловероятно), компилятора (читайте инструкции) или собственной программы. Если ошибка в программе, постарайтесь определить модуль, в котором она возникает, исключите из рассмотрения наименее вероятные источники ошибки, попытайтесь сузить область поиска.

Проверьте, является ли ошибка повторяющейся или устойчивой.

Основные причины неповторяющейся ошибки:

неправильный ввод данных;

сбои в аппаратуре;

в системах реального времени и в мультипрограммных системах источником может оказаться другая программа, операционная система.

Часто устойчивая ошибка может показаться случайной. Например, при отладке процедур, написанных на Turbo PASCAL, интегрированным отладчиком ошибок не возникает, а при нормальной работе программы имеет место устойчивая ошибка, иногда приводящая к краху системы. Этот частный пример может иметь несколько объяснений.

Например, при пошаговой отладке процедуры, имеющей локальную переменную, которая не инициализируется при вызове, и подразумевается, что ее значение по умолчанию должно равняться нулю, видимости ошибки не возникает. На самом деле это не так. Локальные переменные размещаются в стеке, и их начальное значение не определено. При пошаговой отладке среда Turbo PASCAL каждый раз сохраняет и восстанавливает заполненный нулями стек программы. В реальной работе эта ошибка всплывет. Она легко отслеживается такими отладчиками, как Turbo DEBUGGER. Аналогичная ситуация может возникать, когда функции перед выходом не присваивают результата.

При работе с обработчиками прерываний исполнение программы в среде Turbo PASCAL выглядит безупречным, а по завершению своей работы «из командной строки» программа «вешает» систему. Здесь имеет место срабатывания «защитного» механизма отладчика Turbo PASCAL, который по завершению сеанса отладки *восстанавливает* измененные вектора прерываний. Найти такую ошибку, не подойдя системно к ее локализации, очень трудно, если не невозможно. Распространенный отладчик Turbo DEBUGGER точно так же восстанавливает вектора.

Как правило наибольшие трудности с отладкой испытывают программисты, пытающие свалить вину на аппаратуру, операционную систему, компилятор, и другие «внешние» причины.

Одним из способов впоследствии системно подходить к поиску и устранению оши-

бок является ведение журнала учета собственных ошибок. Памятуя о допущенной однажды ошибке, гораздо легче просто избежать ее или локализовать возможное место ее появления.

Будьте скрупулезны, методичны и логичны в поиске ошибок. Применяйте систематический подход к поиску ошибки. Прежде всего, проверяйте самые простые предположения, в них легче и быстрее найти ошибку или убедиться в ее отсутствии.

Ничего не принимайте на веру. Например, доводы «это очень простой модуль, в нем не может быть ошибки», «изменения в работающем модуле были минимальны», «еще вчера это работало правильно» и тому подобные не являются гарантией отсутствия ошибки.

Беседуйте со своими коллегами. Очень часто ошибка лежит на поверхности, и вы просто «пригляделись» к программе. Свежий взгляд, иногда, позволяет не только выявить ошибку, но и повысить эффективность работы кода.

Проверяйте свои предположения с помощью тестовых программ. Иногда ошибки «накладываются» друг на друга совершенно независимо. Тогда выяснить их природу становится труднее во много раз. Не жалейте времени на написание «заглушек», которые имитируют работу того или иного блока. Это поможет выявить дополнительные ошибки, о существовании которых вы могли просто не подозревать.

Для отладки наиболее часто используют разнообразные отладчики, позволяющие контролировать значения переменных, задавать точки останова, обеспечивать пошаговое исполнение и тому подобное.

Другим, хорошо забытым, способом отладки является включение в программу отладочных элементов, которые распечатывают значения подозреваемых переменных, выдают сообщение о нормальной работе того или другого модуля.

Тем не менее, наилучшими критериями отладки являются опыт и терпение.

Вопросы для самопроверки

1. На каких уровнях в программе производится анализ первичных ошибок?
2. Для каких целей нужны характеристики ошибок в процессе проектирования программного комплекса?
3. Приведите классификацию ошибок.
4. Охарактеризуйте классы ошибок технологических и программных.
5. Охарактеризуйте классы ошибок алгоритмических и системных.
6. Назовите основные принципы отладки.
7. Назовите возможные причины неповторяющейся ошибки.

Самостоятельная работа

Подготовка доклада «Возможности встроенного отладчика интегрированной среды Turbo Pascal» 3 ч.)

Тема 1.9. Методы тестирования программ

1.9.1. Основные принципы тестирования программных продуктов

В целом разработчики различают *дефекты* программного обеспечения и *сбои.* В случае сбоя программа ведет себя не так, как ожидает пользователь. Дефект — это ошибка/неточность, которая может быть (а может и не быть) следствием сбоя.

Общепринятая практика состоит в том, что после завершения продукта и до передачи его заказчику независимой группой тестировщиков проводится тестирование ПО. Эта практика часто выражается в виде отдельной фазы тестирования (в общем цикле разработки ПО), которая часто используется для компенсирования задержек, возникающих на предыдущих стадиях разработки. Другая практика состоит в том, что тестирование начинается вместе с началом проекта и продолжается параллельно созданию продукта до завершения проекта. Второй путь обычно требует больших трудозатрат, но качество тестирования при этом будет выше.

Уровни тестирования:

- **модульное тестирование.** Тестируется минимально возможный для тестирования ком-

понент, например отдельный класс или функция;

- **интеграционное тестирование.** Проверяется, есть ли какие-либо проблемы в интерфейсах и взаимодействии между интегрируемыми компонентами, например, не передается информация, передается некорректная информация;
- **системное тестирование.** Тестируется интегрированная система на ее соответствие исходным требованиям:
- **альфа-тестирование** — имитация реальной работы с системой штатными разработчиками либо реальная работа с системой потенциальными пользователями/заказчиком на стороне разработчика. Часто альфа-тестирование применяется для законченного продукта в качестве внутреннего приемочного тестирования. Иногда альфа тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться ПО;
- **бета-тестирование** — в некоторых случаях выполняется распространение версии с ограничениями (по функциональности или времени работы) для некоторой группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

Выполнение программы с целью обнаружения ошибок называется ***тестированием***. Виды ошибок и способы их обнаружения приведены в таблице 8.

Виды программных ошибок и способы их обнаружения Таблица 8

Виды программных ошибок	Способы их обнаружения
Синтаксические	Статический контроль и диагностика компиляторами и компоновщиком
Ошибки выполнения, выявляемые автоматически: а) переполнение, защита памяти; б) несоответствие типов; в) зацикливание	Динамический контроль: аппаратурой процессора; run-time системы программирования; операционной системой — по превышению лимита времени
Программа не соответствует спецификации	Целенаправленное тестирование
Спецификация не соответствует требованиям	Испытания, бета-тестирование

Эффективность контроля 1-го вида зависит и от языка, и от компилятора. Контроль 2-го вида осуществляется с помощью исключений — Exceptions и весьма полезен для проверки правдоподобности промежуточных результатов. ***Тест*** — это набор контрольных входных данных совместно с ожидаемыми результатами. В число входных данных времязависимых программ входят события и временные параметры. Ключевой вопрос полнота тестирования: *какое* количество *каких* тестов гарантирует, возможно, более полную проверку программы? Исчерпывающая проверка на всем множестве входных данных недостижима. Пример: программа, вычисляющая функцию двух переменных: $Y=f(X, Z)$. Если X, Y, Z — real, то полное число тестов $(2^{32})^2=2^{64}\approx10^{31}$. Если на каждый тест тратить 1 мс, то 2^{64}мс=800 млн. лет

Следовательно:

в любой нетривиальной программе на любой стадии ее готовности содержатся необнаруженные ошибки;

тестирование — технико-экономическая проблема, основанная на компромиссе время — полнота. Поэтому нужно стремиться к возможно меньшему количеству хороших тестов с желательными свойствами.

Детективность: тест должен **с** большой вероятностью обнаруживать возможные ошибки

Покрывающая способность: один тест должен выявлять как можно больше ошибок.

Воспроизводимость: ошибка должна выявляться независимо от изменяющихся условий (например, от временных соотношений) — это труднодостижимо для времязависимых программ, результаты которых часто невоспроизводимы.

1.9.2. Методы структурного и функционального тестирования программного обеспечения

Возможен целый ряд подходов к стратегии проектирования тестов. Рассмотрим два крайних подхода. Первый состоит в том, что тесты проектируются на основе внешних спецификаций программ и модулей, либо спецификаций сопряжения программы или модуля (функциональное тестирование). Программа при этом рассматривается как черный ящик (стратегия «черного ящика»). Существо такого подхода - проверить соответствует ли программа внешним спецификациям. При этом логика модуля совершенно не принимается во внимание. Второй подход основан на анализе логики программы (стратегия «белого ящика»). Существо подхода - в проверке каждого пути, каждой ветви алгоритма (структурное тестирование). При этом внешняя спецификация во внимание не принимается. Полное тестирование программы невозможно. Тест для любой программы будет обязательно неполным, то есть тестирование не гарантирует отсутствие всех ошибок. Стратегия проектирования тестов заключается в том, чтобы попытаться уменьшить эту неполноту насколько это возможно. При этом ключевым вопросом является следующий: какое подмножество всех возможных тестов имеет наивысшую вероятность обнаружения ошибок при ограниченных времени, трудовых затратах, стоимости, машинном времени и т.п. Наихудшей из всех методологий является случайный набор тестов, так как он имеет малую вероятность быть оптимальным. Рекомендуется следующая процедура разработки тестов:

-разрабатывать тесты, используя методы стратегии “черного ящика”;

- дополнительное тестирование, используя методы стратегии “белого ящика”.

Методы стратегии «белого ящика»:

покрытия операторов;

покрытия решений (покрытия переходов);

покрытия условий;

критерий решений (условий);

комбинаторного покрытия условий.

Покрытие операторов

Критерием покрытия является **выполнение каждого оператора программы, по крайней мере, один раз**. Это *метод покрытия операторов*. К сожалению, это слабый критерий, так как выполнение каждого оператора, по крайней мере, один раз есть необходимое, но недостаточное условие для приемлемого тестирования по принципу белого ящика (рис. 3).

Предположим, что на рис. 3 представлена небольшая программа, которая должна быть протестирована.

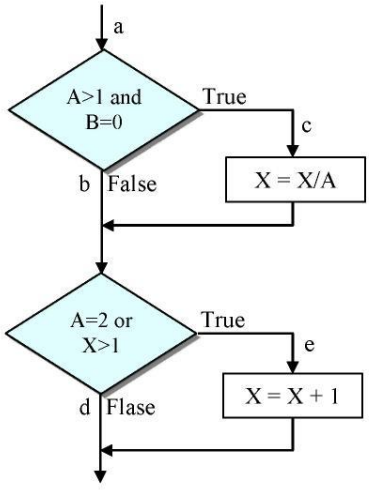


Рис. 19. Блок-схема небольшого участка программы, который должен быть протестирован

Можно выполнить каждый оператор, записав один-единственный тест, который реализовал бы путь *ace*. Иными словами, если бы в точке *a* были установлены значения A = 2, B = 0 и X = 3, каждый оператор выполнялся бы один раз (в действительности *X* может принимать любое значение).

К сожалению, этот критерий хуже, чем он кажется на первый взгляд.

Например, пусть первое решение записано как «*или*», а не как «*и*». Тогда при тестировании с помощью данного критерия эта ошибка не будет обнаружена. Пусть второе решение записано в программе как X > 0 (во втором операторе условия); эта ошибка также не будет обнаружена. Кроме того, существует путь, в котором X не изменяется (путь *abd*). Если здесь ошибка, то и она не будет обнаружена. Таким образом, критерий покрытия операторов является настолько слабым, что его обычно не используют.

Покрытие решений

Более сильный критерий покрытия логики программы (и метод тестирования) известен как *покрытие решений*, или *покрытие переходов*. Согласно данному критерию должно быть записано достаточное число тестов, такое, что каждое решение на этих тестах примет значение *истина* и *ложь* по крайней мере один раз. Иными словами, **каждое направление перехода должно быть реализовано по крайней мере один раз.** Примерами операторов перехода или решений являются операторы **while** или **if**. Можно показать, что покрытие решений обычно удовлетворяет критерию покрытия операторов. Поскольку каждый оператор лежит на некотором пути, исходящем либо из оператора перехода, либо из точки входа программы, при выполнении каждого направления перехода каждый оператор должен быть выполнен. Однако существует, по крайней мере, три исключения. Первое – патологическая ситуация, когда программа не имеет решений. Второе встречается в программах или подпрограммах с несколькими точками входа (например, в программах на языке Ассемблера); данный оператор может быть выполнен только в том случае, если выполнение программы начинается с соответствующей точки входа. Третье исключение – операторы внутри **switch**-конструкций; выполнение каждого направления перехода не обязательно будет вызывать выполнение всех **case**-единнц. Так как покрытие операторов считается необходимым условием, покрытие решений, которое представляется более сильным критерием, должно включать покрытие операторов. Следовательно, покрытие решений требует, чтобы каждое решение имело результатом значения *истина* и *ложь* и при этом каждый оператор выполнялся бы, по крайней мере, один раз. Альтернативный и более легкий способ выражения этого требования состоит в том, чтобы каждое решение имело результатом значения *истина* и *ложь* и что каждой точке входа (включая каждую **case**-единицу) должно быть передано управление при вызове программы, по крайней мере, один раз.

Изложенное выше предполагает только двузначные решения или переходы и должно быть модифицировано для программ, содержащих многозначные решения (как для **case**-единиц). Критерием для них является выполнение каждого возможного результата всех решений, по крайней мере, один раз и передача управления при вызове программы или подпрограммы каждой точке входа, по крайней мере, один раз. В программе, представленной на рис. 3, покрытие решений может быть выполнено двумя тестами, покрывающими либо пути *ace* и *abd,* либо пути *acd* и *abe.* Если мы выбираем последнее альтернативное покрытие, то входами двух тестов являются A = 3, B = 0, X = 3 и A = 2, B = 1, X = 1.

Покрытие условий

Покрытие решений – более сильный критерий, чем покрытие операторов, но и он имеет свои недостатки. Например, путь, где X не изменяется (если выбрано первое альтернативное покрытие), будет проверен с вероятностью 50 %. Если во втором решении существует ошибка (например, X < 1 вместо X > 1), то ошибка не будет обнаружена двумя тестами предыдущего примера.

Лучшим критерием (и методом) по сравнению с предыдущим является *покрытие условий*. В этом случае записывают число тестов, достаточное для того, чтобы **все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз**.

Программа на рис. 3 имеет четыре условия: A > 1, B = 0, A = 2 и X > 1. Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где A > 1, A ≤ l, B = 0 и B ≠ 0 в точке *a* и A = 2, A ≠ 2, X > 1 и X ≤ 1 в точке *b.* Тесты, удовлетворяющие критерию покрытия условий, и соответствующие им пути:

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

Заметим, что, хотя аналогичное число тестов для этого примера уже было создано, покрытие условий обычно лучше покрытия решений, поскольку оно *может* (но не всегда) вызвать выполнение решений в условиях, не реализуемых при покрытии решений.

Хотя применение критерия покрытия условий на первый взгляд удовлетворяет критерию покрытия решений, это не всегда так. Если тестируется решение if(A && B), то при критерии покрытия условий требовались бы два теста – A есть *истина,* B есть *ложь* и A есть *ложь*, B есть *истина.* Но в этом случае не выполнялось бы тело условия. Тесты критерия покрытия условий для ранее рассмотренного примера покрывают результаты всех решений, но это только случайное совпадение. Например, два альтернативных теста:

1. A = 1, B = 0, X = 3.
2. A = 2, B = 1, X = 1,

покрывают результаты всех условий, но только два из четырех результатов решений (они оба покрывают путь *abe* и, следовательно, не выполняют результат *истина* первого решения и результат *ложь* второго решения).

Покрытие решений/условий

Очевидным следствием из этой дилеммы является критерий, названный *покрытием решений/условий.* Он требует такого достаточного набора тестов, чтобы **все возможные результаты каждого условия в решении, все результаты каждого решения выполнялись, по крайней мере, один раз и каждой точке входа передавалось управление, по крайней мере, один раз**.

Недостатком критерия покрытия решений/условий является невозможность его применения для выполнения всех результатов всех условий; часто подобное выполнение имеет место вследствие того, что определенные условия скрыты другими условиями. В качестве примера рассмотрим приведенную на рис. 4 схему передач управления в коде, генерируемым компилятором языка, программы рис. 3.

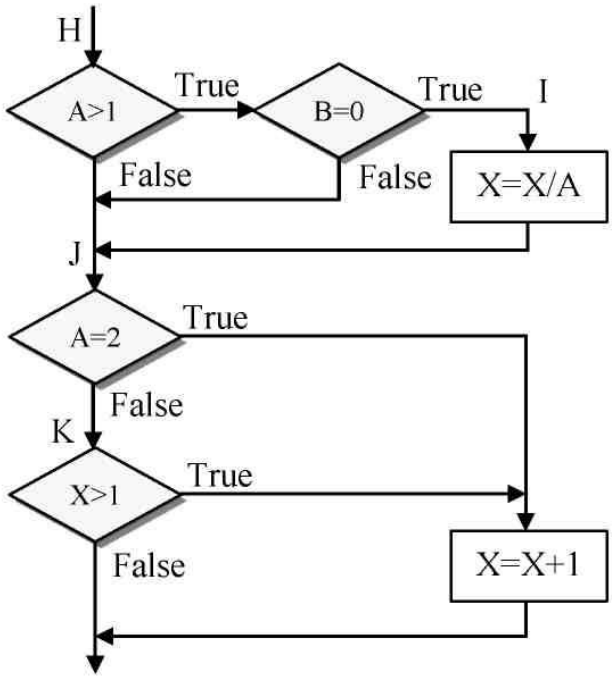


Рис.20. Блок-схема машинного кода программы, изображенной на рис. 3

Многоусловные решения исходной программы здесь разбиты на отдельные решения и переходы, поскольку большинство компьютеров не имеет команд, реализующих решения со многими исходами. Наиболее полное покрытие тестами в этом случае осуществляется таким образом, чтобы выполнялись все возможные результаты каждого простого решения. Два предыдущих теста критерия покрытия решений не выполняют этого; они недостаточны для выполнения результата *ложь* решения H и результата *истина* решения K. Набор тестов для критерия покрытия условий

такой программы также является неполным; два теста (которые случайно удовлетворяют также и критерию покрытия решений/условий) не вызывают выполнения результата *ложь* решения I и результата *истина* решения K. Причина этого заключается в том, что, как показано на рис. 4, результаты условий в выражениях *и* и *или* могут скрывать и блокировать действие других условий. Например, если условие *и* есть *ложь,* то никакое из последующих условий в выражении не будет выполнено. Аналогично если условие *или* есть *истина,* то никакое из последующих условий не будет выполнено. Следовательно, критерии покрытия условий и покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

Комбинаторное покрытие условий

Критерием, который решает эти и некоторые другие проблемы, является *комбинаторное покрытие условий*. Он требует создания такого числа тестов, чтобы **все возможные комбинации результатов условия в каждом решении и все точки входа выполнялись, по крайней мере, один раз**.

По этому критерию для программы на рис. 3 должны быть покрыты тестами следующие восемь комбинаций:

1. $A > 1$, $B = 0$. 2. $A > 1$, $B \neq 0$.

3. $A \leq 1$, $B = 0$. 4. $A \leq 1$, $B \neq 0$.

5. $A = 2$, $X > 1$. 6. $A = 2$, $X \leq 1$.

7. $A \neq 2$, $X > 1$. 8. $A \neq 2$, $X \leq 1$.

Заметим, что комбинации 5–8 представляют собой значения второго оператора **if**. Поскольку *X* может быть изменено до выполнения этого оператора, значения, необходимые для его проверки, следует восстановить, исходя из логики программы с тем, чтобы найти соответствующие входные значения.

Для того чтобы протестировать эти комбинации, необязательно использовать все восемь тестов. Фактически они могут быть покрыты четырьмя тестами. Приведем входные значения тестов и комбинации, которые они покрывают:

$A = 2$, $B = 0$, $X = 4$ покрывает 1, 5;

$A = 2$, $B = 1$, $X = 1$ покрывает 2, 6;

$A = 1$, $B = 0$, $X = 2$ покрывает 3, 7;

$A = 1$, $B = 1$, $X = 1$ покрывает 4, 8.

То, что четырем тестам соответствуют четыре различных пути на рис. 3, является случайным совпадением. На самом деле представленные выше тесты не покрывают всех путей, они пропускают путь *acd*.

В случае циклов число тестов для удовлетворения критерию комбинаторного покрытия условий обычно больше, чем число путей.

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является критерий, набор тестов которого:

1) вызывает выполнение всех результатов каждого решения, по крайней

мере, один раз;

2) передает управление каждой точке входа (например, точке входа, **case**-единице) по крайней мере один раз (чтобы обеспечить выполнение каждого оператора программы по крайней мере один раз).

Для программ, содержащих решения, каждое из которых имеет более одного условия, минимальный критерий состоит из набора тестов, вызывающих выполнение всех возможных комбинаций результатов условий в каждом решении и передающих управление каждой точке входа программы, по крайней мере, один раз.

Методы стратегии «чёрного ящика»: эквивалентного разбиения; анализа граничных значений; тестирования таблицы решений; тестирование модульных программ.

Эквивалентное разбиение

Тестирование программы ограничивается использованием небольшого подмножества всех возможных входных данных. Правильно выбранный тест этого подмножества должен обладать двумя свойствами:

– уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;

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

Указанные свойства, несмотря на их кажущееся подобие, описывают два различных положения. Во-первых, каждый тест должен включать столько различных входных условий, сколько это возможно, с тем, чтобы минимизировать общее число необходимых тестов. Во-вторых, необходимо пытаться разбить входную область программы на конечное число *классов эквивалентности* так, чтобы можно было предположить (конечно, не абсолютно уверенно), что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если один тест класса эквивалентности обнаруживает ошибку, то следует ожидать, что и все другие тесты этого класса эквивалентности будут обнаруживать ту же самую ошибку. Наоборот, если тест не обнаруживает ошибки, то следует ожидать, что ни один тест этого класса эквивалентности не будет обнаруживать ошибки (в том случае, когда некоторое подмножество класса эквивалентности не попадает в пределы любого другого класса эквивалентности, так как классы эквивалентности могут пересекаться).

Эти два положения составляют основу методологии тестирования по принципу черного ящика, известной как *эквивалентное разбиение*. Второе положение используется для разработки набора «интересных» условий, которые должны быть протестированы, а первое – для разработки минимального набора тестов, покрывающих эти условия.

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

1) выделение классов эквивалентности;

2) построение тестов.

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более групп. различают два типа классов эквивалентности: *правильные классы эквивалентности*, представляющие правильные входные данные программы, и *неправильные классы эквивалентности*, представляющие все другие возможные состояния условий (т. е. ошибочные входные значения). Таким образом, придерживаются одного из принципов тестирования о необходимости сосредоточивать внимание на неправильных или неожиданных условиях.

Построение тестов включает в себя:

1. Назначение каждому классу эквивалентности уникального номера.

2. Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых правильных классов эквивалентности, до тех пор пока все правильные классы эквивалентности не будут покрыты (только не общими) тестами.

3. Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы эквивалентности не будут покрыты тестами.

Причина покрытия неправильных классов эквивалентности индивидуальными тестами состоит в том, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами.

Таким образом, **метод эквивалентного разбиения требует покрыть минимальным количеством тестов все правильные классы эквивалентности и индивидуальными тестами все неправильные классы эквивалентности**

Хотя эквивалентное разбиение значительно лучше случайного выбора тестов, оно все же имеет недостатки (т. е. пропускает определенные типы высокоэффективных тестов). Следующие два метода – анализ граничных значений и использование функциональных диаграмм (диаграмм причинно-следственных связей cause-effect graphing) – свободны от многих недостатков, присущих эквивалентному разбиению.

Граничные условия – это ситуации, возникающие непосредственно на, выше или ниже границ входных и выходных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения в двух отношениях:

1. Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных значений осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.

2. При разработке тестов рассматривают не только входные условия (пространство входов), но и *пространство результатов* (т. е. выходные классы эквивалентности).

Достаточно трудно описать принимаемые решения при анализе граничных значений, так как это требует определенной степени творчества и специализации в рассматриваемой проблеме. (Следовательно, анализ граничных значений, как и многие другие аспекты тестирования, в значительной мере основывается на способностях человеческого интеллекта.)

Тем не менее существует несколько общих правил этого метода.

1. Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений. Например, если правильная область входных значений есть от –1.0 до +1.0, то нужно написать тесты для ситуаций –1.0, 1.0, –1.001 и 1.001.

2. Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то получить тесты для 0, 1, 255 и 256 записей.

3. Использовать первое правило для каждого выходного условия. Например, если программа вычисляет ежемесячный расход и если минимум расхода составляет $0.00, а максимум – $1165.25, то построить тесты, которые вызывают расходы с $0.00 и $1165.25. Кроме того, построить, если это возможно, тесты, которые вызывают отрицательный расход и расход больше 1165.25 дол. Заметим, что важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей (например, при рассмотрении подпрограммы вычисления синуса). Не всегда также можно получить результат вне выходной области, но тем не менее стоит рассмотреть эту возможность.

4. Использовать второе правило для каждого выходного условия. Например, если система информационного поиска отображает на экране наиболее релевантные статьи в зависимости от входного запроса, но никак не более четырех рефератов, то построить тесты, такие, чтобы программа отображала нуль, один и четыре реферата, и тест, который мог бы вызвать выполнение программы с ошибочным отображением пяти рефератов.

5. Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

6. Попробовать свои силы в поиске других граничных условий.

Таким образом, **метод анализа граничных значений требует покрыть тестом каждую границу классов эквивалентности входных данных и построить тесты, позволяющие получать минимальные и максимальные значения выходных данных, а также тесты, обеспечивающие нарушение границ выходных**

Применение функциональных диаграмм

Одним из недостатков анализа граничных значений и эквивалентного разбиения является то, что они не исследуют *комбинаций* входных условий. Например, пусть программа из приведенного выше примера не выполняется, если произведение числа вопросов и числа студентов превышает некоторый предел (например, объем памяти). Такая ошибка не обязательно будет обнаружена тестированием граничных значений.

Тестирование комбинаций входных условий – непростая задача, поскольку даже при построенном эквивалентном разбиении входных условий число комбинаций обычно астрономически велико. Если нет систематического способа выбора подмножества входных условий, то, как правило, выбирается произвольное подмножество, приводящее к неэффективному тесту.

Метод функциональных диаграмм или диаграмм причинноследственных связей помогает систематически выбирать высокорезультативные тесты. Он дает полезный побочный эффект, так как позволяет обнаруживать неполноту и неоднозначность исходных спецификаций.

Функциональная диаграмма представляет собой формальный язык, на который транслируется спецификация, написанная на естественном языке. Диаграмме можно сопоставить цифровую логическую цепь (комбинаторную логическую сеть), но для ее описания используется более простая нотация (форма записи), чем обычная форма записи, принятая в электронике.

Вопросы для самопроверки

1. Что такое дефекты и сбои программного обеспечения?
2. Назовите уровни тестирования и охарактеризуйте их.

3. Что называется тестированием? Тестом?
4. Какими свойствами должен обладать тест?
5. В чем заключается стратегия тестирования «белого ящика»?
6. В чем заключается стратегия тестирования «черного ящика»?
7. Назовите методы стратегии «белого ящика».
8. Назовите методы стратегии «черного ящика».
9. Назовите критерий метода покрытия операторов.
10. Назовите критерий метода покрытия решений.
11. Назовите критерий метода покрытия решений / условий.
12. Назовите критерий метода комбинаторного покрытия условий.
13. Назовите критерий метода эквивалентного разбиения.
14. Назовите критерий метода анализа граничных значений.
15. В чем заключается метод функциональных диаграмм?

Самостоятельная работа

Составление опорного конспекта по теме «Методы тестирования программ» (2 ч.)

Тема 1.10. Сопровождение программ

1.10.1. Виды программных документов. Методы разработки программной документации. Технологии разработки документации

Составление программной документации — очень важный процесс. Стандарт, определяющий процессы жизненного цикла программного обеспечения, даже предусматривает специальный процесс, посвященный указанному вопросу. При этом на каждый программный продукт должна разрабатываться документация двух типов: для пользователей различных групп и для разработчиков. Отсутствие документации любого типа для конкретного программного продукта недопустимо.

При подготовке документации не следует забывать, что она разрабатывается для того, чтобы ее можно было использовать, и потому она должна содержать все необходимые сведения.

Виды программных документов

К программным относят документы, содержащие сведения, необходимые для разработки, сопровождения и эксплуатации программного обеспечения. Документирование программного обеспечения осуществляется в соответствии с Единой системой программной документации (ГОСТ 19.XXX). Так, ГОСТ 19.101—77 устанавливает виды программных документов для программного обеспечения различных типов. Ниже перечислены основные программные документы по этому стандарту и указано, какую информацию они должны содержать.

Спецификация должна содержать перечень и краткое описание назначения всех файлов программного обеспечения, в том числе и файлов документации на него, и является обязательной для программных систем, а также их компонентов, имеющих самостоятельное применение.

Ведомость держателей подлинников (код вида документа — 05) должна содержать список предприятий, на которых хранятся подлинники программных документов. Необходимость этого документа определяется на этапе разработки и утверждения технического задания только для программного обеспечения со сложной архитектурой.

Текст программы (код вида документа — 12) должен содержать текст программы с необходимыми комментариями. Необходимость этого документа определяется на этапе разработки и утверждения технического задания.

Описание программы (код вида документа — 13) должно содержать сведения о логической структуре и функционировании программы.

Ведомость эксплуатационных документов (код вида документа — 20) должна содержать перечень эксплуатационных документов на программу, к которым относятся до-

кументы с кодами 30, 31, 32, 33, 34, 35, 46. Необходимость этого документа также определяется на этапе разработки и утверждения технического задания.

Формуляр (код вида документа — 30) должен содержать основные характеристики программного обеспечения, комплектность и сведения об эксплуатации программы.

Описание применения (код вида документа — 31) должно содержать сведения о назначении программного обеспечения, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств.

Руководство системного программиста (код вида документа — 32) должно содержать сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения.

Руководство программиста (код вида документа — 33) должно содержать сведения для эксплуатации программного обеспечения.

Руководство оператора (код вида документа — 34) должно содержать сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программного обеспечения.

Описание языка (код вида документа — 35) должно содержать описание синтаксиса и семантики языка. Руководство по техническому обслуживанию (код вида документа — 46) должно содержать сведения для применения тестовых и диагностических программ при обслуживании технических средств.

Программа и методика испытаний (код вида документа — 51) должны содержать требования, подлежащие проверке при испытании программного обеспечения, а также порядок и методы их контроля.

Пояснительная записка (код вида документа — 81) должна содержать информацию о структуре и конкретных компонентах программного обеспечения, в том числе схемы алгоритмов, их общее описание, а также обоснование принятых технических и технико-экономических решений. Составляется на стадии эскизного и технического проектов. Прочие документы (коды вида документа — 90—99) могут составляться на любых стадиях разработки, т. е. на стадиях эскизного, технического и рабочего проектов.

Пояснительная записка должна содержать всю информацию, необходимую для сопровождения и модификации программного обеспечения: сведения о его структуре и конкретных компонентах, общее описание алгоритмов и их схемы, а также обоснование принятых технических и технико-экономических решений.

Содержание пояснительной записки по стандарту (ГОСТ 19.404—79) должно включать следующие разделы:

- введение;
- назначение и область применения;
- технические характеристики;
- ожидаемые технико-экономические показатели;
- источники, используемые при разработке.

В разделе Введение указывают наименование программы и документа, на основании которого ведется разработка.

В разделе Назначение и область применения указывают назначение программы и дают краткую характеристику области применения.

Раздел Технические характеристики должен содержать следующие подразделы:

• постановку задачи, описание применяемых математических методов и допущений и ограничений, связанных с выбранным математическим аппаратом;

• описание алгоритмов и функционирования программы с обоснованием принятых решений;

• описание и обоснование выбора способа организации входных и выходных данных;

• описание и обоснование выбора состава технических и программных средств на основании проведенных расчетов или анализов.

В разделе Ожидаемые технико-экономические показатели указывают технико-экономические показатели, обосновывающие преимущество выбранного варианта технического решения.

В разделе Источники, использованные при разработке, указывают перечень научно-технических публикаций, нормативно- технических документов и других научно-технических материалов, на которые есть ссылки в исходном тексте.

Пояснительная записка составляется профессионалами в области разработки программного обеспечения для специалистов того же уровня квалификации. Следовательно, в ней уместно использовать специальную терминологию, ссылаться на специальную литературу и т. п.

Руководство пользователя

В настоящее время часто используют еще один эксплуатационный документ, в который отчасти входит руководство системного программиста, программиста и оператора. Этот документ называют Руководством пользователя. Появление такого документа явилось следствием широкого распространения персональных компьютеров, работая на которых пользователи совмещают в своем лице трех указанных специалистов.

Составление документации для пользователей имеет свои особенности, связанные с тем, что пользователь, как правило, не является профессионалом в области разработки программного обеспечения. В книге С. Дж. Гримм даны рекомендации по написанию подобной программной документации:

• учитывайте интересы пользователей — руководство должно содержать все инструкции, необходимые пользователю;

• излагайте ясно, используйте короткие предложения;

• избегайте технического жаргона и узкоспециальной терминологии, если все же необходимо использовать некоторые термины, то их следует пояснить;

• будьте точны и рациональны — длинные и запутанные руководства обычно никто не читает, например, лучше привести рисунок формы, чем долго ее описывать.

Руководство пользователя, как правило, содержит следующие разделы:

• общие сведения о программном продукте;

• описание установки;

• описание запуска;

• инструкции по работе (или описание пользовательского интерфейса);

• сообщения пользователю.

Раздел Общие сведения о программе обычно содержит наименование программного продукта, краткое описание его функций, реализованных методов и возможных областей применения.

Раздел Установка обычно содержит подробное описание действий по установке программного продукта и сообщений, которые при этом могут быть получены.

В разделе Запуск, как правило, описаны действия по запуску программного продукта и сообщений, которые при этом могут быть получены.

Раздел Инструкции по работе обычно содержит описание режимов работы, форматов ввода-вывода информации и возможных настроек.

Раздел Сообщения пользователю должен содержать перечень возможных сообщений, описание их содержания и действий, которые необходимо предпринять по этим сообщениям.

Технологии разработки документации

В технической документации повтор — явление не менее редкое, чем в программе. Приходится описывать сходные функции программ и систем, сходные экранные формы, сходные действия пользователей. Описания одних и тех же объектов приходится полностью или частично дублировать в разных документах. Поскольку материал, из которого состоит техническая документация, — текст на естественном языке, применить к нему модульный подход оказывается сложнее, чем к программному коду. Принцип единого источника и реализующие его технологии позволяют применить модульный принцип к документированию. Сначала проектируется структура единого источника, разрабатываются шаблоны и стили оформления, устанавливается и настраивается инструментарий для формирования документов. Эта стадия отнимает время в начале проекта и предъявляет довольно высокие требования к квалификации разработчика технической документации. Но затем начинается рутинная работа по написанию текста и его загрузке в единый источник. В любой момент на основе введенного текста можно сформировать документы, в большей или меньшей степени готовности. На этой стадии мы получаем отдачу от сделанных вложений. Любой другой фрагмент, который должен появляться в нескольких местах, при необходимости достаточно исправить однократно в едином источнике.

Принцип единого источника в документированиипомогает организовать работу коллектива соавторов, распределив между ними более-менее изолированные подзадачи. Таким образом, единый источник — это не только техническое, но еще и организационное решение. Взаимодействие между участниками разработки технической документации тоже может быть автоматизировано. Крупные интегрированные среды для автоматизации документирования, такие, как AuthorIT, SiberSafe, RoboHELP, предоставляют для этого различные средства: управления правами доступа, версионный контроль, планирование работ и т.п. Но все это относится, скорее, к документообороту отдела документирования или техническому документообороту проекта

Технология DITA была разработана в корпорации IBM в 1999–2000 г., объявлено о ней было в 2001 г. Общая схема создания документа такова:

1.Автор пишет текст документации, размечая его в соответствии с синтаксисом используемого языка разметки. В результате получается текстовый файл, в котором как таковой текст перемежается условными обозначениями, отражающими функции отдельных фрагментов внутри этого текста.

2.К полученному на первом этапе текстовому файлу применяется программа-конвертор. На выходе получается документ в том или ином формате, например, PDF или HTML. Оформление текста в этом документе определяется стилями, которые были подготовлены заранее самим автором или кем-то другим.

Arbortext — семейство продуктов американской компании-гиганта PTC (Parametric Technology Corp.), предназначенное для создания технической документации. В своей основе Arbortext использует принцип единого источника, в основе которого лежит формат XML. Высокая степень интегрированности продуктов Arbortext с другими продуктами PTC позволяет осуществлять разработку документации не просто в XML редакторе, но в специальной среде разработки. Эта среда дает возможность не только создавать XML, но и хранить документацию, создавать интерактивные иллюстрации, организовывать параллельную работу по созданию документации нескольким участникам процесса, в то время, как большинство редакторов — замкнутые решения для локального пользователя.

Первая версия языка разметки DocBook была создана еще в 1991 г.. Сегодня язык разметки DocBook/XML применяется разработчиками технической документации во

всем мире. Основой технологической платформы DocBook/XML служит одноименный проблемно-ориентированный язык разметки. Он предназначен для записи текста технической документации на программы, алгоритмические языки, компьютерное оборудование и другие решения в области информационных технологий, чем принципиально отличается от большинства форматов хранения текстовых данных (но не XML-языков!). В языке DocBook/XML предусмотрены средства описания фрагментов, свойственных технической документации, например, специальными элементами полагается выделять названия элементов интерфейса, обозначения клавиш, имена переменных, термины, различные врезки (замечания, подсказки, предупреждения), листинги, описания выполняемых пользователем процедур. Разметка, задаваемая языком DocBook/XML, носит преимущественно функциональный характер: автору предписано указывать роль, которую тот или иной фрагмент играет в тексте, а не способ его внешнего оформления. Такой подход сковывает автора, зато позволяет добиться заведомой независимости содержания и оформления выходного документа и унифицировать некоторые важные качества стиля изложения при работе нескольких авторов в одном проекте.

Способ формирования документов сильно зависит от особенностей используемого инструментария. Тем не менее, если рассмотренные механизмы полноценно реализованы, можно выделить несколько обязательных стадий формирования документа (табл. 9).

Стадии формирования документа **Таблица 9**

Стадия	Действия	Входные данные	Выходные данные
I. Компиляция документа			
I.1. Компоновка документа	выбор нужных фрагментов из единого источника и соединение их в один документ в соответствии с шаблоном аспекта	предмет, аспект, единый источник	скомпонованный документ
I.2. Профилирование документа	применение к документу фильтров, определяемых параметрами профиля	параметры профиля, скомпонованный документ	профилированный документ
I.3. Корректировка документа	устранение дефектов, которые могут возникнуть в результате предыдущих стадий (например, перекрестных ссылок, для которых отсутствуют ссылочные элементы)	профилированный документ	скомпилированный документ
II. Сборка документа			
II.1. Применение к документу оформления	формирование единого документа, содержащего и текст и оформление	стили оформления, скомпилированный документ	оформленный документ (например в формате XSL:FO или RTF)
II.2. Конвертация документа в целевой формат	формирование электронного документа с помощью специализированного конвертора (например FO-процессора или компилятора файлов справки)	имя и параметры запуска конвертора, оформленный документ	документ в нужном электронном формате (например PDF или CHM)

Формирование документов на основе шаблонов и единого источника — основной принцип, на котором сегодня построены все наиболее известные и развитые инструментальные средства для автоматизации документирования. Второе важное свойство технологий единого источника заключается в возможности придать одному и тому же тексту разное оформление и конвертировать его в любой нужный электронный формат. Интегрированные среды автоматизации документирования часто снабжены средствами для организации групповой работы и технического документооборота, но уже побочные, хотя и

чрезвычайно полезные функции; для реализации которых вполне можно использовать специализированные программные продукты. Чем больше возможностей по настройке правил формирования документов предоставляет та или иная технология, чем эффективнее она позволяет бороться с повторами на все уровнях, тем она лучше и мощнее.

Вопросы для самопроверки

1. Назовите виды программных документов.
2. Опишите содержание программных документов: спецификации и ведомости держателей подлинников.
3. Опишите содержание программных документов: формуляра и руководства программиста.
4. Опишите содержание программных документов: программы и методики испытаний и руководства пользователя.
5. В чем заключается принцип единого источника в технологии разработки документации?
6. Назовите и охарактеризуйте технологии разработки документации.
7. Опишите стадию компиляции документа.
8. Опишите стадию компоновки документа.
9. Перечислите принципы, на которых сегодня построены все наиболее известные и развитые инструментальные средства для автоматизации документирования.

Самостоятельная работа

1. Составление опорного конспекта по теме «Технологии разработки документации»(2 ч.)
2. Составление словаря терминов по разделу «Разработка программного модуля» (2 ч.)

РАЗДЕЛ 4. ЗАДАНИЯ И МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИКУМУ по разделу 1 «Разработка программного модуля» МДК 01.01 «Системное программирование» профессионального модуля «Разработка программных модулей программного обеспечения для компьютерных систем»

Практическая работа №1

Тема: Проведение анализа программ с точки зрения стиля. Составление удобочитаемой программы (2 часа)

Цель работы: получить навыки по анализа выбранного стиля программирования

Теоретическая часть

Этап проектирования программ оказывает влияние на стиль программирования, надежность, эффективность, отладку, тестирование и эксплуатационное свойство программ. Таким образом, это важнейшая часть любой программной разработки.

Текст программы нужен прежде всего самому программисту, а также другим людям, с которыми он совместно работает над проектом. Поэтому для того, чтобы работа была эффективной, программа должна быть легко читаемой, ее структура должна соответствовать структуре и алгоритму решаемой задачи. Как этого добиться? Надо следовать правилам хорошего стиля программирования.

Под ***стилем программирования*** понимается внутренне согласованная совокупность базовых конструкций программ и способов их композиции, обладающая общими фундаментальными особенностями, как логическими, так и алгоритмическими. Стиль включает также совокупность базовых концепций, связанных с этими программами.

При оформлении текста программы хороший стиль программирования предполагает:

- использование комментариев;

- использование несущих смысловую нагрузку имен переменных, процедур и функций;

- использование отступов;

- использование пустых строк.

Следование правилам хорошего стиля программирования значительно уменьшает вероятность появления ошибок на этапе набора текста, делает программу легко читаемой, что, в свою очередь, облегчает процессы отладки и внесения изменений.

Четкого критерия оценки степени соответствия программы хорошему стилю программирования не существует. Вместе с тем достаточно одного взгляда, чтобы понять, соответствует программа хорошему стилю или нет.

Сводить понятие стиля программирования только к правилам записи текста программы было бы неверно. Стиль, которого придерживается программист, проявляется во время работы программы.

Очевидно, что хороший программист должен следовать правилам хорошего стиля:

1. *Стремление к простоте*. Простота проектирования программ – это первый шаг, ведущий к получению легко читаемой программы. Необычное кодирование программы (например, использование скрытых возможностей машины) часто препятствует отладке программы и конечно затрудняет ее использование другими программистами. Структура программы должна раскрывать ее логику.

2. *Чтение программы*. Программу надо снабжать комментариями и параграфами.

3. *Описание задач*. Идеальная программа дает точный желаемый выход при неясно сформированных требованиях пользователя.

4. *Постановка задачи*. В данном разделе надо четко определить задачу, ее цели, этапы и конечный результат. Далее программист должен переписать спецификации задачи ориентируясь на ЭВМ. Необходимо сжатое, но полное описание программы. Затем программист и заказчик должны тщательно изучить написанные спецификации задачи, чтобы быть уверенным в ее правильном понимании.

5. *Выбор алгоритма*. Важнейшим шагом для получения эффективной и правильной программы является составление алгоритма. При этом предполагается правильный выбор языка и спецификации программы. Таким образом, хороший алгоритм необходимое, но недостаточное условие хорошей программы. Если пользователь формирует задачу в виде четкого алгоритма, то процесс проектирования существенно облегчается. Для эффективности необходимо рассмотреть несколько алгоритмов и из них выбрать наиболее эффективный.

6. *Описание данных.* Другим фактором сравнимым по значению с выбором алгоритма является описание данных. Хорошо продуманное описание данных существенно сокращает программу. Так, полезно, использовать массивы данных в том случае, когда это наиболее очевидный способ их организации. Другой пример такого рода – возможность использовать ссылки и указатели. Например, если нужно проследить отношения между родителями и их потомками на протяжении нескольких поколений, то легче всего это сделать с помощью ссылок и указателей.

7. *Выбор языка программирования.* Часто выбор языка программирования предоставлен данной выигрышной системе, или подготовкой программиста. Существуют серьезные основания для установления языковых стандартов для системы. Если применяют много разных языков для написания программ, то использование последних становится затруднительным.

8. *Универсальность.* Хорошая универсальная программа должна обрабатывать вырожденные случаи (например, число элементов равно 0 ил 1) и печатать сообщения об ошибке. Тогда программа является не только универсальной, но и защищенной от ошибок. Используйте в качестве компиляторов переменные, а не константы.

9. *Библиотеки.* Чтобы повысить эффективность разработки программ, облегчить отладку и тестирование, а следовательно и сократить работу по созданию программ используйте библиотеки. Тип библиотечных подпрограмм представляет собой функции и подпрограммы имеющиеся в наличии для данного языка программирования.

10. *Форматы ввода/вывода.* Форматы входных и выходных данных являются частью этапа проектирования, входные данные должны быть разработаны с учетом максимального удобства для пользователя и минимальные ошибки. Постоянство входных форматов, как правило, также способствуют уменьшению ошибок. Выходные спецификации могут сильно различаться. Иногда даются четкие инструкции и выходные данные подгоняются под определенный стандарт. Однако часто отсутствуют какие-либо указания и выходные данные подчас представляют собой страницы, заполненные числами без всякой идентификации. Выходная информация должна идентифицироваться без привлечения других источников. Выходные данные должны содержать: 1. идентификацию выходной информации, 2. описание записи, 3. дату, 4. нумерацию страниц. Кроме того, каждая напечатанная группа элементов должна быть помечена. При табличной форме выдачи должны быть помечены строки и столбцы.

Порядок выполнения работы

В файле 1.pas дано решение задачи, которая по введенному списку учебной группы, включающей для каждого учащегося дату рождения, год поступления в колледж, курс, группу, оценки каждого года обучения, упорядочивает список студентов по среднему баллу и получает его.

```
uses crt;
const
st = 25;
type
student = record
fio:       string[40];
data_rozhd: record
day: integer;
month: integer;
year: integer;
end;
god_post:  integer;
group:      string[8];
kurs:      integer;
ocenki:    array [1 .. 40] of integer;
sr_ball:   real;
end;
var
spisok:array [1..st] of student;
procedure vvod;
var i,j,k,sr:integer;
begin
for i:= 1 to st do
begin
clrscr;
writeln(i,'  fio           data rozhd   god post   group   kurs   ocenki');
gotoxy(1,2);
readln(spisok[i].fio);
gotoxy(18,2);
readln(spisok[i].data_rozhd.day,spisok[i].data_rozhd.month,spisok[i].data_roz
hd.year);
gotoxy(31,2);
readln(spisok[i].god_post);
gotoxy(41,2);
readln(spisok[i].group);
gotoxy(50,2);
readln(spisok[i].kurs);
```

```
sr:=0;
for j:=1 to spisok[i].kurs do
begin
gotoxy(56,1+j);
for k:=1 to 4 do
begin
readln(spisok[i].ocenki[(j-1)*4+k]);
gotoxy(56+2*(k),1+j);
sr:=sr+spisok[i].ocenki[(j-1)*4+k];
end;
end;
spisok[i].sr_ball:=sr/(spisok[i].kurs*4);
end;
end;
procedure vyvod;
var i,j,k,sr,str:integer;
sredn:real;
stud:student;
begin
for i:=1 to st-1 do
for j:=i+1 to st do
begin
if spisok[j].sr_ball < spisok[i].sr_ball then
begin
stud:=spisok[i];
spisok[i]:=spisok[j];
spisok[j]:=stud;
end;
end;
clrscr;
str:=2;
writeln('   fio           data_rozhd  god_post   group   kurs   ocenki
sr_ball');
for i:=1 to st do
begin
gotoxy(1,str);
writeln(spisok[i].fio);
gotoxy(18,str);
writeln(spisok[i].data_rozhd.day,spisok[i].data_rozhd.month,spisok[i].data_ro
zhd.year);
gotoxy(31,str);
writeln(spisok[i].god_post);
gotoxy(41,str);
writeln(spisok[i].group);
gotoxy(50,str);
writeln(spisok[i].kurs);
gotoxy(66,str);
writeln(spisok[i].sr_ball:4:1);
for j:=1 to spisok[i].kurs do
begin
gotoxy(56,str);
for k:=1 to 4 do
begin
writeln(spisok[i].ocenki[(j-1)*4+k]);
gotoxy(56+2*(k),str);
end;
str:=str+1;
end;
end;
end;
begin
vvod;
vyvod;
```

```
end.
```

1) Провести анализ программы с точки зрения стиля.
2) Составить по предложенному примеру удобочитаемую программу.
3) Привести характеристику выбранного стиля.

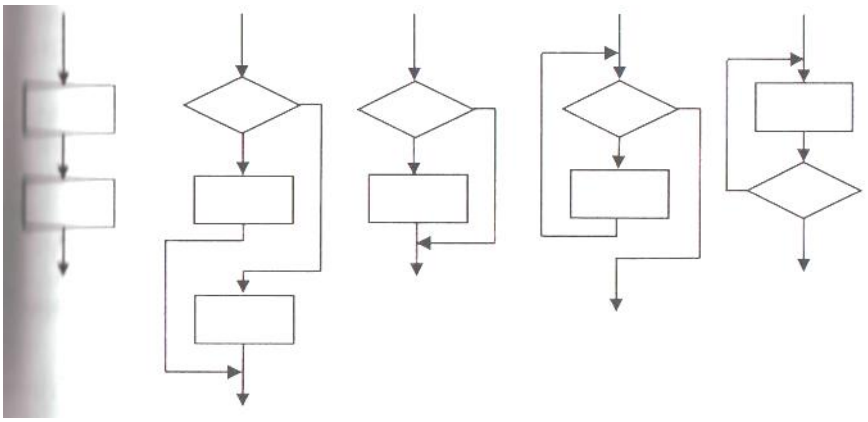
Практическая работа №2

Тема: Создание структурного алгоритма (2 часа)

Цель работы: применить принципы структурного программирования при решении практической задачи.

Теоретическая часть

Структурная блок-схема алгоритма — это блок-схема, которая может быть представлена как композиция элементных блок-схем (рис. 1): композиции — 1, а, выборов — 1,*б, в,* циклов с предусловием и постусловием — 1, *г, д.*



а ***б*** ***в*** ***г*** ***д***

Рис. 1. Элементы структурной блок-схемы:
а- композиции**;** *б, в* — выборов; *г, д* — циклов с предусловием и постусловием

Каждая из элементарных блок-схем имеет один вход и один выход. Отсюда следует, что у любой блок-схемы алгоритма, составленной из этих блок-схем, также будет один вход и один выход. Таким образом, под структурным программированием подразумевают разработку программ на основе алгоритмов, составленных из структурных блок-схем.

Приведенных элементарных блок-схем, вообще говоря, достаточно для построения любых хорошо структурированных алгоритмов.

Однако на практике могут возникать случаи, когда такое построение не будет простейшим. Поэтому структурное программирование допускает большее разнообразие элементарных блок-схем. Например, разветвление процесса обработки информации по *m* направлениям и преждевременный выход из цикла изображаются элементарными блок-схемами, приведенными на рис. 2

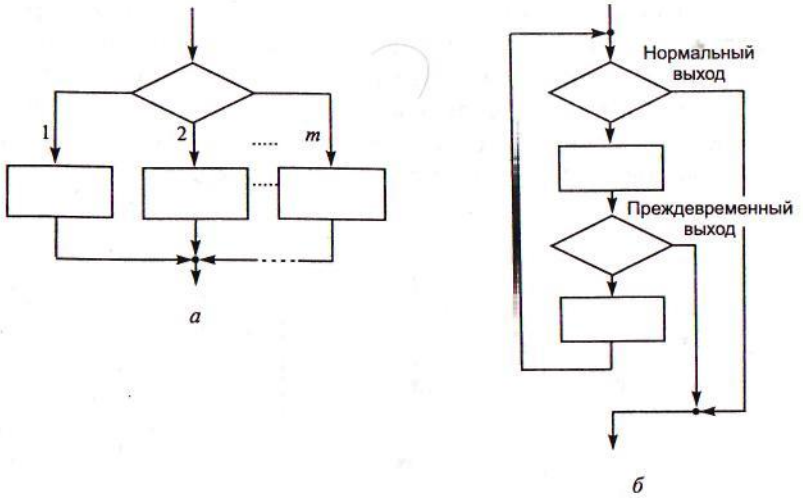


Рис. 2 Элементарные блок-схемы разветвления
по многим путям *(а)* и преждевременного выхода из цикла *(б)*

Структурное программирование сверху вниз, о котором говорилось ранее, — это метод составления программ сверху вниз на основе структурных блок-схем. В свою очередь программирование сверху вниз представляет собой процесс пошагового разбиения алгоритма на все более мелкие части с целью получить такие элементы, для которых легко написать конкретные команды. Продемонстрируем этот процесс на следующем примере.

Не вникая глубоко в проблемы параллельного программирования, отметим лишь то, что его методы предназначены для составления программ для многопроцессорных ЭВМ, или как их часто называют, супер-ЭВМ. Если алгоритм решения сложной задачи допускает независимую параллельную обработку каких-либо его ветвей, то принципиально для обсчета каждой ветви можно составить свою программу и поручить ее выполнение отдельному процессору многопроцессорной ЭВМ. Таким путем можно существенно уменьшить общее время решения задачи.

Среди алгоритмических языков высокого уровня возможностями параллельного программирования располагает FORTRAN, ADA, JAVA. Разработанный в свое время для этих целей язык MODULA-2 не нашел широкого практического применения.

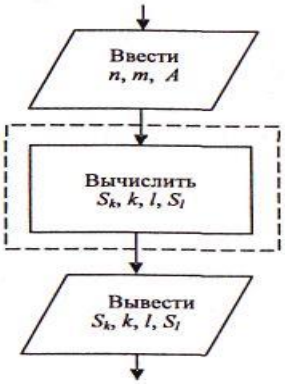
Порядок выполнения работы

Построить последовательность блок- схем, отвечающих принципам структурного программирования сверху вниз для решения задачи согласно вашему варианту

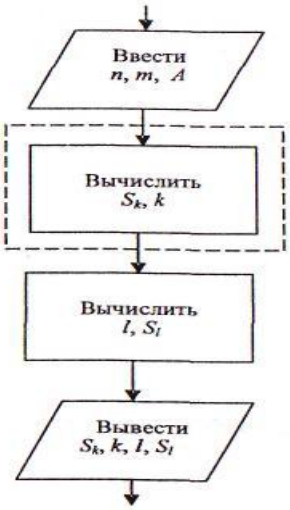
Пример. Дана прямоугольная матрица чисел A = $||a_{ij}||$, состоящая из m строк и n столбцов. Требуется найти строку $к$, сумма S_k элементов которой максимальна, найти в этой строке минимальный ее элемент a_{kl}, отметить номер соответствующего столбца l и вычислить сумму элементов этого столбца S_i

В связи с тем, что в матрице А может оказаться несколько строк с одинаковыми максимальными значениями сумм их элементов, а строке принадлежать несколько равных минимальных элементов, для определенности следует искать первую из строк с максимальной суммой элементов, а в ней — самый первый из минимальных элементов. Кроме этого, матрица А должна содержать не менее двух строк и двух столбцов.

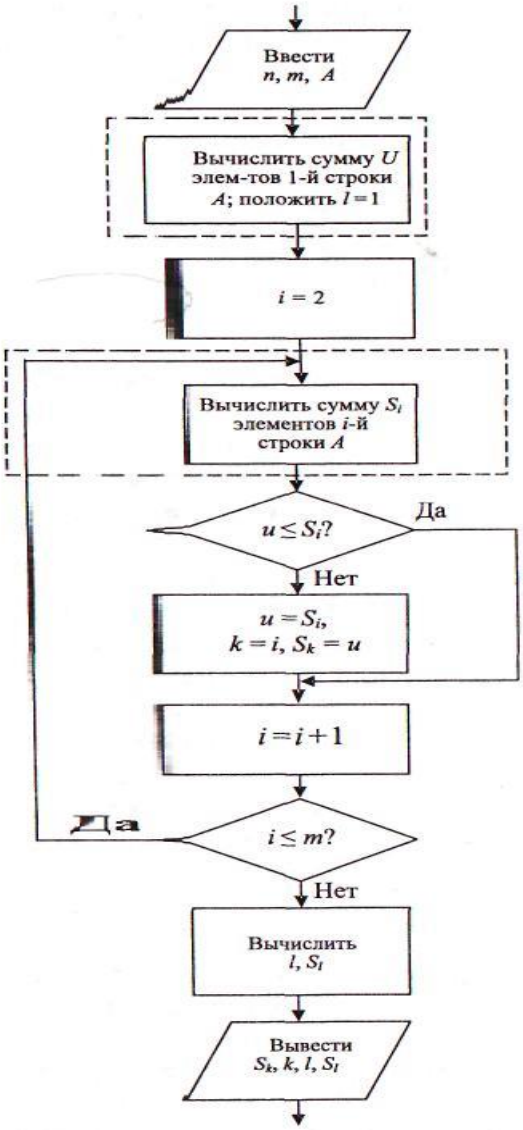
Последовательность блок-схем, отвечающих требованиям структурного программирования сверху вниз представлена на рис. 3. Пунктирными линиями обведены блоки, которые последовательно раскрываются до элементарных блок-схем.


Ввести
n, m, A
Вычислить
Sk, k, l, Sl
Вывести
Sk, k, l, Sl

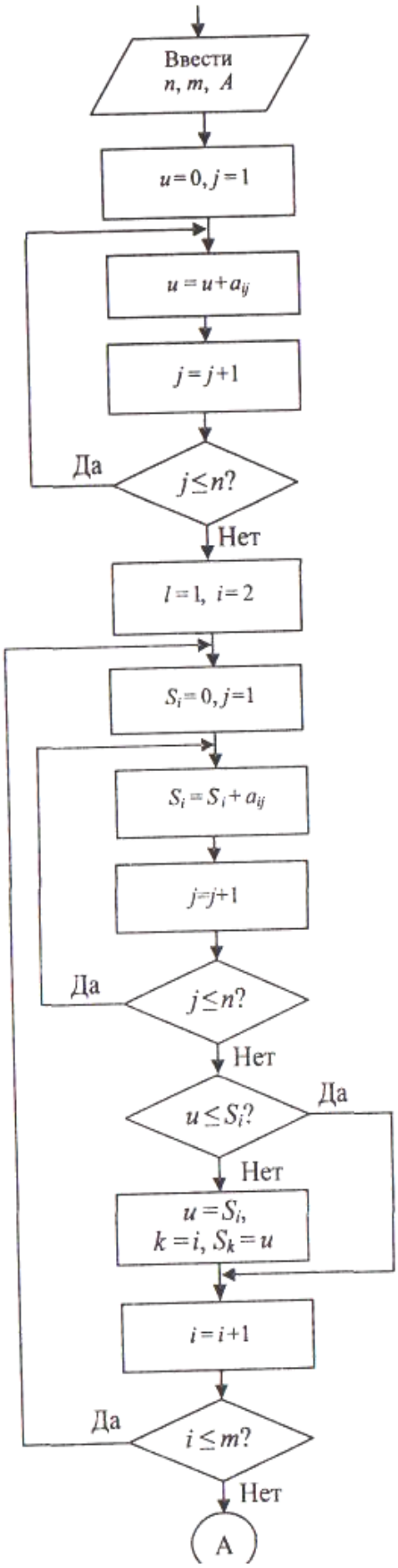
1

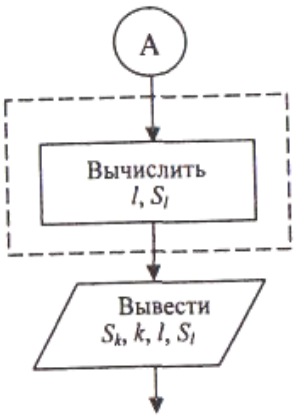

Ввести
n, m, A
Вычислить
Sk, k
Вычислить
l, Sl
Вывести
Sk, k, l, Sl

2


Ввести
n, m, A
Вычислить сумму U
элем-тов 1-й строки
A; положить l=1
i = 2
Вычислить сумму Si
элементов i-й
строки A
u ≤ Si?
Да
Нет
u = Si,
k = i, Sk = u
i = i + 1
Да
i ≤ m?
Нет
Вычислить
l, Sl
Вывести
Sk, k, l, Sl

3


Ввести
n, m, A
u=0, j=1
u = u+a_ij
j = j+1
Да
j≤n?
Нет
l=1, i=2
S_i=0, j=1
S_i = S_i + a_ij
j=j+1
Да
j≤n?
Нет
Да
u≤S_i?
Нет
u = S_i,
k = i, S_k = u
i = i+1
Да
i≤m?
Нет
A


A
Вычислить
l, S_l
Вывести
S_k, k, l, S_l

4

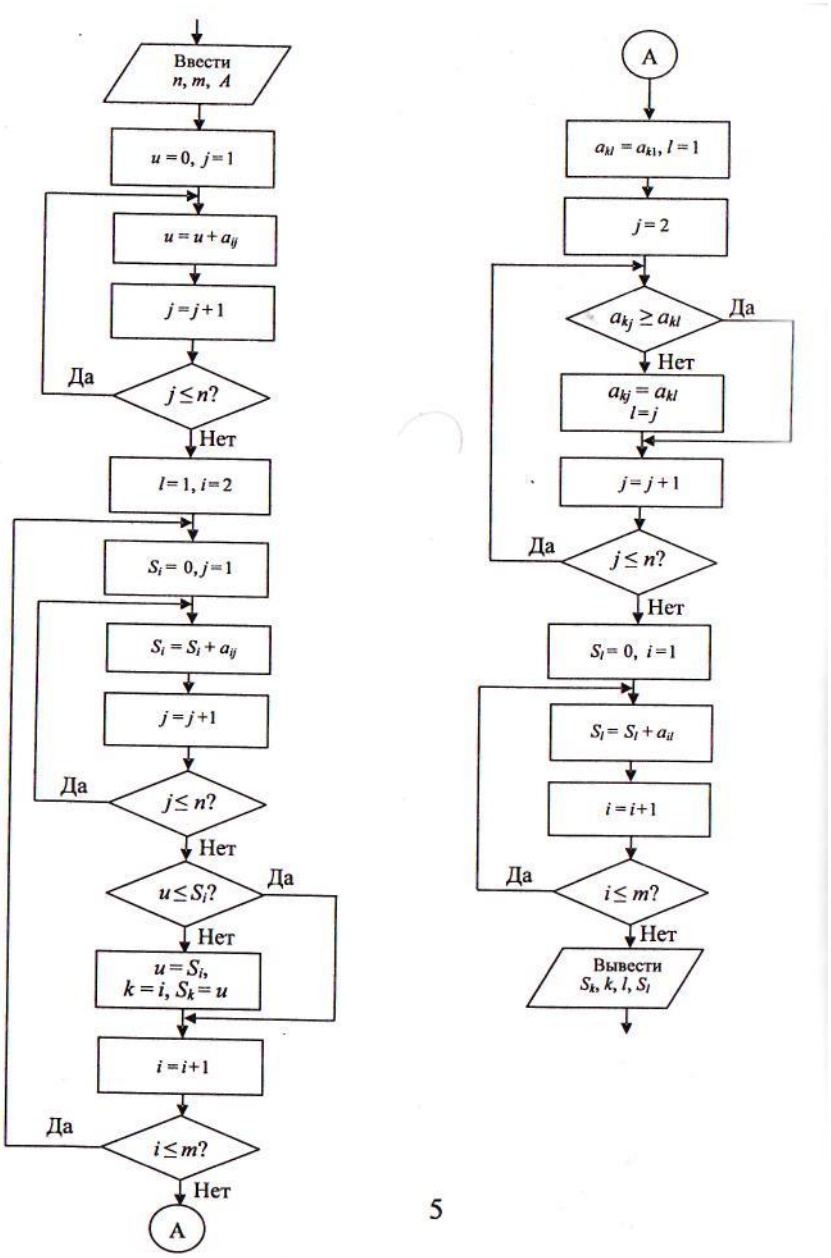


Рис. 3. Последовательность блок- схем 1-5, отвечающих принципам структурного программирования сверху вниз

РАЗДЕЛ 5. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ЛАБОРАТОРНОМУ ПРАКТИКУМУ

по разделу 1 «Разработка программного модуля» МДК 01.01 «Системное программирование» профессионального модуля «Разработка программных модулей программного обеспечения для компьютерных систем»

Лабораторная работа №1

Тема: Применение оптимизирующих компиляторов (2 часа)

Цель работы: изучить методы оптимизации программ, возможности оптимизирующих компиляторов.

Теоретическая часть

В Турбо Паскале выполняется несколько типов оптимизации кода компилируемой программы с целью повышения её быстродействия и уменьшения объёма занимаемой памяти.

Программирование арифметических операций.

Немалые резервы повышения скорости работы программы заключены в правильном программировании арифметических и логических выражений. Различные арифметические операции значительно различаются по быстродействию. Самыми быстрыми являются операции сложения и вычитания. Более медленным является умножение, затем идет деление. Программируя арифметические выражения, следует выбирать такую форму их записи, чтобы количество «медленных» операций было сведено к минимуму. Пусть необходимо вычислить многочлен 4-ой степени:

$ax^4 + bx^3 + cx^2 + dx = e$

В этом выражении содержится 10 умножений и 4 сложения. Это же выражение можно записать в виде $(((ax+b)x+c)x+d)x+e$. Такая форма записи называется схемой Горнера. В этом выражении 4 умножения и 4 сложения. Общее количество операций сократилось почти в 2 раза.

Свёртывание констант. Если операндами выражения являются константы ординального типа, то выражение вычисляется в период компиляции. То же относится к функциям **abs, sqr, succ, pred, odd, lo, hi, swap**, если их аргументами являются константы ординального типа.

Пример 1.

```
Var  k : integer;
Begin
  k:=abs(-15)+sqr(7)-6*12;
```

Для выполняемой программы записанный в исходном тексте оператор будет заменён оператором **k := -23;**

Если индексом массива является константа или выражение, состоящее из констант, то адрес элемента массива вычисляется во время компиляции. Например, доступ к элементам **a[sqr(7)+2,3*8] и a[61,24]** будет таким же эффективным, как и доступ к простой переменной.

Слияние констант. Если в одном блоке несколько раз встречается одна и та же строковая константа, то в объектном коде программы компилятор разместит только одну ее копию.

Пример 2.

```
  Writeln('Введите значение ','x =');
  ..............................
  Writeln('Введите значение ','y =');
  ..............................
  Writeln('Введите значение ','z =')
  ..............................
```

Для объектного кода программы это эквивалентно фрагменту

```
  Const  S = 'Введите значение ';
  Begin
    Writeln(S,'x =');
    ..............
    Writeln(S,'y =');
    ..............
    Writeln(S,'z =');
```

.....

Вычисление по короткой схеме. Логическое выражение в Турбо Паскале вычисляется по короткой схеме. Это означает, что вычисление такого выражения прекращается, как только его результат становится очевидным.

Пример 3.

```
If (x>0) and (x<100) and (y>1) and (y<x) then ...
While (x<0) or (y>1) or (z>x+y) do ...
```

Если x = -1, то вычисление обоих логических выражений прекращается после вычисления значения истинности первого операнда. В первом случае имеем для всего выражения значение **false**, во втором - значение true.

Удаление неиспользуемого кода. Операторы, которые никогда не будут выполняться, в объектный код программы не включаются.

Пусть в исходном тексте программы записан оператор

```
If false then y:=sqr(x)+1;
```

Такой оператор при компиляции программы игнорируется. Используя это свойство компилятора, можно в исходном тексте программы подготовить различные её варианты, формируемые при повторной компиляции.

Пример 4.

```
Const  KeyPrint = 0;  {   Ключ тестовой печати: }
                  { 0 - тестовая печать отсутствует ; }
Begin               { 1 - активизируется тестовая печать }
.............................
If KeyPrint>0 then
  Begin
    Печать массива A
  End;
```

Операторы отладочной печати включаются в объектный код программы, если до её компиляции было установлено значение константы **KeyPrint = 1.**

Эффективная компоновка. Процедуры и функции, к которым в программе нет ни одного обращения, в объектный код не включаются. Не включаются в объектный код также переменные, которые описаны в разделах **Var**, но в программе не используются. Данный тип оптимизации кода имеет особое значение при использовании модулей. Если в программе записана фраза

```
Uses Graph;
```

то в объектный код программы будут переписаны только те процедуры из модуля **Graph**, к которым имеются обращения. Здесь следует отметить, что при компиляции программы в память эффективная компоновка не работает. Этим объясняется, почему некоторые программы становятся меньше при компиляции их на диск.

Организация измерения времени исполнения программных функций.

При работе под Windows задача определения точного времени выполнения программы в общем случае не имеет решения. Дело в том, что в Windows существует планировщик заданий, который может в любой момент переключиться с Вашего приложения на другое. То есть всегда существует вероятность того, что в определенный момент времени после запоминания начального этапа работы программы, но **перед** запоминанием конечного этапа произойдет вытеснение Вашей задачи планировщиком, и результат засекания времени окажется непредсказуемым

Использование процедуры GetTime

Самый простой способ замерить время выполнения кода - это запомнить системное время в начале работы алгоритма, затем (по окончании его работы) снова получить системное время, и вычесть из него то, что было запомнено при старте. Проще всего для этой цели воспользоваться стандартной процедурой GetTime модуля Dos (ниже предлагается функция, возвращающая системное время в сотых долях секунды. Соответственно, время работы программы тоже вычисляется в сотых долях секунды).

```
Function GetTime: LongInt;
Var h, m, s, ms: Word;
begin
    Dos.GetTime(h, m, s, ms);
    GetTime := ms + 100 * (s + 60 * (m + 60 * h));
end;


...
{ Вызывать вот так: }
start := GetTime;
  ... { Здесь - алгоритм, время выполнения которого надо замерить }
WriteLn('Время выполнения = ', GetTime - start);
...
```

Поскольку способ самый простой, у него много недостатков, в частности, результат будет неверным, если во время работы алгоритма (после первого вызова функции GetTime, но перед вторым) сменится дата Тогда данный способ может показать и отрицательное время, но в любом случае оно будет неправильным (даже если будет правдоподобным).

Использование GetTickCount() (32-битные компиляторы)

При использовании 32-битных компиляторов для определения времени выполнения можно использовать функцию GetTickCount(), возвращающую количество миллисекунд, прошедших с момента старта системы. Разумеется, в случае использования GetTickCount проблема с переменой даты не возникает, но тут нас подстерегает другая опасность: каждые 49,8 дней будет происходить целочисленное переполнение этого значения, так что есть небольшая опасность, что если вызвать функцию незадолго до этого события один раз, а затем второй раз - после, то получится неизвестно что, а именно: число, близкое к -2*MaxInt.

Использование:

```
uses Windows;
var Duration: Cardinal;

begin
    Duration := GetTickCount();

     // замеряемый блок

    Duration := GetTickCount() - Duration;
    WriteLn('Время выполнения = ', Duration);
end.
```

Этот метод также не отличается особой точностью (предельно малые интервалы времени при использовании GetTickCount() составляют 10 .. 16 мс), но зато он очень прост, и если **очень высокая** точность не нужна - очень даже имеет право на существование.

Использование QueryPerformanceCounter() (32-битные компиляторы)

Точность этого метода, естественно, гораздо выше, чем точность GetTime, и чем точность метода подсчета тиков таймера. Прежде всего - потому, что эта процедура была создана специально в качестве средства для точной работы со временем...

Используется QueryPerformanceCounter вот так:

```
Uses
    Windows, SysUtils;
Var
    t1, t2, Res: int64;
     bOk: BOOL;

Procedure StartTimer;
Begin
    t1 := 0; t2 := 0; Res := 0;
    bOK := QueryPerformanceFrequency(Res);
    If bOK Then QueryPerformanceCounter(t1);
end;

Procedure StopTimer;
Begin
  If bOK Then QueryPerformanceCounter(t2);
End;

{ Время выполнения этой процедуры будет засекаться }
Procedure SomeProc;
Var
  i: Integer;
  a, b, T: Integer;
Begin
  For i := 1 To 10000 Do Begin
    a := 100; b := 250;
    T := a; a := b; b := T;
  End;
End;

Procedure CheckExecutionTime;
Begin
  StartTimer;
  SomeProc; // Запускаем тестируемую процедуру
  StopTimer;

  If bOK Then Writeln('Execution time: ' + Format('%g sec.',[(t2 - t1) / Res]));
End;

{ Изменяем приоритет потока на TIME_CRITICAL для увеличения точности }
Procedure Check_Time;
Var tp, pc: DWORD;
Begin
  tp := GetThreadPriority(GetCurrentThread());
  pc := GetPriorityClass(GetCurrentProcess());
  SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);
  SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS);
```

```
  Try
    CheckExecutionTime()
  Finally
    SetThreadPriority(GetCurrentThread(), tp);
    SetPriorityClass(GetCurrentProcess(), pc);
  End
End;

{ Основная программа }
begin
  check_time();
end.
```

Порядок выполнения работы

В файле 2.pas дано решение задачи с использованием циклов.
а) Провести анализ программы с точки зрения стиля.
б) Оптимизировать программу. Обосновать выбранные методы оптимизации.

```
uses crt;
      const
            n=10;
            a:array [1..n] of integer = (-9,11,0,38,15,-45,4,-7,4,6);
            b:array [1..n] of integer = (6,3,8,3,7,2,3,14,5,6);
            c:array [1..n] of integer = (5,6,-85,21,6,33,12,-8,43,33);
      var
            mas:array [1..n,1..n] of real;
            i,j:byte;
begin
clrscr;
for i:=1 to n do writeln(a[i],b[i]:8,c[i]:8);
for j:=1 to n do
      begin
            for j:= 1 to n do
                  begin
                        mas[i,j]:=sqr(a[i])-sqrt(b[i])*abs(c[j]);
                        write(mas[i,j]:7:1);
                  end;
            writeln;
      end;
end.
```

Лабораторная работа №2

Тема: Использование средств отладки программ (2 часа)

Цель работы: применить методы отладки программы

Теоретическая часть

Отла́дка — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки. Чтобы понять, где возникла ошибка, приходится :

узнавать текущие значения переменных;

выяснять, по какому пути выполнялась программа.

Существуют две взаимодополняющие технологии отладки.

Использование отладчиков — программ, которые включают в себя пользовательский интерфейс для пошагового выполнения программы: оператор за оператором, функция за функцией, с остановками на некоторых строках исходного кода или при достижении

определённого условия.

Вывод текущего состояния программы с помощью расположенных в критических точках программы операторов вывода — на экран, принтер, громкоговоритель или в файл. Вывод отладочных сведений в файл называется журналированием.

Отладчик представляет из себя программный инструмент, позволяющий программисту наблюдать за выполнением исследуемой программы, останавливать и перезапускать её, прогонять в замедленном темпе, изменять значения в памяти и даже, в некоторых случаях, возвращать назад по времени.

Также полезными инструментами в руках программиста могут оказаться:

Профилировщики. Они позволят определить сколько времени выполняется тот или иной участок кода, а анализ покрытия позволит выявить неисполняемые участки кода.

API логгеры позволяют программисту отследить взаимодействие программы и Windows API при помощи записи сообщений Windows в лог.

Дизассемблеры позволят программисту посмотреть ассемблерный код исполняемого файла

Снифферы помогут программисту проследить сетевой трафик генерируемой программой

Снифферы аппаратных интерфейсов позволят увидеть данные которыми обменивается система и устройство.

Логи системы.

Использование языков программирования высокого уровня, таких как Java, обычно упрощает отладку, поскольку содержат такие средства как обработка исключений, сильно облегчающие поиск источника проблемы. В некоторых низкоуровневых языках, таких как ассемблер, ошибки могут приводить к незаметным проблемам — например, повреждениям памяти или утечкам памяти, и бывает довольно трудно определить что стало первоначальной причиной ошибки. В этих случаях, могут потребоваться изощрённые приёмы и средства отладки.

Порядок выполнения работы

В файле 2.pas дано решение следующей задачи. Пусть дан файл целых чисел. Определите, сколько раз в нем повторяется максимальное значение.

1) Найти ошибки в программе.
2) Провести анализ программы с точки зрения стиля.
3) Составить по предложенному примеру удобочитаемую программу.
4) Привести характеристику выбранного стиля.

```
program zadan2;
uses crt;
var f: file of fi;
max,k,x:integer;
begin
clrscr;
reset(f);
seek(f,1);read(f,x);
max:=x;
while not eof(f) do begin
read(f,x);
if max<x then max:=x;
end;
k:=0;
while not eof(f) do begin
```

```
read(f,x);
if x=max then k:=k+1;
end;
close(f);
writeln('Максимальное значение повторяется ',k,' раз');
readkey;
end;
```

Лабораторная работа №3

Тема: Тестирование программ методами «белого ящика»: покрытия операторов, покрытия решений, покрытия условий (2 часа)

Цель работы: изучить методы тестирования программы, формализованные описания результатов тестирования и стандарты по составлению схем программ.

Теоретическая часть

Тестирование программного обеспечения включает в себя целый комплекс действий, аналогичных последовательности процессов разработки программного обеспечения. В него входят:

- постановка задачи для теста;
- проектирование теста;
- написание тестов;
- тестирование тестов;
- выполнение тестов;
- изучение результатов тестирования.

Наиболее важным является проектирование тестов. Существуют разные подходы к проектированию тестов.

Первый состоит в том, что тесты проектируются на основе внешних спецификаций программ и модулей либо спецификаций сопряжения модуля с другими модулями, программа при этом рассматривается как «черный ящик». Смысл теста заключается в том, чтобы проверить, соответствует ли программа внешним спецификациям. При этом содержание модуля не имеет значения. Такой подход получил название — стратегия «черного ящика».

Второй подход — стратегия «белого ящика», основан на анализе логики программы. При таком подходе тестирование заключается в проверке каждого пути, каждой ветви алгоритма. При этом внешняя спецификация во внимание не принимается.

Ни один из этих подходов не является оптимальным. Реализация тестирования методом «черного ящика» сводится к проверке всех возможных комбинаций входных данных. Невозможно протестировать программу, подавая на вход бесконечное множество значений, поэтому ограничиваются определенным набором данных. При этом исходят из максимальной отдачи теста по сравнению с затратами на его создание. Она измеряется вероятностью того, что тест выявит ошибки, если они имеются в программе. Затраты измеряются временем и стоимостью подготовки, выполнения и проверки результатов теста.

Тестирование методом «белого ящика» также не дает 100%-ной гарантии того, что модуль не содержит ошибок. Даже если предположить, что выполнены тесты для всех ветвей алгоритма, нельзя с полной уверенностью утверждать, что программа соответствует ее спецификациям. Например, если требовалось написать программу для вычисления кубического корня, а программа фактически вычисляет корень квадратный, то реализация будет совершенно неправильной, даже если проверить все пути. Вторая проблема —- отсутствующие пути. Если программа реализует спецификации не полностью (например, отсутствует такая специализированная функция, как проверка на отрицательное значение входных данных программы вычисления квадратного корня), никакое тестирование существующих путей не выявит такой ошибки. И наконец, проблема зависимости результатов тестирования от входных данных. Одни данные будут давать правильные результаты, а другие нет.

Таким образом, полное тестирование программы невозможно, т. е. никакое тестирова-

ние не гарантирует полное отсутствие ошибок в программе. Поэтому необходимо проектировать тесты таким образом, чтобы увеличить вероятность обнаружения ошибки в программе.

Стратегия «белого ящика»

Рассмотрим следующие методы тестирования по принципу «белого ящика»:

- покрытие операторов;
- покрытие решений;
- покрытие условий.

Метод покрытия операторов

Целью этого метода тестирования является выполнение каждого оператора программы хотя бы один раз.

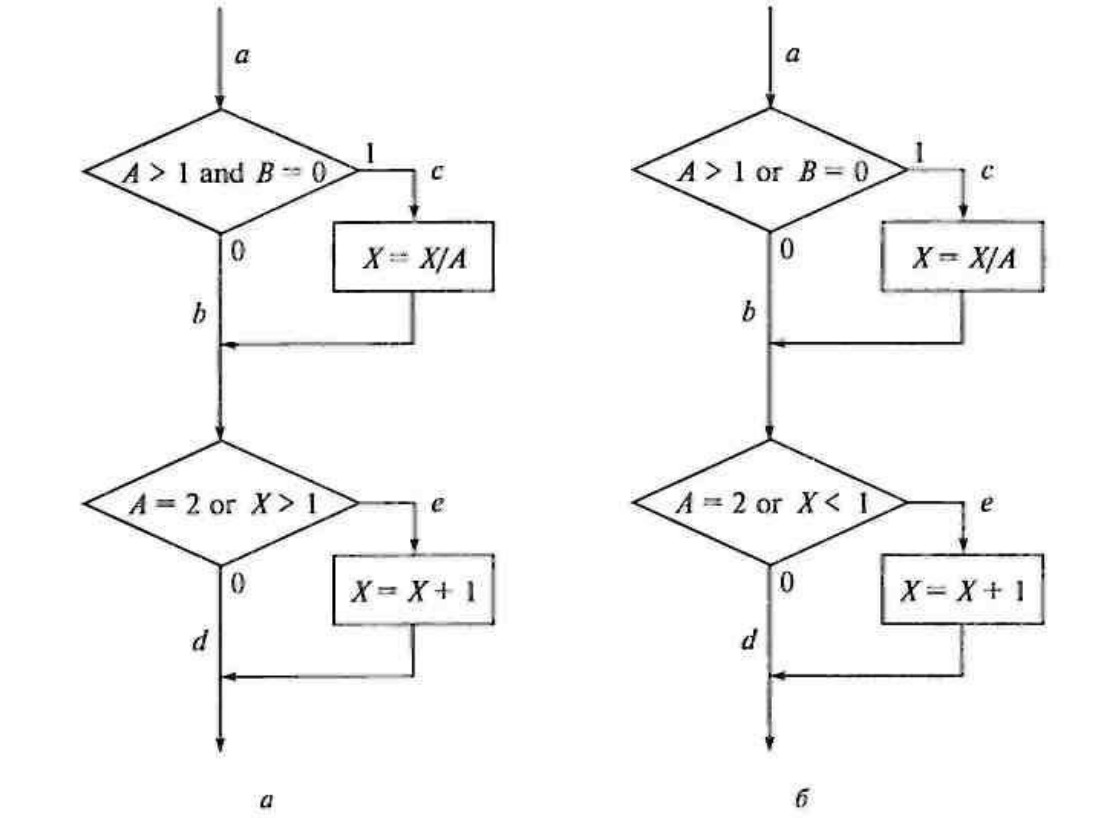
Пример.

Если для тестирования задать значения переменных $A = 2$, $B=0$, $X=3$, будет реализован путь *ace,* т. е. каждый оператор программы выполнится один раз (рис. 1, *а).* Но если внести в алгоритм ошибки — заменить в первом условии *and* на *or,* а во втором $X> 1$ на $X< 1$ (рис. 1, *б),* ни одна ошибка не будет обнаружена (табл. 1). Кроме того, путь *abd* вообще не будет охвачен тестом, и если в нем есть ошибка, она также не будет обнаружена. В табл. 12 ожидаемый результат определяется по блок-схеме на рис. 1, *а,* а фактический — по рис. 1, *б.*

Как видно из этой таблицы, ни одна из внесенных в алгоритм ошибок не будет обнаружена.

Таблица 1 Результат тестирования методом покрытия операторов

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A = 2,\ B = 0,\ X = 3$	$X = 2{,}5$	$X = 2{,}5$	Неуспешно



а *б*

Рис. 1. Пример алгоритма программы: *а* — правильный; *б* — с ошибкой

Метод покрытия решений (покрытия переходов)

Согласно методу покрытия решений каждое направление перехода должно быть реали-

зовано, по крайней мере, один раз. Этот метод включает в себя критерий покрытия операторов, так как при выполнении всех направлений переходов выполнятся все операторы, находящиеся на этих направлениях.

Для программы, приведенной на рис. 1, покрытие решений может быть выполнено двумя тестами, покрывающими пути *{ace, abd},* либо *{acd, abe).* Для этого выберем следующие исходные данные: *{A = 3, B = 0, X=3}* — в первом случае и *{A* = 2, *B=1, X*= 1} — во втором. Однако путь, где А не меняется, будет проверен с вероятностью 50 %: если во втором условии вместо условия *X*> 1 записано *X*< 1, то ошибка не будет обнаружена двумя тестами. Результаты тестирования приведены в табл. 2.

Таблица 2. Результат тестирования методом покрытия решений

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A=3,\ B=0,\ X=3$	$X=1$	$X=1$	Неуспешно
$A=2,\ B=1,\ X=1$	$X=2$	$X=1{,}5$	Успешно

Метод покрытия условий

Этот метод может дать лучшие результаты по сравнению с предыдущими. В соответствии с методом покрытия условий записывается число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз.

В рассматриваемом примере имеем четыре условия: *{A>1, B* = 0}, *{A* = 2, *X*> 1}. Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где *A*> 1, *A*< 1, *B=0* и *B*≠ 0 в точке *a* и *A* = 2, *A* ≠ *2, X*> 1 и *X*< 1 в точке *b.* Тесты, удовлетворяющие критерию покрытия условий (табл.3), и соответствующие им пути:

а) *A* = 2, *B=0, X=4 ace;*

б)*A= 1, B= 1, X = 0 abd.*

Таблица 3. Результаты тестирования методом покрытия условий

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A=2,\ B=0,\ X=4$	$X=3$	$X=3$	Неуспешно
$A=1,\ B=1,\ X=0$	$X=0$	$X=1$	Успешно

Контрольные вопросы

1. Какие действия включает в себя тестирование программного обеспечения?
2. Опишите подходы к проектированию тестов.
3. Дайте сравнительную характеристику известных вам подходов к проектированию тестов.
4. Проиллюстрируйте на примере суть метода
   а) покрытия операторов;
   б) покрытия решений;
   в) покрытия условий.

Порядок выполнения работы

1. Спроектировать тесты по принципу «белого ящика» для алгоритма, разработанного в практической работе № 2.
2. Выбрать несколько алгоритмов для тестирования и обозначить буквами или цифрами ветви этих алгоритмов.
3. Выписать пути алгоритма, которые должны быть проверены тестами для выбранного метода тестирования.

4. Записать тесты, которые позволят пройти по путям алгоритма.

5. Протестировать разработанную вами программу. Результаты оформить в виде таблицы.

Лабораторная работа №4

Тема: Тестирование программ методами «белого ящика»: покрытия решений/условий, комбинаторного покрытия условий (2 часа)

Цель работы: изучить методы тестирования программы, формализованные описания результатов тестирования и стандарты по составлению схем программ.

Теоретическая часть

Метод покрытия решений/условий

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

Недостатки метода:

- не всегда можно проверить все условия;
- невозможно проверить условия, которые скрыты другими условиями;
- • недостаточная чувствительность к ошибкам в логических выражениях.

Так, в рассматриваемом примере два теста метода покрытия условий

а) *A = 2, B=0, X=4 ace;*

б) *A=1, B = 1, X=0 abd* отвечают и критерию покрытия решений/условий. Это является следствием того, что одни условия приведенных решений скрывают другие условия в этих решениях. Так, если условие $A > 1$ будет ложным, транслятор может не проверять условия 5=0, поскольку при любом результате условия $5 = 0$ результат решения *((A>* 1)&(5 = 0)) примет значение *ложь*. То есть в варианте на рис. 1 не все результаты всех условий выполнятся в процессе тестирования.

Рассмотрим реализацию того же примера на рис. 2. Наиболее полное покрытие тестами в этом случае осуществляется так, чтобы выполнялись все возможные результаты каждого простого решения. Для этого нужно покрыть пути *aceg* (тест *A = 2, B=0, X=4*), *acdfh*

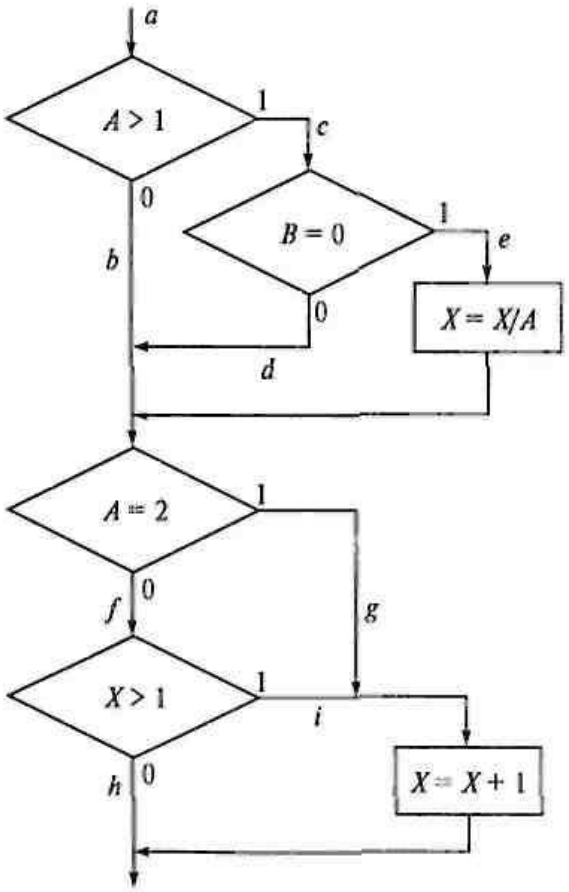


Рис. 2. Пример алгоритма программы

(тест $A = 3, B = 1, X = 0$), *abfh* (тест $A = 0, B = 0, X=0$), *abfi* (тест $A= 0, B = 0, X = 2$).

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

Метод комбинаторного покрытия условий

Критерий комбинаторного покрытия условий удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.

Этот метод требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении выполнялись по крайней мере один раз. По этому критерию в рассматриваемом примере должны быть покрыты тестами следующие восемь комбинаций:

1. $A > 1, 5=0$.
2. $A > 1, 5\neq 0$.
3. $A < 1, 5=0$.
4. $A < 1, 5\neq 0$.
5. $A = 2, X > 1$.
6. $A = 2, X < 1$
7. $A \neq 2, X > 1$.
8. $A \neq 2, X < 1$.

Для того чтобы протестировать эти комбинации, необязательно использовать все 8 тестов. Фактически они могут быть покрыты четырьмя тестами (табл. 3):

- $A = 2, 5 = 0, X=4$ {покрывает 1, 5};
- $A = 2, 5 = 1, X=1$ {покрывает 2, 6};
- $A = 0,5, B= 0, X= 2$ {покрывает 3, 7};
- $A= 1, B = 0, X= 1$ {покрывает 4, 8}.

Таблица 3. Результаты тестирования методом комбинаторного покрытия условий

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A = 2, B = 0, X = 4$	$X = 3$	$X = 3$	Неуспешно
$A = 2, B = 1, X = 1$	$X = 2$	$X = 1,5$	Успешно
$A = 0,5, B = 0, X = 2$	$X = 3$	$X = 4$	Успешно
$A = 1, B = 0, X = 1$	$X = 1$	$X = 1$	Неуспешно

Порядок выполнения работы

1. Спроектировать тесты по принципу «белого ящика» для алгоритма, разработанного в практической работе № 2.
2. Выбрать несколько алгоритмов для тестирования и обозначить буквами или цифрами ветви этих алгоритмов.
3. Выписать пути алгоритма, которые должны быть проверены тестами для выбранного метода тестирования.
4. Записать тесты, которые позволят пройти по путям алгоритма.
5. Протестировать разработанную вами программу. Результаты оформить в виде таблицы.
6. Сделать вывод об эффективности видов тестов.

Лабораторная работа №5

Тема: Оформление программной документации. Стадия «Техническое задание» (4 часа)

Цель работы: ознакомиться с правилами написания технического задания.

Теоретическая часть

Техническое задание представляет собой документ, в котором сформулированы основные цели разработки, требования к программному продукту, определены сроки и этапы разработки и регламентирован процесс приемо-сдаточных испытаний. В разработке технического задания участвуют как представители заказчика, так и представители исполнителя. В основе этого документа лежат исходные требования заказчика, анализ передовых достижений техники, результаты выполнения научно-исследовательских работ, предпроектных исследований, научного прогнозирования и т. п.

Порядок разработки технического задания

Прежде всего, устанавливают набор выполняемых функций, а также перечень и характеристики исходных данных. Затем определяют перечень результатов, их характеристики и способы представления. Далее уточняют среду функционирования программного обеспечения: конкретную комплектацию и параметры технических средств, версию используемой операционной системы и, возможно, версии и параметры другого установленного программного обеспечения, с которым предстоит взаимодействовать будущему программному продукту.

В случаях, когда разрабатываемое программное обеспечение собирает и хранит некоторую информацию или включается в управление каким-либо техническим процессом, необходимо также четко регламентировать действия программы в случае сбоев оборудования и энергоснабжения.

1. Общие положения

1.1. Техническое задание оформляют в соответствии с ГОСТ 19.106—78 на листах формата А4 и А3 по ГОСТ 2.301—68, как правило, без заполнения полей листа. Номера листов (страниц) проставляют в верхней части листа над текстом.

1.2. Лист утверждения и титульный лист оформляют в соответствии с ГОСТ 19.104—78. Информационную часть (аннотацию и содержание), лист регистрации изменений допускается в документ не включать.

1.3. Для внесения изменений и дополнений в техническое задние на последующих стадиях разработки программы или программного изделия выпускают дополнение к нему. Согласование и утверждение дополнения к техническому заданию проводят в том же порядке, который установлен для технического задания.

1.4. Техническое задание должно содержать следующие разделы:

- введение;
- наименование и область применения;
- основание для разработки;
- назначение разработки;
- технические требования к программе или программному изделию;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки;
- приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них. При необходимости допускается в техническое задание включать приложения.

2. Содержание разделов

2.1. Введение должно включать краткую характеристику области применения программы или программного продукта, а также объекта (например, системы), в котором предполагается их использовать. Основное назначение введения — продемонстрировать актуальность данной разработки и показать, какое место эта разработка занимает в ряду подобных.

2.2. В разделе «Наименование и область применения» указывают наименование, краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие.

2.3. В разделе «Основание для разработки» должны быть указаны:

• документ (документы), на основании которых ведется разработка. Таким документом может служить план, приказ, договор и т. п.

• организация, утвердившая этот документ, и дата его утверждения;

• наименование и (или) условное обозначение темы разработки.

2.4. В разделе «Назначение разработки» должно быть указано функциональное и эксплуатационное назначение программы или программного изделия.

2.5. Раздел «Технические требования к программе или программному изделию» должен содержать следующие подразделы:

• требования к функциональным характеристикам;
• требования к надежности;
• условия эксплуатации;
• требования к составу и параметрам технических средств;
• требования к информационной и программной совместимости;
• требования к маркировке и упаковке;
• требования к транспортированию и хранению;
• специальные требования.

2.5.1. В подразделе «Требования к функциональным характеристикам» должны быть указаны требования к составу выполняемых функций, организации входных и выходных данных, временным характеристикам и т. п.

2.5.2. В подразделе «Требования к надежности» должны быть указаны требования к обеспечению надежного функционирования (обеспечение устойчивого функционирования, контроль входной и выходной информации, время восстановления после отказа и т. п.).

2.5.3. В подразделе «Условия эксплуатации» должны быть указаны условия эксплуатации (температура окружающего воздуха, относительная влажность и т. п. для выбранных типов носителей данных), при которых должны обеспечиваться заданные характеристики, а также вид обслуживания, необходимое количество и квалификация персонала.

2.5.4. В подразделе «Требования к составу и параметрам технических средств» указывают необходимый состав технических средств с указанием их технических характеристик.

2.5.5. В подразделе «Требования к информационной и программной совместимости» должны быть указаны требования к информационным структурам на входе и выходе и методам решения, исходным кодам, языкам программирования. При необходимости должна обеспечиваться защита информации и программ.

2.5.6. В подразделе «Требования к маркировке и упаковке» в общем случае указывают требования к маркировке программного изделия, варианты и способы упаковки.

2.5.7. В подразделе «Требования к транспортированию и хранению» должны быть указаны для программного изделия условия транспортирования, места хранения, условия хранения, условия складирования, сроки хранения в различных условиях.

2.5.8. В разделе «Технико-экономические показатели» должны быть указаны: ориентировочная экономическая эффективность, предполагаемая годовая потребность, экономические преимущества разработки по сравнению с лучшими отечественными и зарубежными образцами или аналогами.

2.6. В разделе «Стадии и этапы разработки» устанавливают необходимые стадии разработки, этапы и содержание работ (перечень программных документов, которые должны быть разработаны, согласованы и утверждены), а также, как правило, сроки разработки и определяют исполнителей.

2.7. В разделе «Порядок контроля и приемки» должны быть указаны виды испытаний и общие требования к приемке работы.

2.8. В приложениях к техническому заданию при необходимости приводят:

• перечень научно-исследовательских и других работ, обосновывающих разработку;
• схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие документы,

которые могут быть использованы при разработке;

- другие источники разработки.

В случаях, если какие-либо требования, предусмотренные техническим заданием, заказчик не предъявляет, следует в соответствующем месте указать «Требования не предъявляются».

Порядок выполнения работы

1. Разработать техническое задание на программный продукт.
2. Оформить работу в соответствии с ГОСТ 19.106—78. При оформлении использовать MS Office.

Лабораторная работа №6

Тема: Оформление программной документации. Стадия «Эскизный проект» (4 часа).

Цель работы: научиться создавать формальные модели и на их основе определять спецификации разрабатываемого программного продукта.

Теоретическая часть

Разработка программного обеспечения начинается с анализа требований к нему. В результате анализа получают спецификации разрабатываемого программного обеспечения, строят общую модель его взаимодействия с пользователем или другими программами и конкретизируют его основные функции.

При структурном подходе к программированию на этапе анализа и определения спецификаций разрабатывают три типа моделей: модели функций, модели данных и модели потоков данных. Поскольку разные модели описывают проектируемое программное обеспечение с разных сторон, рекомендуется использовать сразу несколько моделей, разрабатываемых в виде диаграмм, и пояснять их текстовыми описаниями, словарями и т. п. Структурный анализ предполагает использование следующих видов моделей:

- диаграмм потоков данных (DFD — Data Flow Diagrams), описывающих взаимодействие источников и потребителей информации через процессы, которые должны быть реализованы в системе;
- диаграмм «сущность—связь» (ERD — Entity-Relationship Diagrams), описывающих базы данных разрабатываемой системы;
- диаграмм переходов состояний (STD — State Transition Diagrams), характеризующих поведение системы во времени;
- функциональных диаграмм (методика SADT);
- спецификаций процессов;
- словаря терминов.

Спецификации процессов обычно представляют в виде краткого текстового описания, схем алгоритмов, псевдокодов, Flow-форм или диаграмм Насси — Шнейдермана

Словарь терминов представляет собой краткое описание основных понятий, используемых при составлении спецификаций. Он должен включать определение основных понятий предметной области, описание структур элементов данных, их типов и форматов, а также всех сокращений и условных обозначений.

С помощью *диаграмм переходов состояний* можно моделировать последующее функционирование системы на основе ее предыдущего и текущего функционирования. Моделируемая система в любой заданный момент времени находится точно в одном из конечного множества состояний. С течением времени она может изменить свое состояние, при этом переходы между состояниями должны быть точно определены.

Функциональные диаграммы отражают взаимосвязи функций разрабатываемого программного обеспечения.

Они создаются на ранних этапах проектирования систем, для того чтобы помочь проектировщику выявить основные функции и составные части проектируемой системы и, по возможности, обнаружить и устранить существенные ошибки. Для создания функциональ-

ных диаграмм предлагается использовать методологию SADT (см. разд. 3.5.4).

Для описания потоков информации в системе применяются *диаграммы потоков данных* (DFD — Data flow diagrams). DFD позволяет описать требуемое поведение системы в виде совокупности процессов, взаимодействующих посредством связывающих их потоков данных. DFD показывает, как каждый из процессов преобразует свои входные потоки данных в выходные потоки данных и как процессы взаимодействуют между собой (см. разд. 3.5.5).

Диаграмма сущность—связь — инструмент разработки моделей данных, обеспечивающий стандартный способ определения данных и отношений между ними. Она включает *сущности* и взаимосвязи, отражающие основные бизнес-правила предметной области. Такая диаграмма не слишком детализирована, в нее включаются основные *сущности* и *связи* между ними, которые удовлетворяют требованиям, предъявляемым к ИС (см. разд. 3.5.6).

Порядок выполнения работы

1. На основе технического задания из практической работы № 1 выполнить анализ функциональных и эксплуатационных требований к программному продукту.
2. Определить основные технические решения (выбор языка программирования, структура программного продукта, состав функций ПП, режимы функционирования) и занести результаты в документ, называемый «Эскизным проектом»
3. Определить диаграммы потоков данных для решаемой задачи.
4. Определить диаграммы «сущностьсвязь», если программный продукт содержит базу данных.
5. Определить функциональные диаграммы.
6. Определить диаграммы переходов состояний.
7. Определить спецификации процессов.
8. Добавить словарь терминов.
9. Оформить результаты, используя MS Office в виде эскизного проекта.

Лабораторная работа №7

Тема: Оформление программной документации. Стадия «Технический проект»» (4 часа).

Цель работы: изучить вопросы проектирования программного продукта.

.

Теоретическая часть

Технический проект— образ намеченного к созданию объекта, представленный в виде его описания, схем, чертежей, расчетов, обоснований, числовых показателей.

Цель технического проекта — определение основных методов, используемых при создании информационной системы, и окончательное определение ее сметной стоимости.

Техническое проектирование подсистем осуществляется в соответствии с утвержденным техническим заданием.

Технический проект программной системы подробно описывает:

- выполняемые функции и варианты их использования;
- соответствующие им документы;
- структуры обрабатываемых баз данных;
- взаимосвязи данных;
- алгоритмы их обработки.

Технический проект должен включать данные об объемах и интенсивности потоков обрабатываемой информации, количестве пользователей программной системы, характеристиках оборудования и программного обеспечения, взаимодействующего с проектируемым программным продуктом.

При разработке технического проекта оформляются:

- ведомость технического проекта. Общая информация по проекту;
- пояснительная записка к техническому проекту. Вводная информация, позволяющая ее потребителю быстро освоить данные по конкретному проекту;

• описание систем классификации и кодирования;

• перечень входных данных (документов). Перечень информации, которая используется как входящий поток и служит источником накопления;

• перечень выходных данных (документов). Перечень информации, которая используется для анализа накопленных данных;

• описание используемого программного обеспечения. Перечень программного обеспечения и СУБД, которые планируется использовать для создания информационной системы;

• описание используемых технических средств. Перечень аппаратных средств, на которых планируется работа проектируемого программного продукта;

• проектная оценка надежности системы. Экспертная оценка надежности с выявлением наиболее благополучных участков программной системы и ее узких мест;

• ведомость оборудования и материалов. Перечень оборудования и материалов, которые потребуются в ходе реализации проекта.

Структурной называют схему, отражающую состав и взаимодействие по управлению частями разрабатываемого программного обеспечения. Структурная схема определяется архитектурой разрабатываемого ПО

Функциональная схема — это схема взаимодействия компонентов программного обеспечения с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств.

Метод пошаговой детализации реализует нисходящий подход к программированию и предполагает пошаговую разработку алгоритма.

Методика *структурных карт* используется на этапе проектирования ПО для того, чтобы продемонстрировать, каким образом программный продукт выполняет системные требования. Структурные карты Константайна предназначены для описания отношений между модулями.

Техника структурных карт Джексона основана на методе структурного программирования Джексона, который выявляет соответствие между структурой потоков данных и структурой программы. Основное внимание в методе сконцентрировано на соответствии входных и выходных потоков данных

Порядок выполнения работы

1. На основе технического задания из лабораторной работы № 5 и спецификаций из лабораторной работы № 6 разработать уточненные алгоритмы программ, составляющих заданный программный модуль. Использовать метод пошаговой детализации.
2. На основе уточненных и доработанных алгоритмов разработать структурную схему программного продукта.
3. Разработать функциональную схему программного продукта.
4. Оформить результаты, используя MS Office.

Лабораторная работа №8

Тема: Оформление программной документации. Стадия «Реализация»» (4 часа).

Цель работы: разработать программный продукт в соответствии с заданным вариантом.

Теоретическая часть

Важным этапом разработки программного продукта является составление программной документации. Жизненный цикл программного обеспечения содержит специальный процесс, посвященный этому вопросу. На каждый программный продукт должны составляться два типа документации — для разработчиков и для различных групп пользователей. Программная документация пользователей должна содержать все необходимые сведения по эксплуатации ПО. Аналогично, документация разработчика должна содержать сведения, необходимые для разработки и сопровождения программного обеспечения.

Виды программных документов

Документирование программного обеспечения осуществляется в соответствии с Единой системой программной документации (ГОСТ I9.XXX). ГОСТ 19.ЮТ—77 содержит виды программных документов для программного обеспечения различных типов. В данном ГОСТе перечислены документы следующих типов:

- *спецификация* должна содержать перечень и краткое описание назначения всех файлов программного обеспечения, в том числе и файлов документации на него, и является обязательной для программных систем, а также их компонентов, имеющих самостоятельное применение;
- *ведомость держателей подлинников* (код вида документа — 05) должна содержать список предприятий, на которых хранятся подлинники программных документов. Необходимость этого документа определяется на этапе разработки и утверждения технического задания только для программного обеспечения со сложной архитектурой;
- *текст программы* (код вида документа — 12) должен содержать текст программы с необходимыми комментариями. Необходимость этого документа определяется на этапе разработки и утверждения технического задания;
- *описание программы* (код вида документа — 13) должно содержать сведения о логической структуре и функционировании программы. Необходимость данного документа также определяется на этапе разработки и утверждения технического задания;
- *ведомость эксплуатационных документов* (код вида документа — 20) должна содержать перечень эксплуатационных документов на программу, к которым относятся документы с кодами 30, 31, 32, 33, 34, 35, 46. Необходимость этого документа также определяется на этапе разработки и утверждения технического задания;
- *формуляр* (код вида документа — 30) должен содержать основные характеристики программного обеспечения, комплектность и сведения об эксплуатации программы;
- *описание применения* (код вида документа — 31) должно содержать сведения о назначении программного обеспечения, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств;
- *руководство системного программиста* (код вида документа — 32) должно содержать сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения;
- *руководство программиста* (код вида документа-ЗЗ) должно содержать сведения для эксплуатации программного обеспечения;
- *руководство оператора* (код вида документа — 34) содержит сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программы;
- *описание языка* (код вида документа — 35) — описание синтаксиса и семантики языка программы;
- *руководство по техническому обслуживанию* (код вида документа — 46) содержит сведения для применения программы при обслуживании технических средств.

Порядок выполнения работы

1. По результатам лабораторных работ № 5—7 написать код программ для решения поставленной задачи на языке программирования, выбранном на этапе эскизного проектирования.
2. Отладить программный модуль. Получить результаты работы.
3. Оформить документацию к разработанному программному обеспечению.

РАЗДЕЛ 7. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ

7.1. Методические рекомендации по выполнению самостоятельной работы

7.1.1. Методические рекомендации по составлению опорного конспекта

Основные требования, предъявляемые к содержанию и форме записи опорного конспекта.

1. Полнота изложения материала;
2. Последовательность и логичность в отражении темы;
3. Лаконичность записи: опорный конспект по объему должен составлять не более листа и воспроизводиться в устной форме за 5-7 минут;
4. Структурирование записей, т.е. изложение материала по пунктам в форме простого или сложного плана. При этом каждый блок должен выражать законченную мысль;
5. Расстановка акцентов, т.е. выделение ключевых слов, понятий с помощью рамок, шрифтов, различных цветов и графических приемов (столбик, диагональ и т.д.);
6. Наглядность;
7. Связь с материалами учебника, справочника и других видов учебной литературы.

Запишите название темы по предмету. Ознакомьтесь с необходимым материалом по тексту учебника, пособия, справочника и т.д. Выделите главное в изучаемом материале, составьте конспект в виде простых записей.

Выберите ключевые слова или понятия, отражающие суть изучаемой темы. В зависимости от цели составления опорного конспекта, изложение исходного текста может быть самым различным по форме, например: в виде слов, словосочетаний и предложений на уроках гуманитарного цикла; схем, таблиц и формул по физико-математическим дисциплинам. Также можно использовать рисунки и различные графические символы. Каждое из ключевых понятий должно воздействовать на читателя как опорный сигнал.

Продумайте способ «кодирования» знаний, выбрав для этого необходимые приемы.

Используйте прием сокращения слов, для экономии времени при составлении опорного конспекта. Обычно сокращаются слова, наиболее часто употребляемые на уроках, например: ОС (операционные системы), инф.(информация). Также вы можете использовать графические обозначения, отражающие суть излагаемого материала.

7.1.2. Методические рекомендации по подготовке доклада

Доклад - вид самостоятельной научно-исследовательской работы, где автор раскрывает суть исследуемой проблемы; приводит различные точки зрения, а также собственные взгляды на нее. Различают устный и письменный доклад (по содержанию, близкий к реферату).

В докладе соединяются три качества исследователя: умение провести исследование, умение преподнести результаты слушателям и квалифицированно ответить на вопросы.

Отличительной чертой доклада является научный, академический стиль. Академический стиль - это совершенно особый способ подачи текстового материала, наиболее подходящий для написания учебных и научных работ. Данный стиль определяет следующие нормы:

- предложения могут быть длинными и сложными;
- часто употребляются слова иностранного происхождения, различные термины;
- употребляются вводные конструкции типа “по всей видимости”, “на наш взгляд”;
- авторская позиция должна быть как можно менее выражена, то есть должны отсутствовать местоимения “я”, “моя (точка зрения)”;

Общая структура такого доклада может быть следующей:

1. Формулировка темы исследования (причем она должна быть не только актуальной, но и оригинальной, интересной по содержанию).

2. Актуальность исследования (чем интересно направление исследований, в чем заключается его важность, какие ученые работали в этой области, каким вопросам в данной теме уделялось недостаточное внимание, почему учащимся выбрана именно эта тема).

3. Цель работы (в общих чертах соответствует формулировке темы исследования и может уточнять ее).

4. Задачи исследования (конкретизируют цель работы, “раскладывая” ее на составляющие).

5. Гипотеза (научно обоснованное предположение о возможных результатах исследовательской работы. Формулируются в том случае, если работа носит экспериментальный характер).

6. Методика проведения исследования (подробное описание всех действий, связанных с получением результатов).

7. Результаты исследования. Краткое изложение новой информации, которую получил исследователь в процессе наблюдения или эксперимента. При изложении результатов желательно давать четкое и немногословное истолкование новым фактам. Полезно привести основные количественные показатели и продемонстрировать их на используемых в процессе доклада графиках и диаграммах.

8. Выводы исследования. Умозаключения, сформулированные в обобщенной, конспективной форме. Они кратко характеризуют основные полученные результаты и выявленные тенденции. Выводы желательно пронумеровать: обычно их не более 4 или 5.

Можно выделить следующие этапы работы над докладом.

1. Подбор и изучение основных источников по теме (как и при написании реферата, рекомендуется использовать не менее 8-10 источников).
2. Составление библиографии.
3. Обработка и систематизация материала. Подготовка выводов и обобщений.
4. Разработка плана доклада.
5. Написание.
6. Публичное выступление с результатами исследования.

Продолжительность выступления обычно не превышает 10-15 минут. Поэтому при подготовке доклада из текста работы отбирается самое главное. В докладе должно быть кратко отражено основное содержание всех глав и разделов исследовательской работы.

Для успешного выступления с докладом заучите значение всех терминов, которые употребляются в докладе.

При соблюдении этих правил у вас должен получиться интересный доклад, который, несомненно, будет высоко оценен преподавателем.

7.1.3. Методические рекомендации по реферированию темы

По форме студенческий реферат - это небольшое научное исследование. Хотя в реферате излагаются, преимущественно, идеи и их анализ, которые студент нашел в соответствующей литературе, тем не менее в его работе должны быть элементы самостоятельности и новизны. Большую помощь здесь окажут консультации, во время которых студент получает советы: где и как искать научную литературу; как оформить библиографию (список литературы по теме); как работать с научным трудом; каковы современные требования к научному описанию и каковы типовые части изложения.

При оценке реферата учитываются, в первую очередь, следующие критерии:

- соответствие требованиям программы курса;
- правильность выработанной студентом концепции описания проблемы;
- глубина проработки материала;
- правильность и полнота использования источников;
- оформление реферата.

Изучение теоретического материала по теме реферата не ставит перед вами задачу заучивания прочитанного, но, поскольку вы должны хорошо ориентироваться в специаль-

ной литературе, ему следует вести записи.

Объем реферата в среднем должен составлять 18-20 страниц рукописного или 12-15 страниц машинописного текста (не более 28-30 строк на странице, напечатанных через полтора интервала). Текст пишется (печатается) на одной стороне листа (шрифт 12 или 14).

В реферате обязательно должны быть представлены следующие структурные элементы:

1. План
2. Введение
3. Основная часть, содержащая несколько (3-4) разделов
4. Заключение
5. Библиография

Во Введении (объем не менее двух страниц) осуществляется постановка проблемы: объясняется выбор темы, ее значимость и актуальность. Кроме того, определяются цель и задачи реферата.

В Основной части излагается содержательная сторона реферата. Каждый из разделов, входящих в нее, должен быть озаглавлен и раскрывать одну из сторон рассматриваемого вопроса. Разделы по объему равноценны (3-5 страниц) и заканчиваются четкими выводами. Изложение материала должно быть четким, последовательным и соответствовать плану.

Методы описания материала могут быть различны. Кратко охарактеризуем основные из них.

Индуктивный метод - изложение материала от частного к общему. Студент начинает описание с конкретного факта, а затем делает обобщения и выводы.

Дедуктивный метод - изложение материала от общего к частному: сначала выдвигаются какие-то проблемы, а потом на конкретных примерах, фактах разъясняется их смысл.

Метод аналогии - сопоставление различных явлений, событий, фактов.

Концентрический метод - расположение материала вокруг главной проблемы, поднимаемой в реферате. Студент переходит от общего рассмотрения центрального вопроса к белее конкретному и углубленному анализу.

Ступенчатый метод - последовательное изложение одного вопроса за другим. Рассмотрев какую-либо проблему, студент уже больше не возвращается к ней.

Исторический метод - изложение материала в хронологической последовательности, описание и анализ изменений, которые произошли в том или ином предмете, в понимании того или иного явления в течением времени.

Использование различных методов изложения материала в одном и том же реферате позволяет сделать структуру основной части более оригинальной, нестандартной.

Вместе с тем, каким бы методом не пользовался студент, он должен помнить, что его работа без хорошо продуманной системы логических доводов, без аргументации будет неубедительной. Значимость, вес, сила аргументов должны нарастать постепенно, самые сильные доказательства в пользу определенной позиции лучше использовать в конце рассуждения.

7.1.4. Методические рекомендации по составлению аналитического обзора

Аналитический обзор - сокращенное изложение содержания первичных документов с основными фактическими сведениями и выводами. Аналитические обзоры составляются на основании книг, статей, газетных и журнальных публикаций и других источников информации. Главное требование, предъявляемое к аналитическому обзору, звучит так: вся информация должна быть представлена в сжатом и систематизированном виде.

Работа над аналитическим обзором начинается после того, как изучена литература и собран фактический материал. Первым ее шагом является составление плана, в котором

определяется последовательность изложения материала. План помогает лучше продумать структуру аналитического обзора, определить, какие разделы оказались перегруженными материалом, где его недостаточно, какие вопросы следует опустить и т.д. Составление плана помогает избежать ошибок в построении текста.

Хороший аналитический обзор должен содержать ответы на следующие вопросы: кто совершал, что, где, когда и с какой целью совершалось. В нем должно содержаться как можно больше конкретной информации, имеющейся в исходных информационных материалах.

7.1.5. Методические рекомендации по работе с литературой

Работая с книгой, следует прежде всего определенным образом настроить себя, дать себе соответствующую установку. Студент может поставить перед собой задачу изучить по книге тот или иной вопрос, который предстоит освещать в реферате; критически проанализировать содержание книги; проверить, совпадает ли его оценка с мнением автора, других авторитетных лиц; выбрать для реферата наиболее яркие факты, примеры, интересные положения и т.д. Подобные установки помогут студенту более целенаправленно работать с книгой и прежде всего определить вид чтения: сплошное, выборочное, комбинированное. При сплошном чтении книга прочитывается полностью, от начала до конца, без каких-либо пропусков. Иногда для разрабатываемой темы достаточно изучить не всю книгу, а лишь отдельные ее разделы, главы, параграфы. Такое чтение называется выборочным. Комбинированное чтение - это сплошное чтение отдельных частей и выборочное других.

Работу над книгой следует начинать с предварительного знакомства с ней. Сначала читается титульный лист книги. Нередко на титульном листе указывается классификационная характеристика книги (учебник, учебное пособие, справочное пособие, словарь-справочник и др.), позволяющая определить ее назначение. Необходимо обратить внимание и на год издания книги. На титульном листе указывается также наименование издательства и место издания книги. Но на этом не заканчивается предварительное знакомство с книгой. Следует просмотреть оглавление, дающее представление об основных вопросах, которые в ней затрагиваются, обратить внимание на рисунки, схемы, таблицы.

Ознакомиться с книгой помогает и аннотация, которая помещена на обороте титульного листа или в конце книги. В ней кратко рассказывается о содержании книги., говорится о ее назначении, даются сведения об авторе и т.п.

Если в книге есть предисловие и послесловие, рекомендуется прочитать их. В предисловии рассказывается история написания книги, передается ее краткое содержание, характеризуются основные проблемы. В послесловии автор подводит итоги изложенного, кратко формулирует или повторяет главные положения работы.

7.2. Методические рекомендации по организации и методическому сопровождению самостоятельной работы студентов

Одной из важнейших стратегических задач современной профессиональной школы является формирование профессиональной компетентности будущих специалистов. Квалификационные характеристики по всем специальностям среднего профессионального образования новых образовательных стандартов третьего поколения содержат такие требования, как умение осуществлять поиск, анализ и оценку информации. Одной из форм, помогающих формированию этого умения, являются продуманные и систематизированные, логически и целенаправленно разработанные задания и упражнения для самостоятельной работы студентов, в которых перед ними последовательно выдвигаются познавательные задачи, решая которые они осознанно и активно усваивают знания и учатся творчески применять их в новых условиях.

Целевые направления самостоятельной работы студентов:

1.Для овладения и углубления знаний:

- составление различных видов планов и тезисов пот тексту;
- конспектирование текста;
- составление тезауруса;
- ознакомление с нормативными документами;
- создание презентации.

2. Для закрепления знаний:
- работа с конспектом лекции;
- повторная работа с учебным материалом;
- составление плана ответа;
- составление различных таблиц.

3. Для систематизации учебного материала:
- подготовка ответов на контрольные вопросы;
- аналитическая обработка текста;
- подготовка сообщения, доклада;
- тестирование;
- составление кроссворда;
- формирование плаката;
- составление памятки.

4 .Для формирования практических и профессиональных умений.
-решение задач и упражнений по образцу;
-решение ситуативных и профессиональных задач;
- проведение анкетирования и исследования.

Средства обучения необходимы для организации самостоятельной работы:

1. Дидактические средства, которые могут быть источником самостоятельного приобретения знаний (первоисточники, документы, тексты художественных произведений, сборники задач и упражнений, журналы и газеты, учебные фильмы, карты, таблицы);

2. Технические средства, при помощи которых предъявляется учебная информация (компьютеры, аудиовидеотехника);

3. Средства, которые используют для руководства самостоятельной деятельностью студентов

(инструктивно-методические указания, карточки с дифференцированными заданиями для организации индивидуальной и групповой работы, карточки с алгоритмами выполнения заданий).

Разработка и применение средств обучения – это та сторона педагогической деятельности, в которой проявляется индивидуальное мастерство, творческий поиск преподавателя, его умение побудить студентов к творчеству.

Виды практических заданий для самостоятельной работы студентов

1. Составить опорный конспект по теме…
2. Сформулировать вопросы…
3. Сформулировать собственное мнение…
4. Продолжить фразу…
5. Дать определения следующим терминам…
6. Составить опорный конспект своего ответа.
7. Написать реферат.
8. Составить аналитический обзор по теме…
9. Разработать алгоритм последовательности действий…
10. Составить таблицу с целью систематизации материала…
11. Заполнить таблицу, используя…
12. Заполнить блок-схему…
13. Составить тезаурусное поле по теме…
14. Смоделировать конспект урока по теме…
15. Смоделировать домашнее задание.

16. Сделать самоанализ практики: эффективность использования приёмов, методов и средств воспитания детей.

17. Осуществить аналитический разбор публикации по заранее определённой преподавателем теме.

18. Составить тематический кроссворд.

19. Составить план текста, конспект.

20.Решить ситуационные задачи.

21.Подготовиться к семинару, деловой игре.

Приёмы самостоятельной работы студентов:

1. Работа с учебником.

Для обеспечения максимально возможного усвоения материала и с учётом индивидуальных особенностей студенов, можно предложить им следующие приёмы обработки информации учебника:

- конспектирование;
- составление плана учебного текста;
- тезирование;
- аннотирование;
- составление тематического тезауруса;
- выделение проблемы и нахождение путей её решения;
- самостоятельная постановка проблемы и нахождение в тексте путей её решения;
- определение алгоритма практических действий (план, схема).

2. Опорный конспект.

Зачастую педагог обучает от параграфа к параграфу, от пункта к пункту и лишь в конце темы пытается связать весь материал на

обобщающем уроке. Куда целесообразнее, даже с психологической точки зрения, дать студентам представление об изучаемой теме на первом уроке, искусно оформив её содержание как небольшой опорный конспект. Он нужен всем – и сильным, и слабым.

И тогда студенты не будут учиться сегодня, забыв выученное вчера и не зная того, что будет завтра.

Опорный конспект необходимо давать на этапе изучения нового материала, а потом использовать его при повторении.

Опорный конспект позволяет не только обобщать, повторять необходимый теоретический материал, но и даёт педагогу огромный выигрыш во времени при прохождении материала.

Информационное обеспечение обучения

1. Рудаков А. В. Технология разработки программных продуктов.- М.: Издательский центр «Академия», 2016. - 208 с.

2. Культин Н.Б. Turbo Pascal в задачах и примерах: учебное пособие.– БХВ., 2015 –257 с.

3. Павловская Т.А. Паскаль. Программирование на языке высокого уровня. - М.: НОУ "Интуит", 2016 – 156 с.

4. Федорова Г.Н. Разработка модулей программного обеспечения для компьютерных систем.- М.: Издательский центр «Академия», 2017. - 384 с

5. Орлов В.В. Программная инженерия. Технологии разработки программного обеспечения. - СПб.: Питер, 2016. -456 с.

Интернет-ресурсы

3. Технология разработки программных продуктов:
http://chemisk.narod.ru/html/trpp01.html

4. Введение в технологию разработки программных продуктов:
http://www.intuit.ru/department/se/introprogteach/1/