1. **Программа, комплекс программ, программное средство, программное обеспечение, программный продукт**. Концепция программного изделия: непосредственная производительная сила, промышленная технология **проектирования программ, стандарты и сборочное программирование, контроль и управление качеством программ, экономика программных средств, подготовка специалистов по всем этапам жизненного цикла программ, разделение труда специалистов в проектных организациях.**

Программа представляет собой **набор инструкций или кодов, написанных на языке программировани**я, которые выполняются компьютером для выполнения определенных задач. Программа может быть простой, как скрипт, выполняющий одну функцию, или более сложной, как приложение, предоставляющее множество функций и сервисов.

При структурном подходе программы представляют собой иерархию подпрограмм

При объектном подходе - совокупность обменивающихся сообщениями объектов.

Комплекс программ — это совокупность взаимосвязанных программ, которые работают совместно для решения более широкого круга задач. Комплекс обычно включает в себя приложения, библиотеки и системное программное обеспечение, которые взаимодействуют для достижения общих бизнес-целей.

Вызов программ в программном комплексе осуществляется специальной программой - *диспетчером*, который обеспечивает несложный интерфейс с пользователем и, возможно, выдачу некоторой справочной информации. От пакета программ программный комплекс отличается еще и тем, что несколько программ могут последовательно или циклически вызываться для решения одной задачи, и, следовательно, желательно хранить исходные данные и результаты вызовов в пределах одного пользовательского проекта

Программное средство — это термин, который может использоваться для описания любого типа программной системы, включая индивидуальные программы, комплексы программ и инструментальные средства, которые обеспечивают определенные функции или сервисы пользователям или другим программам.

Программное обеспечение - это программы, выполняемые вычислительной системой. Здесь подразумевается как одна, так и несколько программ, которые может выполнять ЭВМ. Охватывает все типы компьютерных программ, включая операционные системы, приложения, утилиты и базы данных, которые предоставляют пользователю возможности для выполнения задач на компьютере. Программное обеспечение может быть классифицировано как системное, прикладное или инструментальное.

Программный продукт — это готовое к использованию программное обеспечение, которое было разработано и протестировано для удовлетворения конкретных требований и которое обычно поставляется с документацией, поддержкой и обновлениями. Программные продукты могут распространяться коммерчески или быть доступны бесплатно.

Программные продукты предназначены для широкого распространения и продажи. Каждый программный продукт предназначен для выполнения определенных функций. По назначению все программные продукты можно разделить натри группы: системные, прикладные и гибридные

![[Untitled 84.png|Untitled 84.png]]

Программное изделие – это не просто программа для ЭВМ, а *продукт* тщательного планирования и целенаправленной разработки, сопровождаемый четкой документацией, прошедший все необходимые испытания, описанный в соответствующих технических публикациях, обеспеченный обученным персоналом, размноженный в требуемом количестве экземпляров, обслуживаемый и контролируемый поставщиком по заранее продуманному плану.

Концепция программного изделия охватывает целый ряд аспектов, связанных с процессом создания и поддержки программного обеспечения. Рассмотрим каждый из этих аспектов более подробно (*тут частями текст из нейронки*):

1. **Непосредственная производительная сила:**
   Программное изделие рассматривается в качестве актива, который напрямую влияет на производственные процессы и эффективность труда. Оно может автоматизировать задачи, ускорять обработку данных и улучшать принятие решений, тем самым увеличивая производительность.
2. **Промышленная технология проектирования программ:**

Под промышленной технологией проектирования понимаются электронные тех-нологии, которые обеспечивают поддержку процессов ЖЦ ПО на всех стадиях, которые, как правило, соответствуют стан-дарту ISO/I EC 12207. Это включает в себя использование методологий разработки, паттернов проектирования и фреймворков, которые обеспечивают структурированный и повторяемый процесс создания программных изделий.

3. **Стандарты и сборочное программирование:**
   Применение стандартов в программировании и использование компонентов, которые могут быть собраны вместе, как в производстве, обеспечивает совместимость, упрощает интеграцию и поддержку программных изделий.
   Программисты, разрабатывая программы без применения каких-либо методов программирования, выделяют повторно используемые операторы и оформляют их в виде отдельных, самостоятельных фрагментов или подпрограмм для дальнейшего использования.
   Сборочное программирование:
   - является одним из методов программирования и подчиняется общим закономерностям;
   - представляет одну из форм повторного использования программных средств;
   - качественно отличаться от процессов сборки в других методах.
4. **Контроль и управление качеством программ:**
   Качество программного изделия является критически важным, и его необходимо контролировать на всех этапах жизненного цикла разработки. Это включает в себя тестирование, ревизию кода, контроль версий и управление изменениями. Основными критериями могут являться:
   - функциональность
   - надежность
   - мобильность
   - эффективность
   - удобство использования
   - сопровождение - процесс улучшения, исправления ошибок, добавления нового функционала.
5. **Экономика программных средств:**
   Экономические аспекты программного изделия охватывают его стоимость разработки, поддержки и использования, а также анализ рентабельности инвестиций в программное обеспечение.
   Прежде всего нужно понять - эффективность готового продукта и оправдаются ли затраты на разработку. Следовательно, проекты традиционно начинаются с анализа и разработки технико-экономического обоснования (ТЭО) предстоящего жизненного цикла (ЖЦ) проекта и эксплуатации предполагаемого продукта.
   ТЭО проектов на начальном этапе их развития должно содержать оценки рисков реализации поставленных целей, обеспечивать возможность планирования и выполнения жизненного цикла продукта или указывать на недопустимо высокий риск его реализации и целесообразность прекращения разработки.
   Должен быть подготовлен согласованный между заказчиком и разработчиком первичный документ, в котором определены цели и задачи проекта, предполагаемые характеристики продукта и необходимые ресурсы для его реализации.
6. **Подготовка специалистов** по всем этапам жизненного цикла программ:
   Эффективная разработка программного изделия требует квалифицированных специалистов, обладающих знаниями и навыками в области проектирования, разработки, тестирования, внедрения и поддержки программного обеспечения.
7. **Разделение труда специалистов** в проектных организациях:
   Разработка программного изделия часто предполагает работу многопрофильной команды, где каждый специалист выполняет свою роль, внося вклад в общий проект. Это подразумевает разделение труда и специализацию, что повышает эффективность и качество конечного продукта.

Критерии качества программного изделия.

1. Программа является **правильной**, если она работает в соответствии с техническим заданием (ТЗ - документ, которым завершается постановка задачи).

2. Программа является **точной**, если выдаваемые ею числовые данные имеют допустимые отклонения от аналогичных результатов, полученных с помощью идеальных математических зависимостей.
3. Программа является **совместимой**, если она работает должным образом не только автономно, но и как часть программной системы.
4. Программа является **надежной**, если она при всех входных данных обеспечивает полную повторяемость результатов.
5. Программа является **универсальной**, если она правильно работает при любых допустимых вариантах исходных данных. В ходе разработки программ предусматриваются специальные средства защиты от ввода неправильных данных, обеспечивающие целостность системы.
6. Программа является **защищенной**, если она сохраняет работоспособность при возникновении сбоев (режим реального времени, программа большого времени выполнения).
7. Программа является **полезной**, если задача, которую она решает, представляет практическую ценность.
8. Программа является **эффективной**, если объем требуемых для ее работы ресурсов ЭВМ не превышает допустимого предела.
9. Программа является **проверяемой**, если ее качества могут быть продемонстрированы на практике (проверка правильности и универсальности). Существуют формальные математические методы проверки и неформальные (прогоны программы с остановками в контрольных точках, обсуждение результатов заинтересованными пользователями).
10. Программа является **адаптируемой**, если она допускает быструю модификацию с целью приспособления к изменяющимся условиям функционирования.

2. **Технология программирования, основные этапы развития: «стихийное» программирование, структурное программирование, объектно-ориентированное программирование, компонентное программирование.**

Технологией программирования - это совокупность методов и средств, используемых в

процессе разработки программного обеспечения.

Как любая другая технология, она представляет собой набор технологических инструкций, включающих:

- указание последовательности выполнения технологических операций;
- перечисление условий, при которых выполняется та или иная операция;
- описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т. п.

Технология также определяет способ описания проектируемой модели, используемой на конкретном этапе разработки

![[Untitled 85.png|Untitled 85.png]]

Различают технологии:

1. используемые на конкретных этапах разработки или для решения отдельных задач этих этапов (в основе лежит ограниченно применимый метод, позволяющий решить конкретную задачу)
2. технологии, охватывающие несколько этапов или весь процесс разработки (в основе вторых обычно лежит базовый метод или подход, определяющий совокупность методов, используемых на разных этапах разработки, или методологию)

Основные этапы развития:

- **Стихийное программирование. 1945-1965г**
  1. Программирование без правил. Появилось с появлением первых ВМ, во времена ВМВ. В СССР в 1948г
  2. Ассемблеры
  3. Языки высокого уровня
  4. Создание подпрограмм.
  Задача разбивалась на под задачи с которыми работали подпрограммы
  5. Подпрограммы получили свои наборы данных (появление локальных данных)
- **Структурное программирование 1965-95г**
  Сложное программное обеспечение не эффективно. (кол-во операторов < 10^5)
  1. Возможность использование принципа процедурной декомпозиции. Появление процедурных языков.
  Появляются элементы передачи управления и вложенных подпрограмм, области видимости.
  Pascal, C, PL1
  2. Структурирование данных и создание своих типов.
  3. Модульное программирование
  связи между модулями осуществлялись с помощью межмодульных таблиц
- **ООП**
  Необходимость использования одного и того же инструментария. Сложности перехода на новые версии.
  1. появление ООП
  2. Системы визуального программирования
  Они позволяли автоматизировать создание интерфейса.
- **Компонентный подход и применение case-технологий с 95**
  1. Программное обеспеченье строиться из отдельных компонентов, которые взаимодействуют через стандартные двоечные интерфейсы.
  COM - компонентная модель объектов - Microsoft
  CORBA - технология создания распределенных приложений. Предполагала посредника для запросов(брокера) - IBM
  2. **Case-системы**

1 уровень более примитивны

- **“Стихийное” программирование**

  Период от момента появления первых вычислительных машин до середины 60-х годов XX в.
  Особенности периода:
  - отсутствовали сформулированные технологии;
  - программы имели простейшую структуру;
  - состояли из собственно программы на машинном языке и обрабатываемых ею данных;

  Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании.
  Появление ассемблеров позволило вместо двоичных или 16-ричных кодов использовать символические имена данных и мнемоники кодов операций. В результате программы стали более «читаемыми».
  Создание языков программирования высокого уровня существенно упростило программирование вычислений и позволило увеличить сложность программ.
  Революционным было появление подпрограмм. Подпрограммы можно было сохранять и использовать в других программах. В результате были созданы огромные библиотеки, которые по мере надобности вызывались из разрабатываемой программы.
  ![[Untitled 1 42.png|Untitled 1 42.png]]
  Тут использовались глобальные данные. При увеличении количества подпрограмм вероятность того, что данные исказятся возрастала. Чтобы сократить количество таких ошибок, было предложено в подпрограммах размещать локальные данные
  ![[Untitled 2 29.png|Untitled 2 29.png]]
  В начале 60-х годов XX в. разразился «кризис программирования». Проект устаревал раньше, чем был готов к внедрению, увеличивалась его стоимость, и в результате многие проекты так никогда и не были завершены.
  Минусы “стихийного” подхода:
  - использовалась разработка «снизу-вверх» - подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых затем пытались построить сложную программу
  - интерфейсы подпрограмм получались сложными
  - при сборке программного продукта выявлялось большое количество ошибок согласования
  - процесс тестирования и отладки программ занимал более 80% времени разработки😐
  Анализ причин возникновения большинства ошибок позволил сформулировать новый подход к программированию

- **Структурный подход**

  60-70-е годы XX в
  > Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения
  В основе структурного подхода лежит декомпозиция. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название процедурной декомпозиции.
  Особенности подхода:
  - требовал представления задачи в виде иерархии подзадач простейшей структуры
  - проектирование «сверху вниз» и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм
  - поддержка принципов структурного программирования была заложена в основу процедурных языков программирования (PL/1, ALGOL-68, Pascal, C)
  Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития структурирования данных:
  - появляется возможность определения пользовательских типов данных
  - стремление разграничить доступ к глобальным данным программы

- появилась и начала развиваться технология модульного программирования
![[Untitled 3 20.png|Untitled 3 20.png]]
> Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые модули (библиотеки подпрограмм).
Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещен.
Итоги:
Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы, размер которых не превышает 100 000 операторов. НО при увеличении размера программы обычно возрастает сложность межмодульных интерфейсов.

- **Объектный подход**
с середины 80-х до конца 90-х годов XX в
> **Объектно-ориентированное программирование** определяется как технология
> создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа (класса), а классы образуют иерархию с наследованием свойств
Сначала использовался только в языке Smalltalk (70-е годы XX в.), а затем был использован в новых версиях универсальных языков программирования, таких, как Pascal, C++, Modula, Java.
Плюсы объектного подхода:
- «более естественная» декомпозиция программного обеспечения
- более полная локализация данных и интегрирование их с подпрограммами обработки
- позволяет вести практически независимую разработку отдельных частей программы
- новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения
В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений.
Были созданы среды, поддерживающие визуальное программирование, например, Delphi, C++ Builder, Visual C++ и т. д
![[Untitled 4 12.png|Untitled 4 12.png]]
Таким образом, при использовании этих языков программирования сохраняется зависимость модулей программного обеспечения от адресов экспортируемых полей и методов, а также структур и форматов данных. Эта зависимость объективна, так как модули должны взаимодействовать между собой, обращаясь к ресурсам друг друга. Связи модулей нельзя разорвать, но можно попробовать стандартизировать их взаимодействие

- **Компонентный подход**
> **Компонентный подход** предполагает построение программного обеспечения из отдельных компонентов физически отдельно существующих частей программного
> обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы
Объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию.
Компонентный подход лежит в основе технологий, разработанных на базе COM и технологии создания распределенных приложений CORBA.
Технология COM (Component Object Model) фирмы Microsoft представляет собой архитектуру для компонентного программирования, которая позволяет различным компонентам программного обеспечения — от библиотек и приложений до целых подсистем операционной системы — взаимодействовать друг с другом.

COM обеспечивает средства для взаимодействия между компонентами, которые могут быть расположены как в пределах одного процесса, так и в разных процессах на одном компьютере или даже на разных компьютерах в сети.
DCOM (Distributed COM) — это расширение модели COM, которое поддерживает взаимодействие компонентов через сетевые границы, предоставляя механизмы для удаленного вызова методов объектов.
В контексте COM приложение предоставляет свои службы через специальные объекты — **объекты COM**, которые являются экземплярами классов COM. Каждый объект COM включает в себя поля и методы, но в отличие от обычных объектов, он может реализовывать несколько интерфейсов.
Это достигается за счет использования отдельной таблицы адресов методов для каждого интерфейса, аналогично таблицам виртуальных методов в объектно-ориентированном программировании.
Интерфейсы обычно объединяют группу связанных функций и имеют глобальный уникальный идентификатор (GUID), известный как IID (Interface Identifier). Каждый объект COM обязательно реализует базовый интерфейс IUnknown, который используется для управления жизненным циклом объекта и получения доступа к другим интерфейсам объекта.
Объект COM функционирует в составе сервера, который может быть реализован как динамическая библиотека (DLL) или исполняемый файл (EXE). Существуют три типа серверов COM:
- **Внутренний сервер (In-process server)**: реализуется в виде динамической библиотеки (DLL), которая загружается в адресное пространство клиентского приложения, обеспечивая высокую производительность и эффективность.
- **Локальный сервер (Local server)**: реализуется отдельным процессом, который работает на том же компьютере, что и клиентское приложение.
- **Удаленный сервер (Remote server)**: реализуется процессом на другом компьютере, что позволяет распределить вычислительную нагрузку в сети.
Технология COM и ее распределенный вариант DCOM легли в основу многих других компонентных технолог
Появление компонентного подхода в программировании открывает новые возможности для разработки сложного программного обеспечения. Этот подход позволяет программным компонентам взаимодействовать между собой независимо от того, находятся ли они в одном процессе или распределены в сети. Разработаны расширения COM, а также ряд технологий, основанных на COM, упрощают создание и интеграцию программных компонентов.
Компонентный подход не исключает другие методы разработки, но его применение ожидается достаточно широкое, учитывая существующие вызовы в разработке сложных систем. Технологии, подобные CORBA, предлагают аналогичные решения для гетерогенных сред, дополняя ландшафт распределенного программирования.
В современной разработке программного обеспечения также активно используются CASE-технологии для автоматизации процессов разработки и сопровождения, что становится необходимым в условиях возрастающей сложности систем и требований к их функциональности.

3. **Программные средства как сложные системы: особенности сложных систем, проблемы определения единого обобщенного критерия эффективности, требования к нему, понятие устойчивости программного средства. Программные средства как сложные системы: особенности сложных систем, проблемы определения единого обобщенного критерия эффективности, требования к нему, понятие устойчивости программного средства**

Сложная система – система с разветвленной структурой и значительным количеством взаимосвязанных и взаимодействующих элементов (подсистем), имеющих разные по своему типу связи, которая может сохранять частичную работоспособность при отказе отдельных элементов

Признаками сложной системы являются:

1. Наличие общей задачи
2. Большое количество взаимодействующих компонентов
3. Возможность декомпозиции системы, (т.е. ее разбиение на взаимодействующие подсистемы, решающие автономные функциональные задачи).
4. Иерархическая архитектура системы и иерархия критериев качества
5. Сложность поведения системы, связанная со случайным характером внешних воздействий и большим количеством обратных связей внутри нее.
6. Устойчивость системы по отношению к внешним воздействиям. Наличие самоорганизации и самоадаптации при различных возмущениях.
7. Высокая надежность системы в целом при абсолютной надежности ее компонентов

Особенности современных сложных систем

1. **Комплексный подход**. Это подразумевает целостный подход к автоматизации технологических процессов в организации. Если раньше в каждом отделе была своя, маленькая ИС, то сейчас вся организация работает в единой ИС.
2. **Оперативность**. скорость обработки и доступность информации.
3. **Гибкость,** т.е. и способность быстро менять конфигурацию или функциональный набор.
4. **Распределенная ИС.** Распределенная ИС подразумевает многоуровневую структуру и наличие иерархии серверов.
5. **Взаимосвязь с другими ИС**. В современных ИС должна быть предусмотрена хотя бы возможность импортировать и экспортировать массивы данных в общепринятых форматах обмена данными
6. **Доступность информации извне.** Современная ИС должна иметь механизмы публикации своих данных в Интернет для внешних пользователей (

Под термином «**критерий эффективности**» понимается чаще всего условие, на основе которого осуществляется определение показателя эффективности.

==СТС - сложная тех сиситема==

Критерий эффективности должен устанавливать соотношение между ожидаемым эффектом, и требуемым или заданным результатом.

Различают критерии «собственные» и «несобственные».

1. **Несобственные** - критерии, на основе которых оценивается эффективность составной части системы
2. **Собственные** - Исследуемый объект рассматривается как самостоятельная система.

Критерии могут быть:

1. векторными, если результаты функционирования СТС определяются совокупностью показателей
2. скалярными, если оценивание осуществляется на основе набора разнородных требований. Тогда удовлетворение требований по одному из показателей осуществляется на основе скалярного критерия.

Критерии могут быть:

1. частными,
2. общими
3. обобщенными.

Проблема выбора критериев достаточно серьезная: для большинства крупномасштабных задач характерно наличие целого ряда критериев и соответствующих им показателей , одни из которых следует максимизировать, а другие - минимизировать. Можно ли найти решение, удовлетворяющее сразу всем требованиям? **Нет**.

Решение, обращающее в максимум какой-либо показатель, как правило, не обращает ни в максимум, ни в минимум другие.
Стараются составить из нескольких показателей один и пользоваться им при выборе решения.
Часто такой показатель принимает форму дроби, у которой в качестве числителя используется показатель, подлежащий увеличению, а в качестве знаменателя - показатель, значение которого должно быть уменьшено. В числителе - полезный эффект, в знаменателе - затраты.
Однако такой метод конструирования объединенного показателя некорректен. Он основан на допущении, что недостаток в одном показателе может быть скомпенсирован за счет другого.
Существуют две основные формы представления показателя эффективности.
Принципом максимизации эффекта - достижение максимума полезного эффекта при заданных затратах ресурсов.
Принцип экономии ресурсов - минимизации затрат ресурсов с обязательным достижением заданного эффекта.
Принципы максимизации эффекта и экономии ресурсов эквивалентны друг другу, в том смысле, что для поиска неизвестного решения можно использовать любую из процедур нахождения экстремума. Результат в обоих случаях должен быть один и тот же. 💀

4. **Особенности функционирования сложных программных средств: работа в реальном времени, многообразие функций, надежность функционирования.**

работа в реальном времени, многообразие функций, надежность функционирования

Создание сложных систем с заданными характеристиками при ограниченных ресурсах требует проведения определенного комплекса мероприятий для достижения поставленной цели, который получил название **проект**.

Целенаправленное управление проектом предназначено для пропорционального распределения ресурсов между работами по созданию системы на протяжении всего цикла проектирования вплоть до внедрения системы в серийное производство.

В общем случае при проектировании необходимо создать в соответствии с принятым критерием эффективности оптимальную систему управления или обработки информации при ограничениях двух типов.

1. **Первый тип ограничений** характеризует уровень современных знаний теории и методов решения поставленных задач, принципов построения основных функциональных алгоритмов, методов структурного построения сложных систем и технологии их проектирования.
2. **Второй тип ограничений** относится в основном к техническим параметрам средств, на которых предполагается реализовать сложную систему, и к ресурсам, которые могут быть выделены на разработку и эксплуатацию системы. При проектировании ПС такими техническими ограничениями прежде всего являются параметры ЭВМ (объемы памяти, быстродействие, характеристики обмена информацией и т.д.), на которых предполагается реализовать КП. Важнейшим ресурсов проектирования являются кадры специалистов соответствующей квалификации, которые могут быть использованы для разработки системы. Кроме того, ресурсами проектирования являются материальные и финансовые затраты, доступные как в процессе создания, так и при последующей эксплуатации системы.

Особенности функционирования сложной системы:

1. **Работа в режиме реального времени** является одним из наиболее сложных режимов функционирования ПО, поскольку от реального времени **зависят не только моменты выполнения отдельных задач, но и получаемые результаты.** В сложных системах реальное время является одним из наиболее сложных параметров. **Его искажение может нарушить временную связь, что может привести к полному отказу системы.**
2. **Настраиваемость ПО**. Один и т.ж. программный комплекс может быть использован для работы с несколькими разнотипными объектами, при этом он сам являясь объектом управления для системы более высокого уровня. Изменение характеристик или состава обрабатываемых объектов не вызывает фундаментальной переработки программы, поскольку возможность подобных изменений закладывается на этапе разработки.
3. Строгая последовательность решения задач не может быть заранее определена из за **большого количества функциональных задач**, решаемых за небольшой промежуток времени из-за сложности связей внутри системы и возможности обмена информацией с большим количеством внешних абонентов (сообщения от абонентов могут поступать в произвольные моменты времени).
4. **Надежность функционирования** при искажениях информации, сбоях и частичных отказах аппаратуры. Для обеспечения необходимой степени надежности широко применяются различные методы контроля, параллельное решение задач, работа в многопроцессорном режиме и т. д.
5. Проблемы проектирования сложных программных средств: рациональное структурное построение, технология разработки, стандартизация; блочно-иерархический подход.

5. **Проблемы проектирования сложных программных средств- рациональное структурное построение, технология разработки, стандартизация; блочно-иерархический подход**

Большинство современных программных систем объективно очень сложны. Эта сложность обуславливается многими причинами, главной из которых является логическая сложность решаемых ими задач.

Дополнительными факторами, увеличивающими сложность разработки программных систем, являются:

- **сложность определения требований** к программным системам
  1. при определении требований необходимо учесть большое количество различных факторов
  2. разработчики систем не являются специалистами в автоматизируемых предметных областях.
- **отсутствие удовлетворительных средств описания поведения дискретных систем с большим числом состояний** при недетерминированной последовательности входных воздействий
  В процессе создания программных систем используют языки сравнительно низкого уровня. Это приводит к ранней детализации операций в процессе создания программного обеспечения и увеличивает объем описаний разрабатываемых продуктов
- **коллективная разработка**
- **необходимость увеличения степени повторяемости кодов**
  компоненты приходится делать более универсальными, что увеличивает сложность разработки.

При разработке структуры программного средства в процессе системного проектирования, прежде всего, необходимо сформулировать критерии ее формирования. Важнейшими критериями могут быть:

1. модифицируемость,
2. отлаживаемость
3. удобство управления разработкой ПС,
4. обеспечение возможности контролируемого изменения конфигурации, состава и функций компонентов с сохранением целостности структурного построения базовых версий ПС.

В зависимости от особенностей проблемной области критериями при выборе архитектуры ПС могут также применяться:

1. эффективное использование памяти или производительности реализующей ЭВМ,
2. трудоемкость или длительность разработки,
3. надежность, безопасность и защищенность.

Основные принципы и правила структурирования ПС можно объединить в группы, которые отражают:

- стандартизированную структуру **целостного построения** ПС определенного класса;
- унифицированные правила структурного построения функциональных п**рограммных компонентов и модулей;**
- стандартизированную структуру **базы данны**х, обрабатываемых программами;
- унифицированные правила структурного построения информационных **модулей**, заполняющих базу данных;
- унифицированные правила организации и структурного построения межмодульных **интерфейсов** программных компонентов;
- унифицированные правила внешнего интерфейса и **взаимодействия** компонентов ПС и БД с внешней средой, с операционной системой и другими типовыми средствами организации вычислительного процесса, защиты и контроля системы.

Структурный анализ, исходя из функционального описания системы в целом, позволяет разделить ее на функциональные части, выделить функциональные описания отдельных частей, исследовать в них информационные потоки и формализовать структуры данных.

Технология разработки

Технология разработки программного обеспечения (ПО) – это комплекс мер по созданию программных продуктов (ПП). Данная деятельность включает в себя несколько этапов, с которыми так или иначе придётся столкнуться при разработке достаточно крупного ПО.

Ключевым понятием в технологии разработки ПО является понятие жизненного цикла программного продукта.

Стандартизация

Для значительного повышения производительности труда требуется стандартизация и комплексная автоматизация всего технологического процесса.
На каждую программу задаваться технические условия, обеспечивающие детальную расшифровку ее функций и возможность полной проверки.
В идеале создание больших программных изделий желательно сводить к сопряжению комплектующих изделий (групп программ или модулей).
Необходимо стандартизировать структуру и формы представления документов на каждую разработанную и испытанную программу, на программное изделие. В настоящее время такой стандарт существует и носит название «Единая система программной документации».
Другой задачей является стандартизация структуры и правил сопряжения программ при передаче управления и при обменной информации. Должны быть унифицированы правила описания и использования переменных, правила распределения памяти, требования к обмену информацией между отдельными программами, комплексами программ и автономными системами управления.
Необходима также стандартизация методов и требований к обеспечению и измерению качества сложных программных изделий. Эти методы должны позволять контролировать надежность функционирования созданных программных изделий в реальных условиях, рассчитывать и прогнозировать возможную достоверность результатов в зависимости от затрат на отладку и принятых мер для автоматического выявления искажений и для исправления его результатов.

- **Блочно-иерархический подход к созданию сложных систем**
  ==//опиши как идеф масштабируется==
  Большинство сложных систем как в природе, так и в технике имеет иерархическую внутреннюю структуру. Это связано с тем, что обычно связи элементов сложных систем различны как по типу, так и по силе, что и позволяет рассматривать эти системы как некоторую совокупность взаимозависимых подсистем.
  > **Блочно-иерархический подход** предполагает сначала создавать части объектов (блоки, модули), а затем собирать из них сам объект.
  Это декомпозиция. При декомпозиции учитывают, что связи между отдельными частями должны быть слабее, чем связи элементов внутри частей.
  При создании очень сложных объектов процесс декомпозиции выполняется многократно:
  каждый блок, в свою очередь, декомпозируют на части пока не получают блоки, которые сравнительно легко разработать. Данный метод разработки получил название
  **пошаговой
  детализации.
  **
  Результат декомпозиции обычно представляют в виде схемы иерархии, на нижнем уровне которой располагают сравнительно простые блоки, а на верхнем - объект, подлежащий разработке
  На каждом иерархическом уровне описание блоков выполняют с определенной степенью детализации, абстрагируясь от несущественных деталей. Следовательно, для каждого уровня используют свои формы документации и свои модели, отражающие сущность процессов, выполняемых каждым блоком.
  Так для объекта в целом, как правило, удается сформулировать лишь самые общие требования, а блоки нижнего уровня должны быть специфицированы так, чтобы из них действительно можно было собрать работающий объект. **Другими словами, чем больше блок, тем более абстрактным должно быть его описание.**
  В основе блочно-иерархического подхода лежат:
  - декомпозиция и иерархическое упорядочение
  - непротиворечивость — контроль согласованности элементов между собой;
  - полнота - контроль на присутствие лишних элементов;
  - формализация - строгость методического подхода;
  - повторяемость - необходимость выделения одинаковых блоков для удешевления и ускорения разработки;
  - локальная оптимизация - оптимизация в пределах уровня иерархии.

6. **Жизненный цикл программного обеспечения, процессы жизненного цикла, связь между процессами.**

Жизненным циклом программного обеспечения называют период от момента появления идеи создания некоторого программного обеспечения до момента завершения его поддержки фирмой разработчиком или фирмой, выполнявшей сопровождение.

Состав процессов жизненного цикла регламентируется международным стандартом **ISO/1EC 12207**: 1995 «Information Technologe - Software Life Cycle Processes» («Информационные технологии - Процессы жизненного цикла программного обеспечения»). Этот стандарт описывает структуру жизненного цикла программного обеспечения и его процессы.

Процесс жизненного цикла определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные

Все процессы можно разделить на 3 группы:

Разработка

Приобретение

Поставка

Эксплуатация

Сопровождение

1. **Основные процессы:**
2. **Организационные процессы:**
   Управление
   Усовершенствование
   Создание инфраструктуры
   Обучение
3. **Вспомогательные процессы:**
   Документирование
   Управление конфигурацией
   Менеджмент качества
   Обеспечение качества
   Верификация
   Аттестация
   Совместная оценка
   Аудит
   Разрешение проблем

   Стандарт предлагает некоторый базовый набор взаимодействий между процессами с различных точек зрения (либо в различных аспектах)

   4. договорный аспект, в котором заказчик и поставщик вступают в договорные отношения и реализуют процессы приобретения и поставки;
   5. аспект управления, который включает действия управления лицами, участвующими в ЖЦ ПО (поставщик, заказчик, разработчик, оператор и др.);
   6. аспект эксплуатации, включающий действия оператора по предоставлению услуг пользователям системы;
   7. инженерный аспект, который содержит действия разработчика или службы сопровождения по решению технических задач, связанных с разработкой или модификацией программных продуктов;
   8. аспект поддержки, связанный с реализацией вспомогательных процессов, с помощью которых службы поддержки предоставляют необходимые услуги всем остальным участникам работ. В этом аспекте можно выделить аспект управления качеством ПО, включающий процессы обеспечения качества, верификацию, аттестацию, совместную оценку и аудит.

   Организационные процессы выполняются на корпоративном уровне или на уровне всей организации в целом, создавая базу для реализации и постоянного совершенствования процессов ЖЦ ПО.

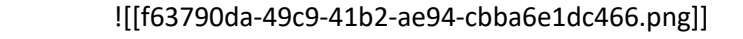
Сложно:

Перечислим основные этапы жизненного цикла программы:

- **Процесс приобретения.** Данный процесс представляет собой действия заказчика разработки ПО, и обычно включает в себя такие мероприятия, как:
  - формирование требований и ограничений к программному продукту (ограничения могут быть связаны с выбором программной архитектуры, а также с приемлемым быстродействием системы и т.д.);
  - заключение договора на разработку;
  - анализ и аудит работы исполнителя.
  В конце данного процесса заказчик осуществляет приёмку готового программного продукта.
- **Процесс поставки** включает в себя мероприятия, проводимые исполнителем по поставке ПО.
  - Исполнитель анализирует требования заказчика,
  - выполняет проектирование и анализ работ,
  - решает, как будет происходить процесс конструирования (программирования): своими силами, либо же с привлечением сторонних команд разработки (подрядчика),
  - также осуществляет оценку и контроль качества готового программного продукта
  - выполняет непосредственно поставку продукта и сопутствующие завершающие мероприятия.
- **Процесс разработки**. Он включает:
  - работа с заказчиком и документирование его видения и его требований к программе.
  - проектирование. На данном этапе создания программного продукта разрабатывается архитектура компонентов ПО, выбираются нужные шаблоны проектирования (паттерны) и составляется схема информационной базы данных системы.
  - начинается разработка ПП. На этапе разработки также выполняется документирование системы.
  - тестирование системы в целом. тем самым подтверждается её соответствие требованиям заказчика. Когда все тесты пройдены, программное обеспечение готово к выпуску.
- **Процесс эксплуатации**. После того, как программное обеспечение будет готово, начинается процесс его эксплуатации организацией-заказчиком и её операторами.
- **Процесс сопровождения**. Фирма-разработчик осуществляет поддержку пользователей программного продукта в случае возникновения у них каких-либо вопросов или проблем. Если в процессе эксплуатации будет обнаружена ошибка в ПП, разработчики должны её устранить. Процесс эксплуатации и процесс сопровождения идут параллельно.

Стандарт предлагает некоторый базовый набор взаимодействий между процессами с различных точек зрения (либо в различных аспектах)

7. **Основные процессы жизненного цикла: приобретение, поставка, разработка, эксплуатация, сопровождение.**



- Процесс **приобретения** состоит из действий и задач заказчика, приобретающего ПС. Данный процесс охватывает следующие действия :
  1. инициирование приобретения;
  2. подготовку заявочных предложений;
  3. подготовку и корректировку договора;
  4. надзор за деятельностью поставщика;
  5. приемку и завершение работ.
- Процесс **поставки** охватывает действия и задачи, выполняемые поставщиком, который снабжает заказчика программным продуктом или услугой. Данный процесс включает следующие действия:
  1. инициирование поставки;
  2. подготовку ответа на заявочные предложения;
  3. подготовку договора;
  4. планирование работ по договору;
  5. выполнение и контроль договорных работ и их оценку;
  6. поставку и завершение работ.
- Процесс **разработки**
  предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов в соответствии с заданными требованиями. Сюда включается оформление проектной и эксплуатационной документации, подготовка материалов, необходимых для проверки.
  Процесс разработки включает следующие действия:
  1. подготовительную работу;
  2. анализ требований, предъявляемых к системе;
  3. проектирование архитектуры системы;
  4. анализ требований, предъявляемых к программному обеспечению;
  5. проектирование архитектуры программного обеспечения;
  6. детальное проектирование программного обеспечения;
  7. кодирование и тестирование программного обеспечения;
  8. интеграцию программного обеспечения;
  9. квалификационное тестирование программного обеспечения;
  10. интеграцию системы;
  11. квалификационное тестирование системы;
  12. установку программного обеспечения;
  13. приемку программного обеспечения.
- **Эксплуатация**
  Процесс эксплуатации включает следующие действия.
  1. Подготовительная работа, которая включает проведение оператором следующих задач:
  2. Эксплуатационное тестирование, осуществляемое для каждой очередной редакции программного продукта, после чего эта редакция передается в эксплуатацию.
  3. Собственно эксплуатация системы, которая выполняется в предназначенной для этого среде в соответствии с пользовательской документацией.
  4. Поддержка пользователей – оказание помощи и консультаций при обнаружении ошибок в процессе эксплуатации ПО.
- **Сопровождение**
  представляет собой действия и задачи, которые выполняются сопровождающей организацией, при изменениях (модификациях) программного продукта и соответствующей документации, вызванных возникшими проблемами или потребностями в модернизации или адаптации ПО.
  Процесс сопровождения охватывает следующие действия:

1. подготовительную работу (планирование действий и работ, определение процедур локализации и разрешения проблем, возникающих в процессе сопровождения);
2. анализ проблем и запросов на модификацию ПО (анализ сообщений о возникшей проблеме или запроса на модификацию, оценка масштаба, стоимости модификации, получаемого эффекта, оценка целесообразности модификации);
3. модификацию ПО (внесение изменений в компоненты программного продукта и документацию в соответствии с правилами процесса разработки);
4. проверку и приемку (в части целостности модифицируемой системы);
5. перенос ПО в другую среду (конвертирование программ и данных, параллельная эксплуатация ПО в старой и новой среде в течение некоторого периода времени);
6. снятие ПО с эксплуатации по решению заказчика при участии эксплуатирующей организации, службы сопровождения и пользователей. При этом программные продукты и документации подлежат архивированию в соответствии с договором.

8. **Вспомогательные процессы жизненного цикла: документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем.**

- **Процесс документирования.**
  Предусматривает формализованное описание информации, созданной в течение ЖЦ ПО. Данный процесс состоит из набора действий, с помощью которых планируют, проектируют, разрабатывают, выпускают, редактируют, распространяют и сопровождают документы, необходимые для всех заинтересованных лиц, таких как руководство, технические специалисты и пользователи системы.
  Процесс документирования включает следующие действия [2]:
  1. подготовительную работу;
  2. проектирование и разработку;
  3. выпуск документации;
  4. сопровождение.
- Процесс **управления конфигурацией**
  включает административные и технические процедуры на всем протяжении ЖЦ ПО для определения состояния компонентов ПО, описания и подготовки отчетов о состоянии компонентов ПО и запросов на модификацию, обеспечения полноты, совместимости и корректности компонентов ПО, управления хранением и поставкой ПО.
  Согласно стандарту IEEE-90 под конфигурацией ПО понимается совокупность его функциональных и физических характеристик, установленных в технической документации и реализованных в ПО. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации по управлению конфигурацией ПО отражены в стандарте ISO/IEC 15288 "Information Technology. Software Life Cycle Process. Configuration Management for Software".
  Процесс управления конфигурацией включает следующие действия:
  1. подготовительную работу, заключающуюся в планировании управления конфигурацией;
  2. идентификацию конфигурации, устанавливающую правила, с помощью которых однозначно идентифицируются компоненты ПО и их версии.
  3. контроль конфигурации – действие, предназначенное для систематической оценки предлагаемых модификаций ПО и координированной их реализации с учетом эффективности каждой модификации и затрат на ее выполнение;
  4. учет состояния конфигурации, представляющий собой регистрацию состояния компонентов ПО. Обеспечивает подготовку отчетов о реализованных и отвергнутых модификациях версий компонентов ПО. Совокупность отчетов дает однозначное отражение текущего состояния системы и ее компонентов, а также обеспечивает ведение истории модификаций;
  5. оценку конфигурации, заключающуюся в определении функциональной полноты компонентов ПО, а также соответствия их физического состояния текущему техническому описанию;
  6. управление выпуском и поставку, охватывающие изготовление эталонных копий программ и документации, их хранение и поставку пользователям в соответствии с порядком, принятом в организации.
- Процесс **обеспечения качества**
  должен обеспечивать гарантии того, что ПО и процессы его ЖЦ соответствуют заданным требованиям и утвержденным планам.
  Под качеством ПО понимается совокупность свойств, которая характеризует способность ПО удовлетворять заданным требованиям. Для получения достоверных оценок о создаваемом ПО процесс обеспечения его качества должен происходить независимо от субъектов, непосредственно связанных с разработкой программного продукта. При этом могут использоваться результаты других вспомогательных процессов, таких как верификация, аттестация, совместная оценка, аудит и разрешение проблем.
  Процесс обеспечения качества включает следующие действия:

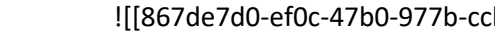
1. подготовительную работу (координацию с другими вспомогательными процессами и планирование самого процесса обеспечения качества ПО с учетом используемых стандартов, методов, процедур и средств);
2. обеспечение качества продукта, подразумевающего гарантированное полное соответствие ПО и его документации требованиям заказчика, предусмотренным в договоре;
3. обеспечение качества процесса, предполагающее гарантированное соответствие процессов ЖЦ ПО, методов разработки, среды разработки и квалификации персонала условиям договора, установленным стандартам и процедурам;
4. обеспечение прочих показателей качества ПО, осуществляемое в соответствии с условиями договора и стандартом качества ISO 9001.

- Процесс **верификации**
  состоит в определении того факта, что ПО, являющееся результатом некоторой деятельности, полностью удовлетворяет требованиям или условиям, обусловленным предшествующими действиями. Для повышения эффективности всего процесса ЖЦ ПО верификация должна как можно раньше интегрироваться с использующими ее процессами (т.е. с поставкой, разработкой, эксплуатацией). Процесс верификации может включать анализ, оценку и тестирование.
  Верификация может проводиться с различными степенями независимости (от самого исполнителя до специалистов другой организации, не зависящей от поставщика, разработчика и т.д.). В процессе верификации проверяются следующие условия:
  1. непротиворечивость требований, предъявляемых к системе, и степень учета потребностей пользователей;
  2. возможность поставщика выполнить заданные требования;
  3. соответствие выбранных процессов ЖЦ ПО условиям договора;
  4. адекватность стандартов, процедур и среды разработки процессам ЖЦ ПО;
  5. соответствие проектных спецификаций ПО заданным требованиям;
  6. корректность описания в проектных спецификациях входных и выходных данных, последовательности событий, интерфейсов, логики и т.д.;
  7. соответствие кода проектным спецификациям и требованиям;
  8. тестируемость и корректность кода, его соответствие принятым стандартам кодирования;
  9. корректность интеграции компонентов ПО в систему;
  10. адекватность, полнота и непротиворечивость документации.
- Процесс **аттестации**
  предназначен для определения полноты соответствия заданных требований и созданного ПО их конкретному функциональному назначению (тому, что требуется потребителю). Под аттестацией обычно понимается подтверждение и оценка достоверности проведенного тестирования программного продукта. Аттестация должна гарантировать полное соответствие ПО спецификациям, требованиям и документации, а также возможность безопасного и надежного применения ПО пользователем.
  Аттестация, как и верификация, может осуществляться с различными степенями независимости (вплоть до организации, не зависящей от поставщика, разработчика, оператора или службы сопровождения).
- Процесс **совместной оценки**
  предназначен для оценки состояния работ по проекту и программному продукту, создаваемому при выполнении этих работ. Он сосредоточен в основном на контроле планирования и управления ресурсами, персоналом, аппаратурой и инструментальными средствами проекта.
  Оценка применяется как на уровне управления проектом, так и на уровне технической реализации проекта и проводится в течение всего срока действия договора. Данный процесс может выполняться двумя сторонами, участвующими в договоре, при этом одна сторона проверяет другую.
- Процесс **аудита**
  представляет собой определение соответствия проекта и продукта требованиям, планам и условиям договора. Аудит может выполняться двумя любыми сторонами, участвующими в договоре, когда одна сторона проверяет другую.

Аудит – это ревизия (проверка), проводимая компетентным органом (лицом) в целях обеспечения независимой оценки степени соответствия ПО или процессов установленным требованиям.
Аудит служит для установления соответствия реальных работ и отчетов требованиям, планам и контракту. Аудиторы не должны иметь прямой зависимости от разработчиков ПО. Они определяют состояние работ, использование ресурсов, соответствие документации спецификациям и стандартам, корректность тестирования и др.

- Процесс **разрешения проблем**
  предусматривает анализ и разрешение проблем (включая обнаруженные несоответствия), которые обнаружены в ходе разработки, эксплуатации или других процессов независимо от их происхождения или источника.

9. **Организационные процессы жизненного цикла: управление, создание инфраструктуры, усовершенствование, обучение.**



- Процесс **управления**
  состоит из действий и задач, которые могут выполняться любой стороной, управляющей своими процессами. Данная сторона (менеджер) отвечает за управление выпуском продукта, управление проектом и управление задачами соответствующих процессов, таких как приобретение, поставка, разработка, эксплуатация, сопровождение и др.
  Процесс управления включает следующие действия:
  1. инициирование и определение области управления – менеджер должен убедиться, что необходимые для управления ресурсы (персонал, оборудование и технология) имеются в его распоряжении в достаточном количестве;
  2. планирование, как действие, подразумевает выполнение следующих задач:
  - составление графиков выполнения работ;
  - оценку затрат;
  - выделение требуемых ресурсов;
  - распределение ответственности;
  - оценку рисков, связанных с конкретными задачами;
  - создание инфраструктуры управления.
- Процесс **усовершенствования**
  предусматривает оценку, измерение, контроль и собственно усовершенствование процессов ЖЦ ПО. Этот процесс включает три основных действия:
  - создание процесса;
  - оценку процесса;
  - усовершенствование процесса.

  Усовершенствование процессов ЖЦ ПО направлено на повышение производительности труда всех участвующих в них специалистов за счет совершенствования используемой технологии, методов управления, выбора инструментальных средств и обучения персонала. Усовершенствование основано на анализе достоинств и недостатков каждого процесса. Такому анализу способствует накопление в организации исторической, технической, экономической и иной информации по реализованным проектам.
- Процесс **создания инфраструктуры**
  охватывает выбор и поддержку технологий, стандартов и инструментальных средств, используемых для разработки, эксплуатации или сопровождения ПО. Инфраструктура должна модифицироваться и сопровождаться в соответствии с изменениями требований к соответствующим процессам. Инфраструктура, в свою очередь, является одним из объектов управления конфигурацией.
  Процесс создания инфраструктуры включает следующие действия:
  - подготовительную работу;
  - создание инфраструктуры;
  - сопровождение инфраструктуры.
- Процесс **обучения**
  включает первоначальное обучение и последующее постоянное повышение квалификации персонала и состоит из трех действий:
  - подготовительной работы;
  - работки учебных материалов;
  - реализации планов обучения.

10. **Модели жизненного цикла: поэтапная, каскадная, спиральная, переиспользования и реверсивной инженерии.**

- **Каскадная модель.**

  ![[Untitled 88.png|Untitled 88.png]]

  1970-1985 годы

  переход на следующую стадию осуществляется после того, как полностью будут завершены проектные операции предыдущей стадии и получены все исходные данные для следующей стадии.

  Достоинствами такой схемы являются:

  - получение в конце каждой стадии законченного набора проектной документации, отвечающего требованиям полноты и согласованности;
  - простота планирования процесса разработки.

  Однако данная схема оказалась применимой только к созданию систем, для которых в самом начале разработки удавалось точно и полно сформулировать все требования. Это уменьшало вероятность возникновения в процессе разработки проблем, связанных с принятием неудачного решения на предыдущих стадиях. На практике такие разработки встречается крайне редко.

  В целом необходимость возвратов на предыдущие стадии обусловлена следующими причинами:

  - неточные спецификации, уточнение которых в процессе разработки может привести к необходимости пересмотра уже принятых решений;
  - изменение требований заказчика непосредственно в процессе разработки;
  - быстрое моральное устаревание используемых технических и программных средств;
  - отсутствие удовлетворительных средств описания разработки на стадиях постановки задачи, анализа и проектирования.

- **Модель с промежуточным контролем.**

  ![[Untitled 1 44.png|Untitled 1 44.png]]

  Схема, поддерживающая итерационный характер процесса разработки, была названа схемой с промежуточным контролем.

  Контроль, который выполняется по данной схеме после завершения каждого этапа, позволяет при необходимости вернуться на любой уровень и внести необходимые изменения.

  Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

- **Спиральная модель.**

  ![[Untitled 2 30.png|Untitled 2 30.png]]

  Для преодоления перечисленных проблем в середине 80-х годов XX в, была предложена спиральная схема.

  В соответствии с данной схемой программное обеспечение создается не сразу, а итерационно с использованием метода прототипирования, базирующегося на создании прототипов.

  Именно появление прототипирования привело к тому, что процесс модификации программного обеспечения перестал восприниматься, как «необходимое зло», а стал восприниматься как отдельный важный процесс.

  > Прототипом называют действующий программный продукт, реализующий отдельные
  > функции и внешние интерфейсы разрабатываемого программного обеспечения.

  На первой итерации, как правило, специфицируют, проектируют, реализуют и тестируют интерфейс пользователя. На второй - добавляют некоторый ограниченный набор функций. На последующих этапах этот набор расширяют, наращивая возможности данного продукта.

  Основным **достоинством** данной схемы является то, что, начиная с некоторой итерации, на которой обеспечена определенная функциональная полнота, продукт можно предоставлять пользователю, что позволяет:

  - сократить время до появления первых версий программного продукта;
  - заинтересовать большое количество пользователей, обеспечивая быстрое продвижение следующих версий продукта на рынке;
  - ускорить формирование и уточнение спецификаций за счет появления практики

использования продукта;
- уменьшить вероятность морального устаревания системы за время разработки.
Основной **проблемой** использования спиральной схемы является определение моментов перехода на следующие стадии. Для ее решения обычно ограничивают сроки прохождения каждой стадии, основываясь на экспертных оценках.

- **Модель переиспользования и реверсивной инженерии**
  ![[1000028220.jpg]]
  В ней основной вес проектных решений падает на проектирование. Целью данной модели является повторное использование (переиспользование) в текущих проектах хорошо проверенных на практике «старых» проектных решений, зафиксированных в библиотеках уже законченных проектов. В процессе анализа и предварительного проектирования намечаются планы работ, в которые включены, в том числе задачи проверки альтернативных проектных решений. Затем запускаются работы по созданию запланированных прототипов по каскадной схеме. В результате выбирается одно из альтернативных решений, разрабатываемых в параллельных витках для остаточного цикла разработки продукта. Возможен выбор смешанного варианта на основе объединения результатов нескольких витков.

11. **Стадии жизненного цикла: формирование требований, проектирование, реализация, тестирование, внедрение, эксплуатация и сопровождение, снятие с эксплуатации. Взаимосвязь между стадиями и процессами жизненного цикла, матрица фазы-функции.**

Под **стадией** создания ПО понимается часть процесса создания ПО, ограниченная некоторыми временными рамками, и заканчивающаяся выпуском какого-то конкретного продукта (моделей ПО, программных компонентов, документации), определяемого заданными для этой стадии требованиями.

Стадии создания ПО выделяются по соображениям рационального планирования и организации работ, заканчивающихся заданными результатами. В состав ЖЦ ПО обычно включают следующие стации:

- **1. Формирование требований к ПО.**
  - **1. Планирование работ**

  предваряющее работы над проектом. Основными задачами являются:
  - определение целей разработки;
  - предварительная экономическая оценка проекта;
  - построение плана – графика выполнения работ;
  - создание и обучение совместной рабочей группы.
  - **2. Проведение обследования деятельности автоматизируемого объекта**

  (организации), в рамках которого осуществляются:
  - предварительное выявление требований к будущей системе;
  - определение структуры организации;
  - определение перечня целевых функций организации;
  - анализ распределения функций по подразделениям и сотрудникам;
  - выявление функциональных взаимодействий между подразделениями, информационных потоков внутри подразделений и между ними, внешних по отношению к организации объектов и внешних информационных взаимодействий;
  - анализ существующих средств автоматизации деятельности организации.
  - **3. Построение моделей деятельности организаций**

  предусматривающее обработку материалов обследования и построение двух видов моделей:
  - ***модели "ASIS" («как есть»),*** отражающей существующее на момент обследования положение дел в организации и позволяющее понять, каким образом функционирует данная организация, а также выявить узкие места и сформулировать предложения по улучшению ситуации;
  - ***модели "TOBE" («как должно быть__»)***, отражающей представление о новых технологиях работы организации.

  Каждая из моделей включает в себя полную функциональную и информационную модель деятельности организации, а также, в случае необходимости, модель, описывающую динамику поведения организации.

  Переход от модели ***"AS-IS__"*** к модели ***"TO-BE"*** может выполняться двумя способами:
  - совершенствованием существующих технологий на основе оценки их эффективности;
  - радикальным изменением технологий и перепроектированием бизнес-процессов (реинжиниринг бизнес-процессов).

  Модель ***"AS-IS__"*** позволяет выявить узкие места в существующих технологиях и предлагать рекомендации по решению проблем независимо от того, предполагается на данном этапе дальнейшая разработка ЭИС или нет. Облегчает обучение сотрудников конкретным направлениям деятельности организации за счет использования наглядных диаграмм.
- **2. Проектирование.**
  - **1. Разработка системного проекта**.

  «Что должна делать будущая система?», а именно:
  - определяются архитектура системы, ее функции,
  - внешние условия функционирования, интерфейсы и распределение функций между пользователями и системой,
  - требования к программным и информационным компонентам,
  - состав исполнителей и сроки разработки.

Основу системного проекта составляют модели проектируемой ЭИС, которые строятся на основе модели ***"ТО-ВЕ"*** . Документальным результатом является техническое задание.

- ***2. Разработка технического проекта__***.

«Как построить систему, чтобы она удовлетворяла предъявленным к ней требованиям?»

На этом этапе на основе системного проекта осуществляется собственно проектирование системы, включающее проектирование архитектуры системы и детальное проектирование. Модели проектируемой ЭИС при этом уточняются и детализируются до необходимого уровня.

Результатом технического проектирования является разработка верифицированной спецификации *ПО*, включающей:

- формирование иерархии программных компонентов, межмодульных интерфейсов по данным и управлению;
- спецификация каждого компонента ПО, имени, назначения, предположений, размеров, последовательности вызовов, входных и выходных данных, ошибочных *выходов, алгоритмов* и логических схем;
- формирование физической и логической структур данных до уровня отдельных полей;
- разработку плана распределения вычислительных ресурсов (времени центральных процессоров, памяти и др.);
- верификацию полноты, непротиворечивости, осуществимости и обоснованности требований;
- предварительный план комплексирования и отладки, план руководства для пользователей и приемных испытаний.

Завершением стадии детального проектирования является сквозной *контроль* проекта или критический поблочный *анализ* проекта.

- **3. Реализация.**

  На этапе реализации строятся прототипы как целой программной системы, так и ее частей, осуществляется физическая реализация структур данных, разрабатываются программные коды, выполняется отладочное тестирование, создается техническая документация. В результате этапа реализации появляется рабочая версия продукта.
  - получение и установка технических и программных средств;
  - тестирование и доводка программного комплекса;
  - разработка инструкций по эксплуатации программно-технических средств.
- **4. Тестирование.**

  Тестирование программного продукта тесно связано с такими этапами разработки, как проектирование и реализация.

  В систему встраиваются специальные механизмы, которые дают возможность производить тестирование программного обеспечения на соответствие требований к нему, проверку оформления и наличия необходимого пакета документации. Результатом тестирования являются устранение всех недостатков программного продукта и заключение о ее качестве.
- **5. Ввод в действие.**
  - ввод технических средств;
  - ввод программных средств;
  - обучение и сертификация персонала;
  - опытная эксплуатация;
  - сдача и подписание актов приёмки-сдачи работ.

  К любой разработке прилагается полный пакет документации, который включает в себя описание программного продукта, руководства пользователей и алгоритмы работы. Поддержка функционирования программного обеспечения должна осуществляться группой технической поддержки разработчика.
- **6. Эксплуатация и сопровождение.**
  - повседневная эксплуатация;
  - общее сопровождение всего проекта.

  Выделяют 4 категории сопровождения:

1. Корректирующее сопровождение – модификация для исправления обнаруженных дефектов: устранение сбоев и т.д.
2. Адаптивное сопровождение – модификация для учёта требуемых изменений: учёт новых требований, изменение окружения ПО и т.д.
3. Совершенствующее сопровождение – модификация для улучшения возможностей ПО: повышение характеристик ПО и т.д.
4. Профилактическое сопровождение – модификация для предупреждения возможных проблем: определение и исправление скрытых дефектов до их реального появления в виде сбоев и т.д.

Эти категории соответствуют разным уровням модификации ПО:

1. Ревизия (пересмотр) – незначительные, часто локальные, изменения кода.
2. Реструктурирование (реструктуризация) – повторная разработка небольшой части кода обычно без изменения интерфейса.
3. Реорганизация (реинжиниринг) – перестройка существующего кода обычно в соответствии с новой моделью или методологией.
4. Реконструирование (реконструкция, повторное конструирование) – перекодирование существенной части кода.

Коррекция предполагает ревизию и реструктурирование, а расширение – реорганизацию и реконструирование. Кроме того, необходимо учитывать класс решаемой задачи и стоимость сопровождения.

- **7. Снятие с эксплуатации.**
  процесс ЖЦ, который заключается в прекращении сопровождения эксплуатируемого ПО. Этот классический процесс выполняется, когда невозможно выполнить ни одну из категорий сопровождения. Он осуществляется предварительным оповещением пользователей о прекращении сопровождения. Но использование ПО возможно вплоть до его морального устаревания.

Границы каждой стадии жизненного цикла программного обеспечения определены некоторыми моментами времени, в которые необходимо принимать определенные критические решения и, следовательно, достигать определенных ключевых целей.

Возможный вариант взаимосвязи и стадий работ с процессами ЖЦ ПО

Модель фазы-функции жизненного цикла программного обеспечения

Чрезвычайно важным мотивом развития моделей жизненного цикла программного обеспечения является потребность в подходящем средстве для *комплексного управления проектом*. По существу, это утверждение указывает на то, что модель должна служить основой организации взаимоотношений между разработчиками, и, таким образом, одной из ее целей является *поддержка функций* менеджера. Это приводит к необходимости наложения на модель контрольных точек, задающих организационно-временные рамки проекта, и организационно-технических, так называемых *производственных функций*, которые выполняются при развитии проекта.

Рис.1.

Наиболее последовательно такое дополнение классической схемы реализовано в *модели Гантера* (рис.1) в виде матрицы «*фазы — функции*». Жирной чертой (с разрывом и стрелкой, обозначающей временное направление) изображен процесс разработки. Контрольные точки и наименования событий указаны под этой чертой. Они пронумерованы. Все развитие проекта в модели привязывается к этим контрольным точкам и событиям.

Функции\Фазы	Постановка задачи	Анализ требований и создание спецификации	Проектирование	Реализация
1.	+			
2.	+	+		
3.		+	+	+
...				+

Уже из упоминания о матрице следует, что *модель Гантера* имеет два измерения:

- фазовое
  выполнение *функции* на одном *этапе* может продолжаться на следующем.
  В данной *модели* жизненный цикл распадается на следующие перекрывающие друг друга *фазы* (*этапы*):
  - ***Этап исследования*** — начинается, когда необходимость *разработки* признана руководством проекта (контрольная точка 0), и заключается в том, что для проекта обосновываются необходимые ресурсы (контрольная точка 1) и формулируются требования к разрабатываемому изделию (контрольная точка 2).
  - ***Анализ осуществимости*** — начинается на *этапе исследования*, когда определены исполнители проекта (контрольная точка 1), и завершается утверждением требований (контрольная точка 3). Цель *этапа* — определить возможность *конструирования* изделия с технической точки зрения (достаточно ли ресурсов, квалификации и т.п.), будет ли изделие удобно для практического *использования*; решение вопросов экономической и коммерческой эффективности.
  - ***Конструирование*** — начинается обычно на *этапе анализа осуществимости*, как только документально зафиксированы предварительные цели проекта (контрольная точка 2), и заканчивается утверждением проектных решений в виде официальной спецификации на *разработку* (контрольная точка 5).
  - ***Программирование*** — начинается на *этапе конструирования*, когда становятся доступными основные спецификации на отдельные компоненты изделия (контрольная точка 4), но не ранее утверждения соглашения о требованиях (контрольная точка 3). Совмещение данной *фазы* с заключительным *этапом конструирования* обеспечивает оперативную проверку проектных решений и некоторых ключевых вопросов *разработки*.
    Цель *этапа* — реализация программ компонентов с последующей сборкой изделия. Он завершается, когда разработчики заканчивают документирование, отладку и компоновку и передают изделие службе, выполняющей независимую *оценку* результатов работы (независимые *испытания* начались — контрольная точка 7).
  - ***Оценка*** — является буферной зоной между началом *испытаний* и практическим *использованием* изделия. *Этап* начинается, как только проведены внутренние (силами разработчиков) *испытания* изделия (контрольная точка 6) и заканчивается, когда подтверждается готовность изделия к эксплуатации (контрольная точка 9).
  - ***Использование*** — начинается ближе к концу *этапа оценки*, когда готовность изделия к эксплуатации проверена и может организовываться передача изделия на распространение (контрольная точка 8). *Этап* продолжается, пока изделие находится в действии и интенсивно эксплуатируется. Он связан с внедрением, обучением, настройкой и *сопровождением*, возможно, с модернизацией изделия. *Этап* заканчивается, когда разработчики прекращают систематическую деятельность по *сопровождению и поддержке* данного программного изделия (контрольная точка 10).
- функциональное

На протяжении *фаз* жизненного цикла разработчики выполняют различные организационные *производственные функции*, которые естественным образом группируются в классы родственных *функций.*

- ***Планирование*** — *функция*, которая должна выполняться с самого начала и до конца развития любого проекта. О содержании того, что должно планироваться на каждом из *этапов*, можно судить по наименованиям контрольных точек.
- ***Разработка***. Как и *планирование*, эта *функция* пронизывает весь проект. Содержание ее, т.е. то, что нужно разрабатывать, меняется, но в целом его можно охарактеризовать как получение рабочего продукта *этапа*.
- ***Обслуживание*** — *функция*, обеспечивающая максимально комфортную обстановку выполнения некоторой деятельности. Обслуживаемые виды деятельности меняются при переходе от одного *этапа* к другому, а могут распространяться и на несколько *этапов*.
- ***Выпуск документации***. Документация в проекте включает в себя не только техническое описание системы. Она рассматривается как полноправный вид рабочих продуктов, сопровождающий другие рабочие продукты: программы, диаграммы моделей системы и прочее.
- ***Испытания***. Поскольку испытывать можно лишь то, что готово для *испытания*, активизация *функции* начинается тогда, когда первичные требования к проекту сформулированы и нуждаются в проверке.
- ***Поддержка*** *использования* рабочих продуктов — это *функция*, выполнение которой необходимо в связи с передачей продукта в эксплуатацию.
- ***Сопровождение*** по Гантеру отличается от *поддержки* тем, что оно организуется для внешнего *использования* продуктов. Оно предполагает организацию *поддержки* и на этой основе выстраивает мероприятия, нацеленные на активную обратную связь с пользователями, чтобы можно было реагировать на их отклики, в том числе на изменение требований к продукту.

В проектах разработки сложных программных систем разнообразие функций и их интенсивность могут варьироваться. Модель Гантера учитывает это разнообразие и фазы жизненного цикла, отличаясь от более простых моделей, которые не предусматривают явное отображение итеративности. В модели Гантера перекрытие фаз может минимизировать возвраты к предыдущим этапам, но для полноценного описания итераций требуется дополнительное развитие модели.

Итеративное развитие в модели не описывается, и для отражения произвольных возвратов модель должна быть адаптирована. Один из подходов — расщепление линии жизненного цикла и матрицы интенсивностей функций для отражения уменьшения интенсивностей при возвратах. Это добавляет новое измерение, отражающее итеративный характер проекта.

Существует два варианта развития проекта с учетом итераций:

- приостановка основного процесса с последующим возобновлением
- параллельное ведение основного и дополнительного процессов с планированием расщеплений.

Первый вариант соответствует традиционному пониманию итеративности, второй — более гибкий, предполагает планирование итераций как часть управления проектом. Итеративность является ключевой при создании сложных систем и требует внимательного планирования и управления.

12. Способ быстрой разработки приложений (RAD): условия применения, стадии жизненного цикла, достоинства и недостатки.

Разработка спиральной модели жизненного цикла программного обеспечения и CASE технологий позволили сформулировать условия, выполнение которых сокращает сроки создания программного обеспечения.
Современная технологии проектирования, разработки и сопровождения программного обеспечения, должна отвечать следующим требованиям:

- поддержка полного жизненного цикла программного обеспечения;
- гарантированное достижение целей разработки с заданным качеством и в установленное время;
- возможность выполнения крупных проектов в виде подсистем, разрабатываемых группами исполнителей ограниченной численности (3-7 человек) с последующей интеграцией составных частей, и координации ведения общего проекта;
- минимальное время получения работоспособной системы;
- возможность управления конфигурацией проекта, ведения версий проекта и автоматического выпуска проектной документации по каждой версии;
- независимость выполняемых проектных решений от средств реализации (СУБД, операционных систем, языков и систем программирования);
- поддержка комплексом согласованных CASE-средств, обеспечивающих автоматизацию процессов, выполняемых на всех стадиях жизненного цикла.

Этим требованиям отвечает технология **RAD (Rapid Application Development - Быстрая разработка приложений)
**.
Эта технология ориентирована на максимально быстрое получение первых версий разрабатываемого программного обеспечения. **Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца**. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.
Она предусматривает выполнение следующих условий:

- ведение разработки небольшими группами разработчиков (3-7 человек), каждая из которых проектирует и реализует отдельные подсистемы проекта - позволяет улучшить управляемость проекта;
- использование итерационного подхода способствует уменьшению времени получения работоспособного прототипа;
- наличие четко проработанного графика цикла, рассчитанного не более чем на три месяца, существенно увеличивает эффективность работы.

Процесс разработки при этом делится на следующие этапы:

- анализ и планирование требований пользователей,
  формулируют наиболее приоритетные требования, что ограничивает масштаб проекта.
- проектирование,
  используя имеющиеся CASE-средства, детально описывают процессы системы, устанавливают требования разграничения доступа к данным и определяют
  состав необходимой документации.
  При этом для наиболее сложных процессов создают частичный прототип. По результатам анализа процессов определяют количество так называемых функциональных точек и принимают решение о количестве подсистем и, соответственно, команд, участвующих в разработке.
  Под функциональной точкой в технологии RAD понимают любой из следующих функциональных элементов разрабатываемой системы:
  • входной элемент приложения (входной документ или экранная форма);
  • выходной элемент приложения (отчет, документ или экранная форма);
  • запрос (пара «вопрос/ответ»);
  • логический файл (совокупность записей данных, используемых внутри приложения);
  • интерфейс приложения (совокупность записей данных, передаваемых другому приложению или получаемых от него).
  Нормы, рассчитанные исходя из экспертных оценок, для систем со значительной

повторяемостью кодов определяются следующим образом:
• менее 1 тыс. функциональных точек- 1 человек;
• от 1 до 4 тыс. функциональных точек - одна команда разработчиков;
• более 4 тыс. функциональных точек - одна команда на каждые 4 тыс. точек.

- реализация,
  выполняют итеративное построение реальной системы, причем при этом для контроля над выполнением требований к создаваемой системе привлекаются будущие пользователи. Части постепенно интегрируют в систему, причем при подключении каждой части выполняют тестирование. На завершающих этапах разработки определяют необходимость создания соответствующих баз данных, которые разрабатываются и подключаются к системе. Далее формулируют требования к аппаратным средствам, устанавливают способы увеличения производительности и завершают подготовку документации по проекту.
- внедрение.
  На этапе внедрения проводят обучение пользователей и осуществляют постепенный переход на новую систему, причем эксплуатация старой версии продолжается до полного внедрения новой системы.

Технология RAD хорошо зарекомендовала себя для относительно небольших проектов, разрабатываемых для конкретного заказчика. Такие системы не требуют высокого уровня планирования и жесткой дисциплины проектирования. Однако эта технология не применима для построения сложных расчетных программ, операционных систем или программ управления сложными объектами в реальном масштабе времени, т. е. программ с большим процентом уникального кода.
Не годится она и в случае создания приложений, от которых зависит безопасность людей, например, систем управления самолетами или атомными электростанциями, так как технология RAD предполагает, что первые несколько версий не будут полностью работоспособны, что в данном случае полностью исключается.

Преимущества:

- благодаря использованию мощных инструментальных средств, время цикла разработки для всего проекта можно сократить
- требуется меньшее количество специалистов, т.к. специалисты хорошо владеют предметной областью
- уменьшаются затраты и риск, связанный с соблюдением графика
- в состав каждого временного блока входит анализ, проектирование и внедрение фазы отделены от действий
- основное внимание переносится с документации на код, причем при этом справедлив принцип получаете то, что видите
- используются современные методы моделирования данных
+ достоинства структурной эволюционной модели быстрого прототипирования.

Недостатки RAD:

- пользователь должен всегда принимать участие на всех этапах разработке
- необходимо достаточное количество высококвалифицированных и хорошо обученных разработчиков
- использование модели может оказаться неудачным в случае, если отсутствуют пригодные для повторного использования компоненты
- жесткие временные ограничения
- существует риск, что работа над проектом никогда не будет завершена

13. **Метод и технология проектирования программного обеспечения, требования к технологии, формализация и автоматизация стадий и этапов жизненного цикла, стандартизация процесса проектирования и разработки: стандарт проектирования, стандарт оформления проектной документации, стандарт интерфейса пользователя, государственные стандарты, стандарты предприятия. Эволюция методов и средств программной инженерии.**

Технология создания ПО — упорядоченная совокупность взаимосвязанных **технологических процессов в рамках ЖЦ ПО.**

Современная технология проектирования должна обеспечивать:

1. Соответствие стандарту ISO/IEC 12207: 1995 (**поддержка всех процессов ЖЦ ПО**).
    1. управление требованиями;
    2. анализ и проектирование ПО;
    3. разработка ПО;
    4. эксплуатация;
    5. сопровождение;
    6. документирование;
    7. управление конфигурацией и изменениями;
    8. тестирование;
    9. управление проектом.
2. Гарантированное **достижение целей** разработки ПО в рамках установленного бюджета, с заданным качеством и в установленное время.
3. Возможность **декомпозиции проекта** на составные части
4. Минимальное время получения работоспособного ПО.
5. Независимость получаемых проектных решений от средств реализации ПО (СУБД, ОС, языков и систем программирования).
6. Поддержка комплексом согласованных CASE – средств, обеспечивающих автоматизацию процессов, выполняемых на всех стадиях ЖЦ ПО.

Метод проектирования ПО представляет собой организованную совокупность процессов создания **ряда моделей,** которые описывают различные аспекты разрабатываемой системы с использованием четко определенной нотации.

На более формальном уровне метод определяется как совокупность составляющих:

- ***Концепций и теоретических основ***. (структурный или объектно-ориентированный подход)
- ***Нотаций*** (DFD, ERD)
- ***Процедур*** (последовательность и правила построения моделей, критерии, используемые для оценки результатов).

Стандарт проектирования должен устанавливать:

- **Набор необходимых моделей** (диаграмм) на каждой стадии проектирования и степень их детализации.
- **Правила фиксации проектных решений на диаграммах**, в том числе правила именования объектов (включая соглашения по терминологии), набор атрибутов для всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм (включая требования к форме и размерам объектов) и т.д.
- **Требования к конфигурации рабочих мест** разработчиков, включая настройки ОС, настройки CASE – средств и т.д.
- **Механизм обеспечения совместной работы** над проектом, в том числе правила интеграции подсистем проекта, правила поддержания проекта в одинаковом для всех разработчиков состоянии (регламент обмена проектной информацией, механизм фиксации общих объектов и т.д.), правила анализа проектных решений на непротиворечивость и т.д.

Стандарт оформления проектной документации. Он должен устанавливать:

- **комплектность, состав и структуру** документации на каждой стадии проектирования (в соответствии со стандартом ГОСТ Р ИСО 9127 – 94 «Системы обработки информации. Документация пользователя и информация на упаковке потребительских программных пакетов»);

- требования к **оформлению** документации (включая требования к содержанию разделов, подразделов, пунктов, таблиц и т.д.);
- правила **подготовки**, рассмотрения, согласования и утверждения документации с указанием предельных сроков на каждой стадии;
- требования к **настройке** издательской системы, используемой в качестве встроенного средства подготовки документации;
- требования к **настройке CASE – средств** для обеспечения подготовки документации в соответствии с установленными правилами.

Стандарт интерфейса конечного пользователя с системой. Он должен регламентировать:

- Правила **оформления экранов** (шрифты и цветовая палитра), состав и расположение окон и элементов управления.
- Правила использования **клавиатуры и мыши.**
- Правила оформления текстов **помощи**.
- Перечень с**тандартных сообщений.**
- Правила обработки **реакций пользователя**.

14. Эффективность технологии проектирования программного обеспечения: критерии оценки технологии проектирования – функциональные, конструктивные, основные затраты в жизненном цикле, распределение затрат на разработку, длительность разработки программного обеспечения.

Эффективность технологии проектирования программного обеспечения: критерии оценки технологии проектирования – функциональные, конструктивные, основные затраты в жизненном цикле, распределение затрат на разработку, длительность разработки программного обеспечения.
(я хз как будто не по теме, но это единственное что я нашел в учебнике и у беляева тоже оно)
Традиционно эффективными считают программы, требующие **минимального времени** выполнения и/или **минимального объема оперативной памяти**. Особые требования к эффективности ПО предъявляют при наличии ограничений.
Для уменьшения времени выполнения чекаем **циклические** фрагменты с большим количеством повторений: экономия времени выполнения одной итерации цикла будет умножена на количество итераций.
Способы снижения временных затрат приводят к увеличению емкостных и, наоборот, уменьшение объема памяти может потребовать дополнительного времени на обработку.
Не следует «платить» за увеличение эффективности снижением технологичности разрабатываемого программного обеспечения. Исключения возможны лишь при очень жестких требованиях и наличии соответствующего контроля за качеством. Частично проблему эффективности программ решают за программиста компиляторы.
Средства оптимизации, используемые компиляторами, делят на две группы:

- *машинно-зависимые*, т. е. **ориентированные на конкретный машинный язык**, выполняют оптимизацию кодов на уровне машинных команд, например, исключение лишних пересылок, использование более эффективных команд и т. п.;
- *машинно-независимые* выполняют **оптимизацию на уровне входного язык**а, например, вынесение вычислений константных (независящих от индекса цикла) выражений из циклов и т. п.

Естественно, нельзя вмешаться в работу компилятора, но существует много возможностей оптимизации программы на уровне команд.
Способы экономии памяти.
Следует обращать особое внимание на выделение памяти под **данные структурных типов** (массивов, записей, объектов и т. п.).
Следует выбирать алгоритмы обработки, ***не требующие дублирования исходных*** *данных структурных типов в процессе обработки.* Примером могут служить алгоритмы сортировки массивов, выполняющие операцию в заданном массиве, например, хорошо известная сортировка методом «пузырька».
Если в программе необходимы большие **массивы**, используемые ограниченное время, то их можно размещать в динамической памяти и удалять при завершении обработки.
Также следует помнить, что при передаче структурных данных в подпрограмму **«по значению»** копии этих данных размещаются в стеке. Избежать копирования иногда удается, если передавать данные **«по ссылке»**, но как неизменяемые (описанные const). В последнем случае в стеке размещается только адрес данных.
Способы уменьшения времени выполнения. Для уменьшения времени выполнения в первую очередь необходимо анализировать циклические участки программы с большим количеством повторений. При их написании необходимо по возможности:

- выносить вычисление **константных выражений из циклов;**
- избегать **«длинных» операций умножения** и деления, заменяя их сложением, вычитанием и сдвигами;
- минимизировать **преобразования типов** в выражениях;
- оптимизировать запись условных выражений - **исключать лишние проверки;**
- исключать многократные **обращения к элементам массивов по индексам** (особенно многомерных)- первый раз прочитав из памяти элемент массива, следует запомнить его в скалярной переменной и использовать в нужных местах;
- избегать использования различных типов в выражении и т. п.

15. **Оценка качества процессов создания программного обеспечения: международные стандарты серии ISO 9000, CMM, SPICE.**

Серия стандартов ISO 9000.

В серии ISO 9000 сформулированы необходимые условия **для достижения некоторого минимального уровня организации процесса**, но не дается никаких рекомендаций по дальнейшему совершенствованию процессов.

CMM

CMM. Представляет собой совокупность критериев **оценки зрелости организации-разработчика** и рецептов улучшения существующих процессов. Изначально CMM разрабатывалась и развивалась как методика, позволяющая крупным правительственным организациям США выбирать наилучших поставщиков программного обеспечения.

CMM определяет пять уровней зрелости организаций-разработчиков, причем каждый следующий уровень включает в себя все ключевые характеристики предыдущих.

1. *Начальный уровень* - описан в стандарте в качестве основы для сравнения со следующими уровнями. На предприятии такого уровня организации не существует стабильных условий для создания качественного ПО. Результат любого проекта целиком и полностью зависит от **личных качеств менеджера и опыта программистов**, причем успех в одном проекте может быть повторен только в случае назначения тех же менеджеров и программистов на следующий проект. Более того, если эти менеджеры или программисты уходят с предприятия, то резко снижается качество производимых программных продуктов. **В стрессовых ситуациях процесс разработки сводится к написанию кода и его минимальному тестированию.**
2. *Повторяемый уровень* - на предприятии внедрены ***технологии управления проектами*.** При этом планирование и управление проектами основывается на накопленном опыте, существуют **стандарты на разрабатываемое ПО** (причем обеспечивается следование этим стандартам) и специальная ***группа обеспечения качества*.** В случае необходимости организация может взаимодействовать с субподрядчиками. **В критических условиях процесс имеет тенденцию скатываться на начальный уровень.**
3. *Определенный уровень* - характеризуется тем, что стандартный процесс создания и сопровождения ПО **полностью документирован** (включая и разработку ПО, и управление проектами). Подразумевается, что в процессе стандартизации происходит переход на наиболее эффективные практики и технологии. Для создания и поддержания подобного стандарта в организации должна быть создана специальная группа. Наконец, обязательным условием для достижения данного уровня является наличие на предприятии **программы постоянного *повышения квалификации и обучения сотрудников*.** Начиная с этого уровня, организация **перестает зависеть от качеств конкретных разработчиков, и процесс не имеет тенденции скатываться на уровень ниже в стрессовых ситуациях.**
4. *Управляемый уровень* - в организации устанавливаются ***количественные показатели качества*__,** как на программные продукты, так и на процесс в целом. Таким образом, более совершенное управление проектами достигается за счет уменьшения отклонений различных показателей проекта. При этом осмысленные вариации в производительности процесса можно отличить от случайных вариаций (шума), особенно в хорошо освоенных областях.
5. *Оптимизирующий уровень* - характеризуется тем, что **мероприятия по улучшению применяются для оценки эффективности ввода новых технологий.** Основной задачей всей организации на этом уровне является **постоянное улучшение существующих процессов.** При этом улучшение процессов в идеале должно помогать предупреждать возможные ошибки или дефекты. Кроме того, должны вестись работы по **уменьшению стоимости разработки программного обеспечения**, например, с помощью создания и повторного использования компонентов.

- ==Упрощенный вариант с лекции==
  1. Начальный уровень - 70% организаций
  Не существует стабильных условий для создания ПО. Результат полностью зависит от личных качеств работников. В стрессовых разработка сводиться к написанию кода и его минимальному тестированию.
  2. Повторяемый - внедрены технологии управления проектами. Используются стандарты разработки ПО. Есть группа проверки качества

3. Определенный - процесс создания ПО полностью документирован. На таких предприятиях существует программа постоянного повышения квалификации. Организация перестает зависеть от качества персонала.
4. Управляемый - Вводятся количественные показатели качества.
5. Оптимизирующий уровень - самые пиздатые

Сертификационная оценка соответствия всех ключевых областей проводится по 10-балльной шкале. Для успешной квалификации данной ключевой области необходимо набрать не менее 6 баллов. Оценка ключевой области осуществляется по следующим показателям:

- заинтересованность руководства в данной области
- насколько широко данная область применяется в организации
- успешность использования данной области на практике

В принципе, можно сертифицировать только один процесс или подразделение организации, например, подразделение разработки ПО компании.

SPICE.

Стандарт SPICE унаследовал многие черты более ранних стандартов, в том числе ISO 9001 и CMM. Больше всего SPICE напоминает CMM. Точно так же, как и в CMM, основной задачей организации является п**остоянное улучшение процесса разработки ПО.** Кроме того, в SPICE тоже используется схема с различными уровнями возможностей (в SPICE определено 6 различных уровней), но эти уровни применяются не только к организации в целом, но **и к отдельно взятым процессам.**

В основе стандарта лежит *оценка процессов*. Эта оценка выполняется путем сравнения процесса разработки ПО с описанной в стандарте моделью. Помогает определить сильные и слабые стороны процесса, а также внутренние **риски.**. Это помогает оценить эффективность процессов, определить причины ухудшения качества и связанные с этим издержки во времени или стоимости.

Затем выполняется ***определение возможностей*** **улучшения** ***процесса***__. В результате в организации может появиться понимание необходимости *улучшения* того или иного *процесса*. К этому моменту цели совершенствования процесса уже четко сформулированы и остается только техническая реализация поставленных задач. После этого весь цикл работ начинается сначала. Следует иметь в виду, что построение «более зрелого» процесса разработки не обязательно обеспечивает создание более качественного ПО.

Использование формальных моделей и методов позволяет создавать **понятные, непротиворечивые спецификации** на разрабатываемое ПО. Конечно, внедрение таких методов имеет смысл, хотя оно весьма дорого и трудоемко, а возможности их применения весьма ограничены. Основная же проблема - проблема сложности разрабатываемого программного обеспечения с совершенствованием процессов разработки пока не разрешена.

Создание программного обеспечения по-прежнему предъявляет повышенные требования к квалификации тех, кто этим занимается: проектировщикам программного обеспечения и непосредственно программистам.

16. **Качество программного обеспечения, управление качеством, общие характеристики качества программного обеспечения: функциональность, надежность, удобство использования, эффективность, сопровождаемость, мобильность; критерии качества, ранжированные по фазам жизненного цикла, метрики характеристик программного обеспечения.Качество - сложный, комплексный показатель, позволяющий оценивать важнейшие свойства программы в пределах ЖЦ Оценка качества осуществляется с помощью критериев. (==С лекции==)**

Качество программного обеспечения — характеристика программного обеспечения (ПО) как степени его соответствия требованиям. При этом требования могут трактоваться довольно широко, что порождает целый ряд независимых определений понятия. Чаще всего используется определение ISO 9001, согласно которому **качество есть «степень соответствия присущих характеристик требованиям».**

Фактор качества ПО — это нефункциональное требование к программе, которое обычно не описывается в договоре с заказчиком, но, тем не менее, является желательным требованием, повышающим качество программы.

Критерии качества:

Этап проектирования:

- сложность программы
- корректность программы
- трудоемкость разработки (в человеко-месяцах)

Этап эксплуатации:

- функциональная сложность
- надежность
- эффективность
- трудоемкость эксплуатации

Этап сопровождения:

- удобство модернизации
- мобильность
- трудоемкость изучения и модификации

Метрика качества программ - система измерений качества программ. Эти измерения могут проводиться на уровне критериев качества программ или на уровне отдельных характеристик качества.

МЕТРИЧЕСКИЕ ШКАЛЫ

В зависимости от характеристик и особенностей применяемых метрик им ставятся в соответствие различные измерительные шкалы.

Номинальной шкале

соответствуют метрики, классифицирующие программы на типы по признаку наличия или отсутствия **некоторой характеристики** без учета градаций.

Порядковой шкале

соответствуют метрики, п**озволяющие ранжировать** некоторое характеристики путем сравнения с опорными значениями, т.е. измерение по этой шкале фактически определяет взаимное положение конкретных программ.

Интервальной шкале

соответствуют метрики, которые показывают не только относительное положение программ, но и то, **как далеко они отстоят друг от друга.**

Относительной шкале

соответствуют метрики, позволяющие не только расположить программы определенным образом и оценить их положение относительно друг друга, но и определить, **как далеко оценки отстоят от границы**, начиная с которой характеристика может быть измерена.

МЕТРИКИ СЛОЖНОСТИ ПРОГРАММ

При оценке сложности программ, как правило, выделяют три основные группы метрик:

- метрики **размера** программ
- метрики сложности потока **управления** программ
- метрики сложности потока **данных** программ

17. Понятийный аппарат метрической теории программ – принципы количественного анализа качества объектов с расплывчатыми свойствами.

Качество - сложный, комплексный показатель, позволяющий оценивать важнейшие свойства программы в пределах ЖЦ Оценка качества осуществляется с помощью критериев.

Характеристика качества программы - понятие, отражающее отдельные факторы, влияющие на качество программ и поддающиеся измерению.

Критерий качества - численный показатель, характеризующий степень, в которой программе присуще оцениваемое свойство.

Критерий должен:

- численно характеризовать основную **целевую функцию** программы;
- обеспечивать возможность **определения затрат**, необходимых для достижения требуемого уровня качества, а также степени влияния на показатель качества различных внешних факторов;
- быть по возможности **простым**, хорошо измеримым и иметь малую дисперсию

Для измерения характеристик и критериев качества используют метрики. ==(Я хз, это вроде тоже сюда, я этот билет вообще не понял)==

Метрика качества программ - система измерений качества программ. Эти измерения могут проводиться на уровне критериев качества программ или на уровне отдельных характеристик качества.

МЕТРИЧЕСКИЕ ШКАЛЫ

В зависимости от характеристик и особенностей применяемых метрик им ставятся в соответствие различные измерительные шкалы.

Номинальной шкале

соответствуют метрики, классифицирующие программы на типы по признаку наличия или отсутствия **некоторой характеристики** без учета градаций.

Порядковой шкале

соответствуют метрики, п**озволяющие ранжировать** некоторое характеристики путем сравнения с опорными значениями, т.е. измерение по этой шкале фактически определяет взаимное положение конкретных программ.

Интервальной шкале

соответствуют метрики, которые показывают не только относительное положение программ, но и то, **как далеко они отстоят друг от друга.**

Относительной шкале

соответствуют метрики, позволяющие не только расположить программы определенным образом и оценить их положение относительно друг друга, но и определить, **как далеко оценки отстоят от границы**, начиная с которой характеристика может быть измерена.

МЕТРИКИ СЛОЖНОСТИ ПРОГРАММ

При оценке сложности программ, как правило, выделяют три основные группы метрик:

- метрики **размера** программ
- метрики сложности потока **управления** программ
- метрики сложности потока **данных** программ

18. Модель и метрики оценки сложности Боэма.

Модель оценки Боэма – оценка сложности, качества модуля. Учитывает объем кода, и сложность модуля. Определяется числом строк.
В данной модели для вывода формул использовался статистический подход — учитывались реальные результаты огромного количества проектов.
К – количество строк программного кода
Е – трудоемкость (чел/мес)
Т – продолжительность разработки (мес)
к прикладному ПО
$E = 2,4 * K^{1.05}-$
$T = 2.5 * E^{0,38}$
к системному ПО
$E = 3,6 * K^{1.2}$
$T = 2.5 * E^{0,32} $

19. Модель и метрики оценки сложности Холстэда.

Учитывает реализационный контекст(объемные показатели программного кода).
индикаторы:
Реляционный контекст
$n = n1+n2 $
n1, n2 - количество различных операторов и операндов соответственно
Длина исходного кода
$N = N1+N2$
N1, N2 - количество всех операторов и операндов соответственно
объём
$V = N * log_2 n $
трудоемкость (чел/мес)
$E ≈ V^2$
срок разработки (мес)
$T ≈ V^2$
наибольшее вероятностное количество ошибок в модуле (прогнозируемое количество ошибок)
$B ≈ E ^{2/3} $

Применять данную модель можно при известном тексте модуля.

```
int a,b,c;

a=1;

b=1;

c=a+b;
```

n1=3 N1=5
n2=3 N2=8
n=6 N=13

20. Модель и метрики оценки сложности Мак-Кейба (основанные на потоковых графах).

С помощью данной модели можно определить логическую сложность программы.
Основывается на потоковых графах модуля. Программного кода может и не быть, но известен алгоритм или код описанный на псевдоязыке.

Вершины графа – операторы, стрелка – переход от одного оператора к другому.
V(M) – логическая сложность модуля;
$V(M) = G(M)+1 = K2-K1+2;$
M– имя модуля;
G(M) – количество базисных путей в графе;
K2 – количество вершин;
K1– количество дуг;
!Ершов давал только V = k2-k1+2!
Case считается как один оператор.

21. Модель и метрики, основанные на информационных потоках.

Модель, основанная на информационных потоках, проходящих через интерфейс модуля. Она позволяет оценить информационную сложность модуля.

$J= m(F^+ * F^-)^2$

m – метрика (модель) Мак-Кейба или Холстэда.

$F+$ - количество потоков управления и структур данных которые приходят в модуль

$F-$ - количество потоков управления и структур данных которые уходят из модуля

22. Методы оценки качества программного обеспечения: анкетирование, рабочие списки, контрольные задачи, метрики. Государственные стандарты в области оценки качества программного обеспечения.

1. Метрики

Метод основанный на метриках

Тут не понятно что за метрики, нашел в тимсе файл “Метрики качества ПО”, оно основывается на Холстетде. Там еще к каждому пункту по формуле, но наврятли оно нам надо

1. Длина текста программы
   Общее колво операторов и операндов
2. Прогнозируемая длина текста
   пропорциональна логарифму по2 размера словаря
3. Объем текста программы
   Колво символов, необходимых для записи всех операторов и операндов
4. Потенциальный объем текста
   Количество символов в наиболее компактном тексте программы
5. Уровень качества программирования
   Степень расширения текста программы относительно ее потенциального объема
6. Косвенная оценка уровня качества программирования
   Пропорциональность уровня качества компактности представления операторов и операндов при минимальном количестве типов операторов
7. Уровень языка программирования
8. Интеллектуальное содержание программы
   Пропорциональность степени информативности программы ее образу и уровню качества программирования
9. Интеллектуальная сложность программы
   Пропорциональность затрат интеллектуальных усилий на разработку.

2. Анкетирование

   В основе лежит применение специальных анкет из 100-200 вопросов. С помощью этих анкет создаются атласы ПО, которые используются для качественной и количественной оценки. Пользователи сравнивают разные ПО и заполняют анкеты. Данный метод следует применять для ПО большого объема и широкого применения.

3. Рабочие списки

   Позволяет оценить характеристики ПО на следующих уровнях ЖЦ:

   - уровень анализа систем;
   - уровень проекта систем;
   - уровень проекта модуля;
   - уровень программы модуля;

4. Метод контрольных задач

   Используются для сравнения различных частей программ, типа компилятора.
   Оценивается не алгоритм, а метод его реализации.
   Решение задач малого и среднего объема

Государственные стандарты

1. ГОСТ 28195-89 «Оценка программных средств»
2. Международные стандарты серии ISO 9000 (ISO 9000 - ISO 9004)
   Серия стандартов ISO 9000. В серии ISO 9000 сформулированы необходимые условия **для достижения некоторого минимального уровня организации процесса**, но не дается никаких рекомендаций по дальнейшему совершенствованию процессов

23. Модули, сцепление и связность-критерии независимости модулей, библиотеки ресурсов.

Модуль - независимая автономно компилируемая единица

Первоначально, когда размер программ был сравнительно невелик, и все подпрограммы компилировались отдельно, под модулем понималась подпрограмма, т. е. ***последовательность связанных фрагментов программы, обращение к которой выполняется по имени***__. Со временем, когда размер программ значительно вырос, и появилась возможность создавать библиотеки ресурсов: констант, переменных, описаний типов, классов и подпрограмм, термин «модуль» стал использоваться и в смысле ***автономно компилируемый набор программных ресурсов.***

Сцепление модулей

Является ***мерой взаимозависимости модулей***, которая определяет, насколько хорошо модули отделены друг от друга. Модули независимы, если каждый из них не содержит о другом никакой информации. Чем больше информации о других модулях хранит модуль, тем больше он с ними сцеплен.

1. По данным (Слабое) ✅

модули обмениваются данными, представленными **скалярными значениями**. Круто при небольшом количестве передаваемых параметров.

```
int fmax(int a, int b)
```

2. По образцу✅

модули обмениваются данными, объединенными в **структуры**. Этот тип хуже, чем предыдущий , так как конкретные передаваемые данные «**спрятаны**» в структуры, и потому **уменьшается «прозрачность»** связи между модулями. Кроме того, при изменении структуры передаваемых данных необходимо **модифицировать** все использующие ее модули.

```
int fmax(int[]a, int b)
```

3. По управлению ✅

модуль посылает другому некоторый информационный объект (**флаг**), предназначенный для управления внутренней логикой модуля. **Снижает наглядность** взаимодействия модулей

int fmax(int a, int b, bool c) где c влияет на режимы работы функции

4. По общей области данных ⛔

Например работа с глобальным массивом

Этот тип сцепления считается недопустимым, поскольку:

- программы, очень сложны для понимания;
- ошибка одного модуля, приводящая к изменению общих данных, может проявиться при выполнении другого модуля;
- маленькая гибкость из за конкретных имен.

5. По содержимому⛔

один модуль содержит обращения к внутренним компонентам другого, что противоречит блочно-иерархическому подходу.

Volatile — ключевое слово языков C/C++, которое информирует компилятор о том, что значение переменной может меняться из вне и что компилятор не будет оптимизировать эту переменную

Первые три сцепления являются допустимыми

Связность элементов внутри модуля

Связность - **мера прочности соединения** функциональных и информационных объектов внутри одного модуля. Если сцепление характеризует качество отделения модулей, то связность характеризует степень взаимосвязи элементов

Любой модуль внутри имеет два типа элементов - операторы (операционные) и переменные (информационные)

Различают следующие виды связности (в порядке убывания уровня):

1. Функциональная✅

все объекты модуля предназначены для выполнения **одной функции**

2. Последовательная ✅

Выход одной функции служит исходными данными для другой функции. Можно разбить на части.

3. Информационная✅

функции, обрабатывающие одни и те же данные.

4. Процедурная ⛔
В модуль объединяются две и более функции являющиеся альтернативными частями.
5. Временная⛔
Разные функции. Должны начинать работу примерно в одно и тоже время.
6. Логическая⛔
Разные функции, имеющие одинаковый тип (например, вычисление математических функций).
7. Случайная⛔
связь между элементами мала или отсутствует.
==Первые три вида связности допустимы==

Библиотеки ресурсов.

- **Библиотеки подпрограмм** реализуют функции, близкие по назначению, например, библиотека графического вывода информации.
  Связность подпрограмм между собой в такой библиотеке - логическая
  связность самих подпрограмм - функциональная, так как каждая из них обычно реализует одну функцию.
- **Библиотеки классов** реализуют близкие по назначению классы.
  Связность элементов класса -информационная
  Связность классов между собой может быть функциональной - для родственных или ассоциированных классов и логической - для остальных.

В качестве средства улучшения технологических характеристик библиотек ресурсов в настоящее время широко используют разделение тела модуля на интерфейсную часть и область реализации.
Каждый модуль должен иметь заголовок - три строки комментариев там :

1. Название модуля
2. Назначение
3. Вх/вых данные
4. Список используемых модулей
5. Краткое описание алгоритма
6. Фамилия автора
7. Идентификатор

#СтруктурныйПодход #ОбъектныйПодход

24. Нисходящий и восходящий подход к разработке программного обеспечения, средства описания структурных алгоритмов: базовые и дополнительные алгоритмические структуры, псевдокоды, Flow-формы, диаграммы Насси-Шнейдермана.

При проектировании, реализации и тестировании компонентов структурной иерархии, полученной при декомпозиции, применяют два подхода:

Восходящий подход.

Сначала реализуются модули нижнего уровня, они отвечают за обработку информации, за ввод информации.

+ по мере реализации нижних модулей есть время для отладки и тестирования;
− при сборке в единое целое проявляются ошибки интерфейсов;

-если интерфейс может не устраивать и придется переделывать большой объем работы.

Для массового программирования такой подход не применим.

Нисходящий подход.

Вначале проектируют модули верхних уровней, затем следующих и так далее до самых нижних уровней. Есть «заглушки» для не деланых модулей - программы ни чего не выполняющие, что позволяет тестировать и отлаживать уже реализованную часть.

Нисходящий подход может быть реализован 3 способами:

1. *Иерархический метод*
   предполагает выполнение разработки **строго по уровням.** Основной проблемой данного метода является **большое количество достаточно сложных заглушек**. Кроме того, при использовании данного метода **основная масса модулей разрабатывается и реализуется в конце работы над проектом**, что затрудняет распределение человеческих ресурсов.
   - Количество модулей все растет, поэтому нужно будет постепенно увеличивать число людей ==?? оч странно, но так сказал ерш==
2. Операционный метод
   связывает **последовательность создания с выполнением при запуске программы.** Применение метода усложняется тем, что порядок выполнения модулей **может зависеть от данных**. Кроме того, модули вывода результатов, несмотря на то, что они вызываются последними, должны разрабатываться одними из первых, чтобы не проектировать сложную заглушку, обеспечивающую вывод результатов при тестировании.
3. Комбинированный метод *==**- про него ерш не рассказывал***==
   Усложненный метод, в котором порядок создания определятся по большому колву параметров, например наличие необходимых ресурсов, готовность вспомогательных модулей и тп

Базовые алгоритмические структуры:

- *следование* обозначает последовательное выполнение действий;
- *ветвление (**if**)* соответствует выбору одного из двух вариантов действий;
- *цикл-пока(**while**)* определяет повторение действий, пока не будет нарушено некоторое условие, выполнение которого проверяется в начале цикла.

Дополнительные алгоритмические структуры (можно составить из базовых):

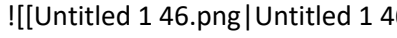
- **swith** обозначает выбор одного варианта из нескольких в зависимости от значения некоторой величины;
- *цикл-__**dowhile*** обозначает повторение некоторых действий до выполнения заданного условия, проверка которого осуществляется после выполнения действий в цикле;
- *цикл с заданным числом повторений* (**for**) - обозначает повторение некоторых действий указанное количество раз.

Псевдокод — компактный, зачастую неформальный язык описания алгоритмов, использующий ключевые слова языков программирования, но опускающий несущественные для понимания алгоритма подробности и специфический синтаксис. Предназначен для представления алгоритма человеку, а не для компьютерной трансляции и последующего исполнения программы.

Flow-формы

Графическая нотация описания структурных алгоритмов, которая иллюстрирует вложенность структур. **Каждый символ Flow-формы имеет вид прямоугольника и может быть вписан в любой внутренний прямоугольник любого другого символа.** 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 определяется длиной вписанного в него текста и размерами вложенных прямоугольников.

Диаграммы Насси-Шнейдермана.
Диаграммы Насси-Шнейдермана являются развитием Flow-форм. Область обозначения условий и вариантов ветвления изображают в виде **треугольников**. Такое обозначение обеспечивает **большую наглядность** представления алгоритма.

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

25. **Программирование с защитой от ошибок: проверка выполнения операций, контроль промежуточных результатов, снижение погрешностей результатов, обработка исключений; сквозной структурный контроль.**

Многие ошибки отлавливаются еще на этапе **компиляции и компоновке программе**, но не все. Те, что не удалось автоматически найти могут привести к выдаче системного сообщения об ошибке, «зависанию» компьютера и получению неверных результатов.

Программирование с защитой от ошибок (защитное программирование) - это программирование, при котором применяют специальные приемы раннего обнаружения и нейтрализации ошибок.

Разработчику следует:

- правильность выполнения операций ввода-вывода;
  Причинами неверного определения исходных данных могут являться, как внутренние ошибки-ошибки устройств ввода-вывода или программного обеспечения, так и внешние ошибки - ошибки пользователя.
  - *ошибки передачи* _(_аппаратные средства в следствии неисправностей искажает данные)
  - *ошибки преобразования (программа неверно преобразует данные из входного формата во внутренний)*
  - *ошибка перезаписи (пользователь ошибается при вводе данных, лишний символ)*
  - *ошибки данных (пользователь неверно вводит данные)*
- допустимость промежуточных результатов.
  Проверки промежуточных результатов позволяют снизить вероятность позднего проявления не только ошибок неверного определения данных, но и некоторых **ошибок кодирования и проектирования.**

Обработка исключений. Поскольку полный контроль данных на входе и в процессе вычислений, как правило, невозможен, следует предусматривать перехват обработки аварийных ситуаций.

Сквозной контроль - это совокупность технологических операций контроля. Он позволяет обеспечить как можно более раннее обнаружение ошибок в процессе разработки.

- сквозной - выполнение контроля на всех этапах разработки
- структурный - наличие четких рекомендаций по выполнению контролирующих операций на каждом этапе.

Сквозной структурный контроль должен выполняться на специальных контрольных сессиях, в которых, помимо разработчиков, могут участвовал специально приглашенные эксперты.

Одна из первых таких сессий должна произойти на этапе разработке спецификаций, при этом желательно присутствие заказчика на этой встрече.

26. **Разработка и анализ требований к программному обеспечению: определение целей проектируемого программного обеспечения, определение целей управления проектом; техническое задание и спецификации программного обеспечения; функциональные и нефункциональные требования.**

Этап постановки задачи - один из наиболее ответственных этапов создания программного продукта. На этом этапе формулируют основные требования к разрабатываемому программному обеспечению. По *назначению* все программы можно разделить на несколько групп:

- Эксплуатационные требования определяют некоторые **характеристики разрабатываемого программного обеспечения**, проявляемые в процессе его функционирования. К таким характеристикам относят:
  - *правильность* - функционирование в соответствии с техническим заданием;
  - *универсальность* - обеспечение правильной работы при любых допустимых данных и защиты от неправильных данных;
  - *надежность* (помехозащищенность) - обеспечение полной повторяемости результатов, т. е. обеспечение их правильности при наличии различного рода сбоев;
  - *проверяемость* - возможность проверки получаемых результатов;
  - *точность результатов* - обеспечение погрешности результатов не выше заданной;
  - *защищенность* - обеспечение конфиденциальности информации;
  - *программная совместимость* - возможность совместного функционирования с другим программным обеспечением;
  - *аппаратная совместимость* - возможность совместного функционирования с некоторым оборудованием;
  - *эффективность* - использование минимально возможного количества ресурсов технических средств, например, времени микропроцессора или объема оперативной памяти;
  - *адаптируемость* - возможность быстрой модификации с целью приспособления к изменяющимся условиям функционирования;
  - *повторная входимость* - возможность повторного выполнения без перезагрузки с диска;
  - *реентерабельность* - возможность «параллельного» использования несколькими процессами.

Техническое задание представляет собой документ, в котором сформулированы основные цели разработки, требования к программному продукту, определены сроки и этапы разработки и регламентирован процесс приемно-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.

Требования к программе или программному изделию должен включать 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.

Наиболее важным из перечисленных выше является подраздел
Требования к функциональным характеристикам.

В этом разделе должны быть перечислены выполняемые функции и описаны состав, характеристики и формы представления исходных данных и результатов. В этом же разделе при необходимости указывают критерии эффективности: максимально допустимое время ответа системы, максимальный объем используемой оперативной и/или внешней памяти и др.

27. Технологические требования: выбор архитектуры ПО, выбор типа пользовательского интерфейса, выбор подхода к разработке, выбор языка и среды программирования.

Технологические требования- выбор архитектуры ПО, выбор типа пользовательского интерфейса, выбор подхода к разработке, выбор языка и среды программирования

Архитектурой программного обеспечения

называют совокупность базовых концепций (принципов) его построения.

Различают:

•

однопользовательскую архитектуру, при которой программное обеспечение рассчитано на одного пользователя, работающего за персональным компьютером;

• программы - *для решения конкретной задачи*;

• пакеты программ -

задачи некоторой прикладной области;

• программные комплексы -

совокупность программ, которые решают небольшой класс сложных задач одной области;

• программные системы -

решают широкий класс задач.

• многопользовательскую архитектуру, которая рассчитана на работу в локальной или глобальной сети.

Многопользовательские программные системы в отличие от обычных программных систем должны организовывать *сетевое взаимодействие отдельных компонентов программного обеспечения*, что еще усложняет процесс его разработки

Выбор типов пользовательского интерфейса:

Различают четыре типа пользовательских интерфейсов:

•

примитивные - реализуют единственный сценарий работы, например, ввод данных - обработка - вывод результатов;

•

меню - реализуют множество сценариев работы, операции которых организованы в иерархические структуры, например, «вставка»: «вставка файла», «вставка символа» и т. д.;

•

со свободной навигацией - реализуют множество сценариев, операции которых не привязаны к уровням иерархии, и предполагают определение множества возможных операций на конкретном шаге работы; интерфейсы данной формы в основном используют Windows-приложения;

•

прямого манипулирования - реализуют множество сценариев, представленных в операциях над объектами, основные операции инициируются перемещением пиктограмм объектов мышью, данная форма реализована в интерфейсе самой операционной системы Windows альтернативно интерфейсу со свободной навигацией.

Выбор подхода: Если выбран интерфейс со свободной навигацией или прямого манипулирования, то, как указывалось выше, это практически однозначно предполагает использование событийного программирования и объектного подхода, так как современные среды визуального программирования, такие как Visual C++, Delphi, Builder C++ и им подобные, предоставляют интерфейсные компоненты именно в виде объектов библиотечных классов.

Примитивный интерфейс и интерфейс типа меню совместимы как со структурным, так и с объектным подходами

к разработке. Поэтому выбор подхода осуществляют с использованием дополнительной информации.

Выбор языка программирования:

Язык может быть определен:

• организацией, ведущей разработку; например, если фирма владеет лицензионным вариантом

C++ Builder, то она будет вести разработки преимущественно в данной среде;
• программистом, который по возможности всегда будет использовать хорошо знакомый язык;
• устоявшимся мнением («все разработки подобного рода должны выполняться на C++ или на Java или на ...») и т. п,

Если же все-таки выбор языка реально возможен, то нужно иметь в виду, что все существующие языки программирования можно разделить на следующие группы:
• универсальные языки высокого уровня;
• специализированные языки разработчика программного обеспечения (языки баз данных);
• специализированные языки пользователя; (узкая направленность, обычно не используется разработчиками ПО)
• языки низкого уровня.

Средой программирования называют программный комплекс, который включает специализированный текстовый редактор, встроенные компилятор, компоновщик, отладчик, справочную систему и другие программы, использование которых упрощает процесс написания и отладки программ.

28. Планирование процесса проектирования, виды планов: календарный, индивидуальный, сетевой график разработки и проектирования программного обеспечения.

Планирование процесса проектирования является ключевым этапом в разработке программного обеспечения. Оно помогает команде управлять временем, ресурсами и обеспечивать координацию между участниками проекта. Существует несколько видов планов, каждый из которых имеет свои особенности и применяется в зависимости от потребностей проекта:

1. Календарный план:
    - Отображает расписание работы над проектом во времени.
    - Включает в себя список задач, их продолжительность и запланированное время выполнения.
    - Может быть представлен в виде таблицы или диаграммы Ганта, где по горизонтальной оси отложено время, а по вертикальной — задачи.
2. Индивидуальный план:
    - Разрабатывается для каждого члена команды отдельно.
    - Определяет индивидуальные задачи, сроки и ответственность.
    - Помогает каждому участнику понять свои обязанности и сроки их выполнения.
3. Сетевой график:
    - Использует методы сетевого планирования, такие как CPM (Critical Path Method — метод критического пути) или PERT (Program Evaluation and Review Technique — метод оценки и анализа программ).
    - Представляет собой график, состоящий из узлов (работ, этапов проекта) и связей между ними, которые показывают зависимости.
    - Позволяет определить критический путь проекта, наименьшее время его выполнения и возможные задержки.

    ![[Untitled 94.png|Untitled 94.png]]
4. График разработки и проектирования программного обеспечения:
    - Комплексный план, который включает в себя все этапы разработки ПО: от сбора требований до тестирования и внедрения.

    - Может сочетать в себе элементы календарного планирования, индивидуальных планов и сетевых графиков.
    - Помогает управлять процессом разработки, определять ресурсные потребности и отслеживать прогресс.

Планирование процесса проектирования требует тщательного анализа задач и ресурсов, а также постоянного мониторинга и корректировки планов в процессе работы над проектом. Использование современного программного обеспечения для управления проектами, такого как Microsoft Project, Jira, Trello и других, позволяет эффективно создавать и поддерживать различные виды планов.

29. **Структурный подход к проектированию программного обеспечения: основные принципы, лежащие в основе структурного подхода, средства описания функциональной структуры, средства описания отношения между данными, применение средств на стадиях жизненного цикла программного обеспечения.**

Структурный подход к проектированию программного обеспечения: основные принципы, лежащие в основе структурного подхода, средства описания функциональной структуры, средства описания отношения между данными, применение средств на стадиях жизненного цикла программного обеспечения

Сущность структурного подхода к разработке ИС заключается в ее декомпозиции (разбиении) на автоматизируемые функции: система разбивается на функциональные подсистемы, которые в свою очередь делятся на подфункции, подразделяемые на задачи и так далее. Процесс разбиения продолжается вплоть до конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны.

Все наиболее распространенные методологии структурного подхода базируются на ряде общих принципов. В качестве двух базовых принципов используются следующие:

- принцип "разделяй и властвуй" - принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;
- принцип иерархического упорядочивания - принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.
- принцип абстрагирования - заключается в выделении существенных аспектов системы и отвлечения от несущественных;
- принцип формализации - заключается в необходимости строгого методического подхода к решению проблемы;
- принцип непротиворечивости - заключается в обоснованности и согласованности элементов;
- принцип структурирования данных - заключается в том, что данные должны быть структурированы и иерархически организованы.

В структурном анализе используются в основном две группы средств, иллюстрирующих функции, выполняемые системой и отношения между данными. Каждой группе средств соответствуют определенные виды моделей (диаграмм), наиболее распространенными среди которых являются следующие:

- SADT модели и соответствующие функциональные диаграммы;
- DFD диаграммы потоков данных;
- ERD диаграммы "сущность-связь".

30. Спецификации ПО при структурном подходе: формальные модели, зависящие от подхода к разработке и не зависящие от подхода – диаграммы переходов состояний, математические модели предметной области.

Формальные модели, не зависящие от подхода к разработке:

- Диаграммы переходов состояний: Представляют собой графические модели поведения системы, где состояния отображаются в виде узлов, а переходы — в виде направленных дуг между ними.
- Математические модели предметной области: Включают в себя различные математические конструкции, такие как уравнения, неравенства и логические выражения, которые описывают аспекты предметной области и логику системы

Структурный подход к анализу и проектированию систем акцентирует внимание на декомпозиции системы на подсистемы и описании взаимодействия между этими подсистемами**:**

1. Функциональные диаграммы:
    - Представляют функциональную структуру системы, выделяют основные функции и подфункции.
    - Позволяют увидеть иерархию функций и то, как они взаимосвязаны.
    - В структурном анализе часто используются для первоначальной декомпозиции системы на более мелкие и управляемые части.
2. Диаграммы потоков данных (DFD):
    - Иллюстрируют, как данные движутся внутри системы, как они преобразуются функциями и какие данные хранятся.
    - Состоят из следующих элементов:
      - Процессы (обозначаются кругами или прямоугольниками с закругленными углами), которые показывают, как входные данные преобразуются в выходные.
      - Данные (обозначаются стрелками), которые представляют потоки данных между элементами.
      - Хранилища данных (обозначаются двумя параллельными линиями), где данные могут накапливаться и храниться для последующего использования.
      - Внешние сущности (обозначаются прямоугольниками), которые представляют источники и приемники данных за пределами системы.
    - DFD могут быть разработаны на различных уровнях абстракции, от общего обзора (уровень контекста) до детализированных диаграмм для каждого процесса.
3. Диаграммы отношений компонентов данных:
    - Описывают структуру баз данных или другие структуры для хранения данных, используемые в системе.
    - Сюда могут входить схемы баз данных, ER-диаграммы (Entity-Relationship), которые показывают сущности, их атрибуты и отношения между ними.
    - Эти диаграммы помогают определить, как данные организованы, какие связи существуют между различными частями данных и как эти данные могут быть использованы или изменены в процессе работы системы.

Объектно-ориентированный подход к разработке программного обеспечения фокусируется на моделировании программных систем с использованием объектов – сущностей, объединяющих данные и поведение

4. Диаграммы вариантов использования (Use Case Diagrams):
    - Описывают функциональность системы с точки зрения внешних акторов (пользователей или других систем), которые взаимодействуют с системой.
    - Варианты использования представляют цели или задачи, которые акторы стремятся выполнить с помощью системы.
    - Диаграммы вариантов использования помогают определить требования к системе и обеспечивают основу для планирования тестирования и написания пользовательских руководств.
5. Контекстные диаграммы классов (Class Diagrams):
    - Показывают статическую структуру системы в терминах классов и отношений между ними.

- Классы описывают объекты системы, их атрибуты (данные) и методы (поведение).
- Отношения могут включать ассоциации, наследование, реализацию интерфейсов и зависимости.
- Контекстные диаграммы классов помогают разработчикам понять структуру системы и способы взаимодействия её частей.

6. Диаграммы последовательностей (Sequence Diagrams):
   - Иллюстрируют взаимодействие объектов в рамках определенного сценария на основе временной последовательности.
   - Показывают, как объекты обмениваются сообщениями в процессе выполнения варианта использования или сценария.
   - Диаграммы последовательностей полезны для понимания динамического поведения системы и для определения требований к интерфейсам объектов.
7. Диаграммы деятельности (Activity Diagrams):
   - Отображают рабочий процесс или операции системы, показывая последовательность действий и контрольные структуры, такие как ветвления и параллельное выполнение.
   - Могут использоваться для моделирования бизнес-процессов или для детализации вариантов использования.
   - Диаграммы деятельности помогают в определении и оптимизации рабочих процессов и в улучшении понимания логики работы системы.

Поскольку разные модели описывают проектируемое программное обеспечение с разных сторон, рекомендуется использовать сразу несколько моделей и сопровождать их текстами: словарями, описаниями и т. п., которые поясняют соответствующие диаграммы.

31. Метод функционального моделирования SADT: функциональная модель SADT, стандарт IDEFO; синтаксис и семантика моделей IDEFO: действия-функции; стрелки входа, управления, выхода, механизма исполнения; комбинированные стрелки, разбиение и соединение стрелок; туннели.

Метод функционального моделирования **SADT** (Structured Analysis and Design Technique), также известный как **IDEF0** (Integration Definition for Function Modeling), является стандартным подходом к моделированию функций системы и их взаимосвязей. Модель SADT/IDEF0 представляет собой графическое описание процессов и систем, позволяя анализировать функции и их взаимодействия на разных уровнях детализации.
Методология функционального моделирования IDEFO — это технология описания системы в целом как множества взаимозависимых действий, или функций.
Важно отметить функциональную направленность IDEFO — функции системы исследуются независимо от объектов, которые обеспечивают их выполнение.
Чаще всего она применяется на логическом уровне и используется на ранних стадиях разработки.
IDEFO сочетает в себе небольшую по объему *графическую нотацию* (она содержит только два обозначения: блоки и стрелки) со строгими и четко определенными *рекомендациями*.
Важно определить:

1. *Назначение модели*
2. *Границы моделирования*
3. *Целевую аудиторию*
4. *точку зрения* (перспектива, с которой наблюдалась система при построение модели)

Поскольку модели IDEFO представляют систему как множество иерархических (вложенных) функций, в первую очередь должна быть определена функция, описывающая систему в целом — *контекстная функция*.
Любой блок может быть декомпозирован на составляющие его блоки.
В IDEFO также моделируются управление и механизмы исполнения. Под управлением понимаются объекты, воздействующие на способ, которым блок преобразует вход в выход.
Механизм исполнения — объекты, которые непосредственно выполняют преобразование входа в выход, но не потребляются при этом сами по себе.
Для отображения категорий информации, присутствующих на диаграммах IDEFO, существует аббревиатура ICOM, отображающая четыре возможных типа стрелок:
I (Input) — вход — нечто, что потребляется в ходе выполнения
процесса;
C (Control) — управление — ограничения и инструкции, влияющие на ход выполнения процесса;
O (Output) — выход — нечто, являющееся результатом выполнения процесса;
M (Mechanism) — исполняющий механизм — нечто, что используется для выполнения процесса, но не потребляется само по себе.

Комбинированные стрелки:

- выход-вход
  один из блоков должен полностью завершить работу перед началом работы другого блока.
- выход-управление
  один блок управляет работой другого
- выход-механизм исполнения
  выход одного функционального блока применяется в качестве оборудования для работы другого блокавыход-обратная связь на управление
  зависимые блоки формируют обратные связи для управляющих ими блоков.
- выход-обратная связь на выход
  применяется для описания циклов повторной обработки чего-либо
  Разбиение и соединение стрелок. Выход функционального блока может использоваться в *нескольких* других блоках.
  Туннели

Понятие *связанные стрелки* используется для управления уровнем детализации диаграмм. Если одна из стрелок диаграммы отсутствует на родительской диаграмме (например, ввиду своей несущественности для родительского уровня) и не связана с другими стрелками той же диаграммы, точка входа этой стрелки на диаграмму или выхода с нее обозначается ***туннелем***

32. **Построение моделей IDEFO: диаграммы, нумерация блоков и диаграмм, границы моделирования, наименование контекстного блока; типы связей между функциями: случайная, логическая, временная, процедурная, коммуникационная, последовательная, функциональная; дерево модели, презентационные диаграммы (FEO-диаграммы).**

#СтруктурныйПодход #ДиаграммыДанных

Префикс повторяется для каждого блока модели. Номера используются для отражения уровня декомпозиции, на котором находится блок. Блок А0 декомпозируется в блоки А1, А2, А3 и т.д. А1 декомпозируется в А11, А12, А13 и т.д. А11 декомпозируется в А111, А112, А113 и т.д.

Функциональный блок *декомпозируется*, если необходимо *детально* описать его работу.

На концах **граничных стрелок** (начинающихся или заканчивающихся за пределами диаграммы) детских диаграмм помещаются коды ICOM, чтобы показать, где находится соответствующая стрелка на родительской диаграмме

![[Untitled 97.png|Untitled 97.png]]

Есть 2 подхода к началу моделирования:

- в ширину
  каждая диаграмма максимально детализируется перед декомпозицией
- в глубину
  сначала определяется иерархия блоков, а затем создаются стрелки

Перед декомпозированием блока нужно удостовериться, не приведет ли это к превышению установленной ранее глубины детализации для данной модели. _Еще одно правило состоит в том, что моделирование IDEFO должно продолжаться до тех пор, пока стрелки предшествования (вход и выход) преобладают на диаграммах.

_

Типы связей между функциями:

- Случайная: Связь без четкой последовательности.
- Логическая: Связь, основанная на логических условиях.
- Временная: Связь, зависящая от времени.
- Процедурная: Строго определенная последовательность действий.
- Коммуникационная: Обмен информацией между функциями.
- Последовательная: Одна функция следует за другой.
- Функциональная: Связь, основанная на функциональной зависимости.

Дерево модели:

- Иерархическое представление всех диаграмм и блоков в модели IDEF0.
- Показывает структуру модели и взаимосвязи между ее частями.

![[Untitled 1 48.png|Untitled 1 48.png]]

Презентационные диаграммы

(For Exposition Only diagrams — FEO diagrams) часто включают в модели, чтобы проиллюстрировать другие точки зрения или детали, выходящие за рамки традиционного синтаксиса IDEFO.

- Используются для наглядного представления информации о функциях, их взаимосвязях и потоках данных.
- Могут включать различные визуальные элементы, такие как изображения, цвета и символы, для улучшения понимания и коммуникации.

При построении моделей IDEF0 важно следовать стандартам и соглашениям, чтобы обеспечить четкость и возможность интерпретации модели другими участниками проекта или заинтересованными сторонами.

33. **Метод описания процессов IDEF3: синтаксис и семантика моделей IDEF3: единица работы – действие (процесс); связи: временное предшествование, объектный поток, нечеткое отношение; соединения: «и», «или», «исключающее или»; синхронные и асинхронные соединения, парность соединений, комбинации соединений; указатели; декомпозиция действий.**

#СтруктурныйПодход #ДиаграммыДанных

IDEF3 — способ описания процессов, основной целью которого является обеспечение структурированного метода, используя который эксперт в предметной области может описать положение вещей как упорядоченную последовательность событий с одновременным описанием объектов, имеющих непосредственное отношение к процессу.

IDEF3 *не имеет* жестких синтаксических или семантических ограничений,
делающих неудобным описание неполных или нецелостных систем.

Основой модели IDEF3 служит так называемый сценарий бизнес-процесса, который выделяет последовательность действий или подпроцессов анализируемой системы.

Все связи являются **однонаправленными** обычно организовываются **слева на права**.

Наиболее часто нечеткие отношения используются для описания специальных случаев связей предшествования, например для описания *альтернативных вариантов временного предшествования*.
Завершение одного действия может инициировать начало выполнения сразу нескольких других действий, или, наоборот, определенное действие может требовать завершения нескольких других действий для начала своего выполнения.

Соединения разбивают или соединяют внутренние потоки и используются для описания ветвления процесса.

-

Разворачивающие соединения используются для разбиения потока. Завершение одного действия вызывает начало выполнения нескольких других.

-

Сворачивающие соединения объединяют потоки. Завершение одного или нескольких действий вызывает начало выполнения только одного другого действия

Синхронное выполнение - случаи, когда время начала или окончания параллельного выполняемых действий должно быть одинаковым.

Заметим, что синхронное разворачивающее соединение не обязательно должно иметь парное себе сворачивающее соединение. Действительно, начинающиеся одновременно действия вовсе не обязаны оканчиваться одновременно, как это видно из примера с состязаниями. Также возможны ситуации синхронного окончания асинхронно начавшихся действий.

Все соединения на диаграммах должны быть **парными**, любое разворачивающее соединение имеет парное себе сворачивающее. Однако типы соединений вовсе не обязательно должны совпадать.

Указатели — это специальные символы, которые ссылаются на
другие разделы описания процесса.

Действия в IDEF3 могут быть декомпозированы, или разложены
на составляющие, для более детального анализа. Декомпозировать действие можно несколько раз. Это позволяет документировать альтернативные потоки процесса в одной модели.

34. Построение моделей IDEF3: диаграммы, нумерация блоков и диаграмм, сценарий, границы моделирования, определение действий и объектов.

#СтруктурныйПодход #ДиаграммыДанных

Поскольку модели IDEF3 могут одновременно разрабатываться несколькими командами, каждому аналитику выделяется уникальный диапазон номеров действий, что обеспечивает их независимость друг от друга.

Перед тем как попросить экспертов предметной области подготовить описание моделируемого процесса, должны быть документированы границы моделирования, чтобы экспертам была понятна необходимая глубина и полнота требуемого от них описания.

Кроме того, если точка зрения аналитика на процесс отличается от обычной точки зрения для эксперта, это должно быть ясно и аккуратно описано.

Вполне возможно, что эксперты не смогут сделать приемлемое описание без применения формального опроса автором модели. В таком случае автор должен заранее приготовить набор вопросов таким же образом, как журналист заранее подготавливает вопросы для интервью.

Итак, **IDEF3** — это способ описания бизнес-процессов, который нужен для описания положения вещей как упорядоченной последовательности событий с одновременным описанием объектов, имеющих непосредственное отношение к процессу. IDEF3 хорошо приспособлен для сбора данных, требующихся для проведения структурного анализа системы. Кроме того, IDEF3 применяется при проведении стоимостного анализа поведения моделируемой системы.

https://studfile.net/preview/4129228/page:10/

35. Метод структурного анализа потоков данных: назначение диаграмм потоков данных (DFD); синтаксис и семантика DFD: функциональные блоки (системы и подсистемы, процессы), внешние сущности, потоки данных, хранилища данных, ветвление и объединение потоков данных.

#СтруктурныйПодход #ДиаграммыДанных

Диаграммы потоков данных (Data Flow Diagrams — DFD) представляют собой иерархию функциональных процессов, связанных потоками данных. Цель такого представления — продемонстрировать, как каждый процесс преобразует свои входные данные в выходные, а также выявить отношения между этими процессами.

Так же, как и диаграммы IDEFO, диаграммы потоков данных моделируют систему как набор действий, соединенных друг с другом стрелками. Диаграммы потоков данных также могут содержать два новых типа объектов: объекты, собирающие и хранящие инфор-мацию — хранилища данных и внешние сущности — объекты, которые моделируют взаимодействие с теми частями системы (или другими системами), которые выходят за границы моделирования. В отличие от стрелок в IDEFO, которые иллюстрируют отношения, стрелки в DFD показывают, как объекты (включая и данные) реально перемещаются от одного действия к другому. Это представление потока вкупе с хранилищами данных и внешними сущностями обеспечивает отражение в DFD-моделях таких физических характеристик системы, как двиэюение объектов (потоки дт^ых), хранение объектов (хранилища данных), источники и потребители объектов (внешние сущности). Построение DFD-диаграмм в основном ассоциируется с разработ-кой программного обеспечения, поскольку нотация DFD изначально была разработана для этих целей. В частности, графическое изображение объектов на DFD-диаграммах этой главы соответствует принятому Крисом Гейном (Chris Gane) и Тришем Сарсоном (Trish Sarson), авторами DFD-метода, известного как метод Гейна — Сарсона. Другой распространенной нотацией DFD является так называемый метод Йордана — Де Марко (Yourdon — DeMarco).

3.2 Синтаксис и семантика диаграмм потоков данных

В отличие от IDEFO, рассматривающего систему как множество взаимопересекающихся действий, в названиях объектов DFD-диарамм преобладают имена существительные. Контекстная DFD-диаграмма часто состоит из одного функционального блока и нескольких внешних сущностей. Функциональный блок на этой диаграмме обычно имеет имя, совпадающее с именем всей системы (рис. 3.2). Добавление на диаграмму внешних ссылок не изменяет фундаментального требования, что модель должна строиться с единственной точки зрения и должна иметь четко определенные цель и границы,что уже обсуждалось ранее.

3.2.1 Функциональные блоки

Функциональный блок DFD моделирует некоторую функцию, которая преобразует какое-либо сырье в какую-либо продукцию (или, в терминах IDEF, вход в выход). Хотя функциональные блоки DFD и изображаются в виде прямоугольников с закругленными углами, они почти идентичны функциональным блокам IDEFO и действиям IDEF3. Как и действия IDEF3, функциональные блоки DFD имеют входы и выходы, но не имеют управления и механизма исполнения как IDEFO. В некоторых интерпретациях нотации DFD Гейна — Сарсона механизмы исполнения IDEFO моделируются как ресурсы и изображаются в нижней части прямоугольника (рис. 3.3).

3.2.2 Внешние сущности

Внешние сущности обеспечивают необходимые входы для системы и/или являются приемниками для ее выходов. Одна внешняя сущность может одновременно предоставлять входы (функционируя как поставщик) и принимать выходы (функционируя как получатель). Внешние сущности изображаются как прямоугольники (рис. 3.4) и обычно размещаются у краев диаграммы. Одна внешняя сущность может быть размещена на одной и той же диаграмме в нескольких экземплярах. Этот прием полезно применять для сокращения количества линий, соединяющих объекты на диаграмме.

3.2.3 Стрелки (потоки данных) Стрелки описывают передвижение (поток) объектов от одной час-ти системы к другой. Поскольку все стороны обозначающего функциональный блок DFD прямоугольника равнозначны (в отличие от IDEFO), стрелки могут начинаться и заканчиваться в любой части бло-ка. В DFD также используются двунаправленные стрелки, которые нужны для отображения взаимодействия между блоками (например, диалога типа приказ — результат выполнения). На рис. 3.5 двунаправ-ленная стрелка обозначает взаимный обмен информацией между департаментами маркетинга и рекламы и пластиковых карт.

3.2.4 Хранилища данных

В то время как потоки данных представляют объекты в процессе их передвижения, хранилища данных моделируют их во всех остальных состояниях. При моделировании производственных систем хранилищами данных служат места временного складирования, где хранится продукция на промежуточных стадиях обработки. В информационных системах хранилища данных представляют любой механизм, который поддерживает хранение данных для их промежуточной обработки.

3.2.5 Ветвление и объединение

Стрелки на DFD-диаграммах могут быть разбиты (разветвлены) на части, и при этом каждый получившийся сегмент может быть переименован таким образом, чтобы показать декомпозицию данных, переносимых данным потоком (рис. 3.7).

Стрелки могут и соединяться между собой (объединяться) для формирования так называемых комплексных объектов.

См далее [[36. DFD Построение диаграмм потоков данных]]

36. **Построение диаграмм потоков данных: нумерация объектов, построение контекстных диаграмм, правила детализации – балансировка, нумерация; спецификация процесса, требования, предъявляемые к спецификации, структурированный естественный язык описания спецификации процессов, верификация модели DFD – проверка на полноту и согласованность.**

#СтруктурныйПодход #ДиаграммыДанных

> Диаграммы потоков данных (DFD) — методология графического структурного анализа, описывающая внешние по отношению к системе источники и адресаты данных, логические функции, потоки данных и хранилища данных, к которым осуществляется доступ.

Добавление на диаграмму внешних ссылок не изменяет фундаментального требования, что модель должна строиться с единственной точки зрения и должна иметь четко определенные цель и границы.

В DFD нумерация объектов осуществляется следующим образом:

1. Каждый номер функционального блока может включать в себя префикс, номер родительской диаграммы и собственно номер объекта.
2. Номер объекта уникальным образом идентифицирует функциональный блок на диаграмме.
3. Номер родительской диаграммы и номер объекта в совокупности обеспечивают уникальную идентификацию каждого блока модели.

Уникальные номера присваиваются также каждому хранилищу данных и каждой внешней сущности вне зависимости от расположения объекта на диаграмме. Каждый номер хранилища данных содержит префикс D (от английского Data Store) и уникальный номер хранилища в модели (например, D3). Аналогично каждый номер каждой внешней сущности содержит префикс E (от английского External entity) и уникальный номер сущ-
ности в модели (например, E5).

![[Untitled 100.png|Untitled 100.png]]

Построение контекстной диаграммы (1 этап)- включает выполнение следующих действий:

• классификацию множества требований и организацию их в основные функциональные группы - процессы;

• идентификацию внешних объектов - внешних сущностей, с которыми система должна быть связана;

• идентификацию основных видов информации - потоков данных, циркулирующей между системой и внешними объектами;

• предварительную разработку контекстной диаграммы;

• изучение предварительной контекстной диаграммы и внесение в нее изменений по результатам ответов на возникающие при изучении вопросы по всем ее частям;

• построение контекстной диаграммы путем объединения всех процессов предварительной диаграммы в один процесс, а также группирования потоков.

2 этап - **формирование иерархии диаграмм потоков данных** - включает для каждого уровня:

• проверку и изучение основных требований по диаграмме соответствующего уровня (для первого уровня - по контекстной диаграмме);

• декомпозицию каждого процесса текущей диаграммы потоков данных с помощью детализирующей диаграммы или -- если некоторую функцию сложно или невозможно выразить комбинацией процессов, построение спецификации процесса;

• добавление определений новых потоков в словарь данных при каждом появлении их на диаграмме;

• проведение ревизии с целью проверки корректности и улучшения наглядности модели после построения двух-трех уровней.

При разработке контекстных диаграмм происходит **детализация** функциональной структуры будущей системы, что особенно важно, если разработка ведется несколькими коллективами разработчиков.

Полученную таким образом модель системы

**проверяют на полноту исходных данных об объектах системы и изолированность объектов

**(отсутствие информационных связей с другими объектами).

На следующем этапе каждую подсистему контекстной диаграммы детализируют при помощи диаграмм потоков данных. В процессе детализации соблюдают

правило б а л а н с и р о в к и - при детализации подсистемы можно использовать компоненты только тех подсистем, с которыми у разрабатываемой подсистемы существует информационная связь (т. е. с которыми она связана потоками данных).

Для облегчения восприятия процессы детализируемой подсистемы нумеруют, соблюдая иерархию номеров: так процессы, полученные при детализации процесса или подсистемы «1», должны нумероваться «1.1», «1.2» и т. д. Кроме этого желательно размещать на каждой диаграмме от 3-х до 6-7-ми процессов и не загромождать диаграммы деталями, не существенными на данном уровне.

Декомпозицию потоков данных необходимо осуществлять параллельно с декомпозицией процессов.

Решение о завершении детализации процесса принимают в следующих случаях:

• процесс взаимодействует с 2-3-мя потоками данных;

• возможно описание процесса последовательным алгоритмом;

• процесс выполняет единственную логическую функцию преобразования входной информации в выходную.

На недетализируемые процессы составляют спецификации, которые должны содержать описание логики (функций) данного процесса. Такое описание может, выполняться:

на естественном языке, с применением структурированного естественного языка (псевдокодов), с применением таблиц и деревьев решений, в виде схем алгоритмов, в том числе flow-форм и диаграмм Насси-Шнейдермана (см. § 2.4 Ивановой).

Требования к спецификациям

Применительно к функциональным спецификациям подразумевается, что:

• требование полноты означает, что спецификации должны содержать всю существенную информацию, где ничего важного не было бы упущено, и отсутствует несущественная информация, например детали реализации, чтобы не препятствовать разработчику в выборе наиболее эффективных решений;

• требование точности означает, что спецификации должны однозначно восприниматься как заказчиком, так и разработчиком.

Структурированный естественный язык (инфа из инета) применяется для читабельного, строгого описания спецификаций процессов.

Он является разумной комбинацией строгости языка программирования и читабельности естественного языка и состоит из подмножества слов, организованных в определённые логические структуры, арифметических выражений и диаграмм.

В состав языка входят следующие основные символы:

- глаголы, ориентированные на действие и применяемые к объектам;
- термины, определённые на любой стадии проекта ПО (например, задачи, процедуры, символы данных и т.п.);
- предлоги и союзы, используемые в логических отношениях;
- общеупотребительные математические, физические и технические термины;
- арифметические уравнения;
- таблицы, диаграммы, графы и т.п.;
- комментарии.

Полная спецификация процессов включает также описание структур данных, используемых как при передаче информации & потоке, так и при хранении в накопителе. Описываемые структуры данных могут содержать альтернативы, условные вхождения и итерации. Условное вхождение означает, что соответствующие элементы данных в структуре могут отсутствовать. Альтернатива означает, что в структуру может входить один из перечисленных элементов. Итерация означает, что элемент может повторяться некоторое количество раз (см. § 4.5 Ивановой).

Кроме того, для данных должен быть указан тип: непрерывное или дискретное значение. Для непрерывных данных могут определяться единицы измерений, диапазон значений, точность представления и форма физического кодирования. Для дискретных - может указываться таблица допустимых значений.
Полученную законченную модель необходимо проверить на
полноту и согласованность. Под согласованностью модели в данном случае понимают выполнение для всех потоков данных правила сохранения информации: все поступающие куда-либо данные должны быть считаны и записаны.

37. Структуры данных: несвязанные, с неявными связями, с явными связями; иерархические модели Джексона-Орра.

Структурой данных называют совокупность правил и ограничений, которые отражают связи, существующие между отдельными частями (элементами) данных.

Различают абстрактные структуры данных, используемые для уточнения связей между элементами, и конкретные структуры, используемые для представления данных в программах.

Все абстрактные структуры данных можно разделить на три группы: структуры, элементы которых не связаны между собой, структуры с неявными связями элементов — таблицы и структуры, связь элементов которых указывается явно-графы (рис. 4.18).

![[Untitled 101.png|Untitled 101.png]]

В **первую группу** входят **множества** (рис. 4.19, а) и **кортежи** (рис. 4.19, б). Наиболее существенная характеристика элемента данных в этих структурах - его принадлежность некоторому набору, т. е. отношение вхождения. Данные абстрактные структуры используют, если никакие другие отношения элементов не являются существенными для описываемых объектов.

![[Untitled 1 51.png|Untitled 1 51.png]]

Ко **второй группе** относят **векторы, матрицы, массивы (многомерные), записи, строки, а также таблицы

**, включающие перечисленные структуры в качестве частей. Использование этих абстрактных типов может означать, что существенным является не только вхождение элемента данных в некоторую структуру, но и их порядок, а также отношения иерархии структур, т. е. вхождение структуры в структуру более высокой степени общности (рис. 4.20).

![[Untitled 2 33.png|Untitled 2 33.png]]

В тех случаях, когда существенны связи элементов данных между собой, в качестве **модели структур данных используют графы

**[55]. На рис. 4.21 показаны различные варианты графовых моделей.

![[Untitled 3 23.png|Untitled 3 23.png]]

Иерархические модели позволяют описывать упорядоченные или неупорядоченные отношения вхождения элементов данных в компонент более высокого уровня, т. е. множества, таблицы и их комбинации. К иерархическим моделям относят модель Джексона Орра, для графического представления которой можно использовать:

- **диаграммы Джексона,** предложенные в составе методики проектирования программного обеспечения того же автора в 1975 г.;
- **скобочные диаграммы Орра**, предложенные в составе методики проектирования программного обеспечения Варнье-Орра (1974).

Диаграммы Джексона.

В основе диаграмм Джексона лежит предположение о том, что структуры данных, так же, как и программ, можно строить из элементов с использованием всего **трех основных конструкций: последовательности, выбора и повторения**. Каждая конструкция представляется в виде двухуровневой иерархии, на верхнем уровне которой расположен блок конструкции, а на нижнем - блоки элементов. Нотации конструкций различаются специальными символами в правом верхнем углу блоков элементов. В изображении последовательности дополнительный символ отсутствует. В изображении выбора ставится символ «о» (латинское) - сокращение английского «или» (or). Конструкции последовательности и выбора должны содержать по два или более элементов второго уровня. В изображении повторения в блоке единственного (повторяющегося) элемента ставится символ «*».

Так схема, показанная на рис. 4.22, а, означает, что конструкция A состоит из элементов B, C и D, следующих в указанном порядке. Схема на рис. 4.22, б означает, что конструкция S состоит либо из элемента P, либо из элемента Q, либо из элемента R. Схема, изображенная на рис. 4.22, в, показывает, что конструкция I может не содержать элементов или содержать один или более элементов X.

![[Untitled 4 15.png|Untitled 4 15.png]]

Скобочные диаграммы Орра.

Диаграмма Орра базируется на том же предположении о сходстве структур программ и данных, что и диаграмма Джексона. **Отличие состоит лишь в нотации.** Автор предлагает для представления конструкций данных использовать фигурные скобки (рис. 4.24).


38. Моделирование данных – диаграммы «сущность-связь» (ERD): сущность, связь, атрибут.

#СтруктурныйПодход #ДиаграммыДанных

Сетевые модели данных используют в тех случаях, если отношение между компонентами данных не исчерпываются включением. Для графического представления разновидностей этой модели используют несколько нотаций. Наиболее известны из них следующие:

• нотация П. Чена;

• нотация Р. Баркера;

• нотация IDEF1 (более современный вариант этой нотации - IDEF1X используется в CASEсистемах, например в системе ERWin).

Нотация Баркера является наиболее распространенной. Далее в настоящем разделе будем придерживаться именно этой нотации.

Базовыми понятиями сетевой модели данных являются: сущность, атрибут и связь.

Сущность — реальный или воображаемый объект, имеющий существенное значение для

рассматриваемой предметной области.

Каждая сущность должна:

• иметь уникальное имя;

• обладать одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через связь;

• обладать одним или несколькими атрибутами, которые однозначно идентифицируют каждый экземпляр сущности.

Сущность представляет собой множество экземпляров реальных или абстрактных объектов (людей, событий, состояний, предметов и т. п.). Имя сущности должно отражать тип или класс объекта, а не его конкретный экземпляр (Аэропорт, а не Внуково).

На диаграмме в нотации Баркера сущность изображается прямоугольником, иногда с закругленными углами (рис. 4.27, а).

Каждая сущность обладает одним или несколькими атрибутами.

Атрибут - любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности (рис. 4.27, б).

В сетевой модели атрибуты ассоциируются с конкретными сущностями, и, соответственно, экземпляр сущности должен обладать единственным определенным значением для ассоциированного атрибута. Атрибут, таким образом, представляет собой некоторый тип характеристик или свойств, ассоциированных с множеством реальных или абстрактных объектов.

Экземпляр атрибута - определенная характеристика конкретного экземпляра сущности. Он определяется типом характеристики и ее значением, называемым значением атрибута.

Атрибуты делятся на ключевые, т. е. входящие в состав уникального идентификатора, который называют первичным ключом, и описательные - прочие.

Первичный ключ - это атрибут или совокупность атрибутов и/или связей, предназначенная для уникальной идентификации каждого экземпляра сущности (совокупность признаков, позволяющих идентифицировать объект). Ключевые атрибуты помещают в начало списка и помечают символом «#» (рис. 4.27, в).

Описательные атрибуты бывают обязательными или необязательными. Обязательные атрибуты для каждой сущности всегда имеют конкретное значение, необязательные - могут быть не определены. Обязательные и необязательные описательные атрибуты помечают символами «*» и «о» соответственно.

Для сущностей определено понятие супертип и подтип.

Супертип - сущность обобщающая некую группу сущностей (подтипов).

Супертип характеризуется общими для подтипов атрибутами и отношениями. Например, для некоторых задач супертип «учащийся» обобщает подтипы «школьник» и «студент» (рис. 4.28).

Связь - поименованная ассоциация между двумя или более сущностями, значимая для

рассматриваемой предметной области.
Связь, таким образом, означает, что каждый экземпляр одной сущности ассоциирован с произвольным (в том числе и нулевым) количеством экземпляров второй сущности и наоборот. Если любой экземпляр одной сущности связан хотя бы с одним экземпляром другой сущности, то **связь является обязательной** (рис. 4.29, а).
Необязательная связь представляет собой условное отношение между сущностями (рис. 4.29, б).
![[Untitled 2 34.png|Untitled 2 34.png]]
Каждая сущность может быть связана любым количеством связей с другими сущностями модели. Связь предполагает некоторое отношение сущностей, которое характеризуется количеством экземпляров сущности, участвующих в связи с каждой стороны.
Различают три типа отношений (рис. 4.30):
1
_1 - «один-к-одному» - одному экземпляру первой сущности соответствует один экземпляр второй;
1
_n - «один-ко-многим» - одному экземпляру первой сущности соответствуют несколько экземпляров второй;
n*m - «многие-ко-многим» -> каждому экземпляру первой сущности может соответствовать несколько экземпляров второй и, наоборот, каждому экземпляру второй сущности может соответствовать несколько экземпляров первой.
![[Untitled 3 24.png|Untitled 3 24.png]]
Кроме того, **сущности бывают независимыми, зависимыми и ассоциированными.** Независимая сущность представляет независимые данные, которые всегда присутствуют в системе. Они могут быть связаны или не связаны с другими сущностями той же системы.
Зависимая сущность представляет данные, зависящие от других сущностей системы, поэтому она всегда должна быть связана с другими сущностями.
Ассоциированная сущность представляет данные, которые ассоциируются с отношениями между двумя и более сущностями. Обычно данный вид сущностей используется в модели для разрешения отношения «многие-ко-многим» (рис. 4.31).
![[Untitled 4 16.png|Untitled 4 16.png]]
Если экземпляр сущности полностью идентифицируется своими ключевыми атрибутами, то говорят о
полной идентификации сущности. В противном случае идентификация сущности осуществляется с использованием атрибутов связанной сущности, что указывается черточкой на линии связи (рис. 4.32).
![[Untitled 5 13.png|Untitled 5 13.png]]
Кроме этого, модель включает понятия взаимно **исключающих, рекурсивных и неперемещаемых связей.
**При наличии взаимно исключающей связи экземпляр сущности участвует только в одной связи из некоторой группы связей (рис. 4.33, а). Рекурсивная связь предполагает, что сущность может быть связана сама с собой (рис. 4.33, б). Неперемещаемая связь означает, что экземпляр сущности не может быть перенесен из одного экземпляра связи в другой (рис. 4.33, в).

39. Метод Баркера.

Метод Баркера — это метод моделирования данных, который обеспечивает разработчика ИС концептуальной схемой базы данных в форме одной модели или нескольких локальных моделей, которые относительно легко могут быть отображены в любую систему баз данных.

Наиболее распространенным средством моделирования данных являются диаграммы «сущность-связь» (ERD). С их помощью определяются важные для предметной области объекты (сущности), их свойства (атрибуты) и отношения друг с другом (связи). ERD непосредственно используются для проектирования реляционных баз данных.

40. Метод IDEF1.

#СтруктурныйПодход #ДиаграммыДанных

IDEF1 (integration definition for information modeling) — **одна из методологий семейства IDEF, которая применяется для построения информационной модели, представляющей структуру информации, необходимой для поддержки функций производственной системы или среды**.

Основные характеристики:

- Помогает выяснить структуру и содержание существующих потоков информации на любом предприятии.
- Помогает определить, какие проблемы, выявленные в результате функционального анализа и анализа потребностей, вызваны недостатком управления соответствующей информацией.
- Выявляет информационные потоки, требующие дополнительного управления для эффективной реализации модели.

Результаты анализа информационных потоков могут быть использованы для:

- стратегического и тактического планирования деятельности предприятия и её улучшения.

Методология IDEF1 позволяет на основе простых графических изображений моделировать информационные взаимосвязи и различия между реальными объектами, физическими и абстрактными зависимостями, информацией, относящейся к реальным объектам, и структурой данных, используемой для приобретения, накопления, применения и управления информацией.

![[Untitled 103.png|Untitled 103.png]]

Метод IDEF1, разработанный Т.Рэмей (T.Ramey), также основан на подходе П.Чена и позволяет построить модель данных, эквивалентную реляционной модели в третьей нормальной форме. В настоящее время на основе совершенствования методологии IDEF1 создана ее новая версия - методология IDEF1X. IDEF1X разработана с учетом таких требований, как простота изучения и возможность автоматизации. IDEF1X-диаграммы используются рядом распространенных CASE-средств (в частности, ERwin, Design/IDEF).

Сущность в методологии IDEF1X является независимой от идентификаторов или просто независимой, если каждый экземпляр сущности может быть однозначно идентифицирован без определения его отношений с другими сущностями. Сущность называется зависимой от идентификаторов или просто зависимой, если однозначная идентификация экземпляра сущности зависит от его отношения к другой сущности (рисунок 2.30).

![[Untitled 1 53.png|Untitled 1 53.png]]

Каждой сущности присваивается уникальное имя и номер, разделяемые косой чертой "/" и помещаемые над блоком.

Связь может дополнительно определяться с помощью указания степени или мощности (количества экземпляров сущности-потомка, которое может существовать для каждого экземпляра сущности-родителя). В IDEF1X могут быть выражены следующие мощности связей:

- каждый экземпляр сущности-родителя может иметь ноль, один или более связанных с ним экземпляров сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не менее одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя должен иметь не более одного связанного с ним экземпляра сущности-потомка;
- каждый экземпляр сущности-родителя связан с некоторым фиксированным числом экземпляров сущности-потомка.

Если экземпляр сущности-потомка однозначно определяется своей связью с сущностью-родителем, то связь называется **идентифицирующей**, в противном случае - **неидентифицирующей.**

Связь изображается линией, проводимой между сущностью-родителем и сущностью-потомком с точкой на конце линии у сущности-потомка. Мощность связи обозначается как показано на рис. 2.31 (мощность по умолчанию - N).

![[Untitled 2 35.png|Untitled 2 35.png]]

Идентифицирующая связь между сущностью-родителем и сущностью-потомком изображается сплошной линией. Сущность-потомок в идентифицирующей связи является зависимой от идентификатора сущностью. Сущность-родитель в идентифицирующей связи может быть как независимой, так и зависимой от

идентификатора сущностью (это определяется ее связями с другими сущностями). Пунктирная линия изображает неидентифицирующую связь. Сущность-потомок в неидентифицирующей связи будет независимой от идентификатора, если она не является также сущностью-потомком в какой-либо идентифицирующей связи.

Атрибуты изображаются в виде списка имен внутри блока сущности. Атрибуты, определяющие первичный ключ, размещаются наверху списка и отделяются от других атрибутов горизонтальной чертой.

Сущности могут иметь также внешние ключи (Foreign Key), которые могут использоваться в качестве части или целого первичного ключа или неключевого атрибута. Внешний ключ изображается с помощью помещения внутрь блока сущности имен атрибутов, после которых следуют буквы FK в скобках (рисунок 2.35).

41. **Структурная и функциональная схемы: структурные схемы пакетов программ, программного комплекса, программной системы; функциональная схема-схема данных, основные обозначения по ГОСТ 19.701-90.**

#СтруктурныйПодход #ДиаграммыДанных
программного комплекса, программной системы; функциональная схема-схема данных, основные обозначения по ГОСТ 19.701-90
Процесс проектирования сложного программного обеспечения начинают с уточнения его структуры, т. е. определения структурных компонентов и связей между ними.
Результат уточнения структуры может быть представлен в виде структурной и/или функциональной схем и описания (спецификаций) компонентов.
Структурная схема разрабатываемого программного обеспечения.

> Структурной называют схему, отражающую состав и взаимодействие по управлению частей разрабатываемого программного обеспечения.

Структурные схемы *пакетов программ* не информативны, поскольку организация программ в пакеты не предусматривает передачи управления между ними. Поэтому **структурные схемы разрабатывают для каждой программы пакета**, а список программ пакета определяют, анализируя функции, указанные в техническом задании.
Самый простой вид программного обеспечения - программа, которая в качестве структурных компонентов может включать только подпрограммы и библиотеки ресурсов.
**Разработку структурной схемы программы обычно выполняют методом пошаговой детализации **(см. § S.2 Ивановой).
**Структурными компонентами программной системы или программного комплекса могут служить программы, подсистемы, базы данных, библиотеки ресурсов и т. п.
**
Структурная схема *программного комплекса* демонстрирует передачу управления от программы-диспетчера соответствующей программе (рис. 5.1).
![[Untitled 104.png|Untitled 104.png]]
Структурная схема программной системы, как правило, показывает наличие подсистем или других структурных компонентов. В отличие от программного комплекса отдельные части (подсистемы) программной системы интенсивно обмениваются данными между собой и, возможно, с основной программой. Структурная же схема программной системы этого обычно не показывает (рис. 5.2).
![[Untitled 1 54.png|Untitled 1 54.png]]
Функциональная схема.

> Функциональная схема или схема данных (ГОСТ 19.701-90) - схема взаимодействия компонентов программного обеспечения с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств.

Основные обозначения схем данных по ГОСТ 19.701-90 приведены в табл. 5.1.
Функциональные схемы, более информативны, чем структурные. На рис. 5.3 для сравнения приведены функциональные схемы программных комплексов и систем. Все компоненты структурных и функциональных схем должны быть описаны. При структурном подходе особенно тщательно необходимо прорабатывать спецификации межпрограммных интерфейсов, так как от качества их описания зависит количество самых дорогостоящих ошибок. К самым дорогим относятся ошибки, обнаруживаемые при комплексном тестировании, так как для их устранения могут потребоваться серьезные изменения уже отлаженных текстов.
![[Untitled 2 36.png|Untitled 2 36.png]]
![[Untitled 3 25.png|Untitled 3 25.png]]
![[Untitled 4 17.png|Untitled 4 17.png]]

42. Проектирование структуры программного обеспечения с использованием метода пошаговой детализации: основное правило и рекомендации по применению.

43. Структурные карты Константайна: назначение, типы вызов модулей-последовательный, параллельный, вызов сопрограммы; особые условия вызова-циклический, условный, однократный; диаграммы реализации параллельного вызова и вызова сопрограммы; типы связи – по данным, по управлению.

#СтруктурныйПодход #ДиаграммыДанных

Структурные карты Константайна

Используются для анализа технологичности иерархии модулей

Структурные карты Константайна **позволяют наглядно представить результат декомпозиции программы на модули и оценить ее качество**, т. е. соответствие рекомендациям структурного программирования (сцепление и связность).

На структурной карте отношения между модулями представляют в виде графа, вершинам которого соответствуют модули и общие области данных, а дугам - межмодульные вызовы и обращения к общим областям данных. Различают четыре типа вершин (рис. 5.7):

- модуль - подпрограмма,
- подсистема - программа,
- библиотека - совокупность подпрограмм, размещенных в отдельном модуле,
- область данных - специальным образом оформленная совокупность данных,, к которой возможно обращение извне.

![[Untitled 105.png|Untitled 105.png]]

При этом отдельные части программной системы (программы, подпрограммы) могут вызываться **последовательно, параллельно или как сопрограммы** (рис. 5.8).

![[Untitled 1 55.png|Untitled 1 55.png]]

Чаще всего используют последовательный вызов, при котором модули, передав управление, ожидают завершения выполнения вызванной программы или подпрограммы, чтобы продолжить прерванную обработку.

Под параллельным вызовом понимают распараллеливание вычислений на нескольких вычислителях, когда при активизации другого процесса данный процесс продолжает работу (рис.5.9, а). На однопроцессорных компьютерах в мультипрограммных средах в этом случае начинается попеременное выполнение соответствующих программ.

**Параллельные процессы бывают синхронные и асинхронные.

**Для синхронных процессов определяют точки синхронизации - моменты времени, когда производится обмен информацией между процессами. Асинхронные процессы обмениваются информацией только в момент активизации параллельного процесса.

Под вызовом сопрограммы понимают возможность поочередного выполнения двух одновременно запущенных программ, например, если одна программа подготовила пакет данных для вывода, то вторая может ее вывести, а затем перейти в состояние ожидания следующего пакета. Причем в мультипрограммных системах основная программа, передав данные, продолжает работать, а не переходит в состояние ожидания, как изображено на рис. 5.9, б.

![[Untitled 2 37.png|Untitled 2 37.png]]

Если стрелка, изображающая вызов, касается блока, то обращение происходит к модулю целиком, а если входит в блок, то - к элементу внутри модуля.

При необходимости на структурной карте можно уточнить **особые условия вызова** (рис. 5.10):

циклический вызов, условный вызов и однократный вызов - при повторном вызове основного модуля однократно вызываемый модуль не активизируется!

![[Untitled 3 26.png|Untitled 3 26.png]]

Связи по данным и управлению обозначают стрелками, параллельными дуге вызова, направление стрелки указывает направление связи (рис. 5.11).

![[Untitled 4 18.png|Untitled 4 18.png]]

44. Проектирование структур данных: представление данных в оперативной памяти – векторная структура, списковые структуры; представление данных во внешней памяти – способы организации данных с последовательным и прямым доступом.

#СтруктурныйПодход #моделиДанных

Под проектированием структур данных понимают разработку их представлений в памяти.

Основными параметрами, которые необходимо учитывать при проектировании структур данных, являются:

• вид хранимой информации каждого элемента данных;

• связи элементов данных и вложенных структур;

• время хранения данных структуры («время жизни»);

• совокупность операции над элементами данных, вложенными структурами и структурами в целом.

Связи элементов и вложенных структур, а также их

**устойчивость и совокупность операций

над элементами и вложенными структурами определяют структуры памяти

**, используемые для представления данных. Время жизни учитывают при размещении данных в статической или динамической памяти, а также во внешней памяти.

Представление данных в оперативной памяти

Различают **две базовые структуры организации данных** в оперативной памяти: **векторную и списковую**.

Векторная структура представляет собой последовательность байт памяти, которые используются для размещения полей данных (рис. 5.14). Последовательное размещение организованных структур данных позволяет осуществлять прямой доступ к элементам: по индексу (совокупности индексов) - в массивах или строках или по имени поля - в записях или объектах.

![[Untitled 106.png|Untitled 106.png]]

Однако **выполнение операций добавления и удаления элементов при использовании

векторных структур

для размещения элементов массивов **может потребовать осуществления многократных сдвигов элементов.

**Структуры данных в векторном представлении можно размещать как в статической, так и в динамической памяти.

**Расположение векторных представлений в динамической памяти иногда позволяет существенно увеличить эффективность использования оперативной памяти.

Желательно размещать в динамической памяти временные структуры, хранящие промежуточные результаты, и структуры, размер которых сильно зависит от вводимых исходных данных.

Списковые структуры строят из специальных элементов, включающих помимо информационной части еще и один или несколько указателей-адресов элементов или вложенных структур, связанных с данным элементом. Размещая подобные элементы в динамической памяти можно организовывать различные внутренние структуры (рис. 5.15).

![[Untitled 1 56.png|Untitled 1 56.png]]

Однако **при использовании списковых структур следует помнить**, что:

• для хранения указателей необходима дополнительная память;

• поиск информации в линейных списках осуществляется последовательно, а потому требует больше времени;

• построение списков и выполнение операций над элементами данных, хранящимися в списках, требует более высокой квалификации программистов, более трудоемко, а соответствующие подпрограммы содержат больше ошибок и, следовательно, требуют более тщательного тестирования.

Обычно

векторное представление используют для хранения статических множеств, таблиц (одномерных и многомерных), например, матриц, строк, записей, а также графов, представленных матрицей смежности, матрицей инцидентности или аналитически [55].

Списковое представление удобно для хранения динамических (изменяемых) структур и структур со сложными связями.

Представление данных во внешней памяти. Современные операционные системы поддерживают два способа организации данных во внешней памяти:
**последовательный и с
прямым доступом.
**

При последовательном доступе к данным возможно выполнение только последовательного чтения элементов данных или последовательная их запись. Такой вариант предполагается при работе с логическими устройствами типа клавиатуры или дисплея, при обработке текстовых файлов или файлов, формат записей которых меняется в процессе работы.

Прямой доступ возможен только для дисковых файлов, обмен информацией с которыми осуществляется записями фиксированной длины (двоичные файлы С или типизированные файлы Pascal). Адрес записи такого файла можно определить по ее номеру, что и позволяет напрямую обращаться к нужной записи.

При выборе типа памяти для размещения структур данных **следует иметь в виду**, что:

- в оперативной памяти размещают данные, к которым необходим быстрый доступ как для чтения, так и для их изменения;
- во внешней - данные, которые должны сохраняться после завершения программы.

45. Проектирование программного обеспечения с использованием методов декомпозиции данных: метод Джексона, метод Варнье-Орра.

Обе методики предназначены для создания «простых» программ, работающих со сложными, но иерархически организованными структурами данных. При необходимости разработки программных систем в обоих случаях предлагается вначале разбить систему на отдельные программы, а затем использовать данные методики.

Методика Джексона.

При создании своей методики М. Джексон исходил из того, что структуры исходных данных и результатов определяют структуру программы. **Методика основана на** поиске соответствий структур исходных данных и результатов. Однако при ее применении возможны ситуации, когда на каких-то уровнях соответствия отсутствуют. Например, записи исходного файла сортированы не в том порядке, в котором соответствующие строки должны появляться в отчете. Такие ситуации были названы **«столкновениями»**. Выделяют **несколько типов столкновений**, которые разрешают по-разному. При различной последовательности записей их просто сортируют до обработки.

Разработка структуры программы в соответствии с методикой **выполняется следующим образом:

- строят изображение структур входных и выходных данных;
- выполняют идентификацию связей обработки (соответствия) между этими данными;
- формируют структуру программы на основании структур данных и обнаруженных соответствий;
- добавляют блоки обработки элементов, для которых не обнаружены соответствия;
- анализируют и обрабатывают несоответствия, т.е. разрешают «столкновения»;
- добавляют необходимые операции (ввод, вывод, открытие/закрытие файлов и т. п.);
- записывают программу в структурной нотации (псевдокоде).

Методика Варнье-Орра.

Методика Варнье-Орра **базируется на том же положении**, что и методика Джексона, **но основными** при построении программы **считаются структуры выходных данных** и, если структуры входных данных не соответствуют структурам выходных, то их допускается менять. Таким образом, **ликвидируется основная причина столкновений**.

Однако на практике не всегда существует возможность пересмотра структур входных данных: эти структуры уже могут быть строго заданы, например, если используются данные, полученные при выполнении других программ, поэтому **данную методику применяют реже**.

Как следует из вышеизложенного, методики Джексона и Варнье-Орра могут использоваться только в том случае, если **данные** разрабатываемых программ **могут быть представлены в виде иерархии или совокупности иерархий**.

46. Структурный подход к проектированию программного обеспечения: основные достоинства и недостатки, особенности применения, перспективы развития.

Иванова:

Структурный подход к программированию (60-70-е годы XX в.).

Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших (до 40 - 50 операторов) подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название процедурной декомпозиции.

Использование модульного программирования существенно упростило разработку программного обеспечения несколькими программистами. Теперь каждый из них мог разрабатывать свои модули независимо, обеспечивая взаимодействие модулей через специально оговоренные межмодульные интерфейсы. Кроме того, модули в дальнейшем без изменений можно было использовать в других разработках, что повысило производительность труда программистов.

Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы, размер которых не превышает 100 000 операторов [10]. Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки программного обеспечения большого объема было предложено использовать объектный подход.

Яндекс нейро:

Структурный подход к проектированию программного обеспечения заключается в декомпозиции (разбиении) сложных систем на части с целью последующей реализации в виде отдельных небольших подпрограмм. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны.

Достоинства структурного подхода:

- возможность проведения глубокого анализа бизнес-процессов, выявления узких мест;
- применение универсальных графических языков моделирования обеспечивает логическую целостность и полноту описания, необходимую для достижения точных и непротиворечивых результатов;
- проверенность временем и широкое распространение среди аналитиков и разработчиков.

Недостатки структурного подхода:

- низкая наглядность для неподготовленных пользователей модели;
- сложность восприятия иерархически упорядоченной информации;
- необходимость следования жёсткой (не всегда необходимой) структуре.

Перспективы развития структурного подхода:

- возможность быстрого прототипирования и применения алгоритмических языков четвёртого поколения, что позволит внедрить эволюционный подход.

Применение структурного подхода рекомендуется для правильного, точного и полного определения требований к проектируемой системе на начальных этапах.

Что я бы отвечал:

Рассказал бы определение из Ивановой;

Плюсы:

- Модульный подход позволил вести разработку нескольким программистам
- Иерархическая структура хорошо подходит для описания и анализа бизнес процессов
- Повышается производительность труда программистов засчёт повторного использования модулей

Минусы:

- Не подходит для проектов с количеством операторов больше 100 000

- У разработанного по такому методу проектирования ПО есть проблемы с выявлением ошибок (особенно на глубоких уровнях модели).
- Иерархия моделей обладает низкой наглядностью для людей без подготовки.

Особенности применения исходят из плюсов и минусов

Перспективу я бы взял из нейронки:

Перспективы развития структурного подхода:

- возможность быстрого прототипирования и применения алгоритмических языков четвёртого поколения, что позволит внедрить эволюционный подход.